climpred

Jun 08, 2020

Getting Started

1 Version 2.0.0 Release 3

2 Installation 5

Bibliography 157

Index 159

CHAPTER 1

Version 2.0.0 Release

We now support sub-annual (e.g., seasonal, monthly, weekly, daily) forecasts. We provide a host of deterministic and probabilistic metrics. We support both perfect-model and hindcast-based prediction ensembles, and provide `PerfectModelEnsemble` and `HindcastEnsemble` classes to make analysis easier.

See quick start and our examples to get started.

CHAPTER 2

Installation

You can install the latest release of `climpred` using `pip` or `conda`:

```
pip install climpred
```

```
conda install -c conda-forge climpred
```

You can also install the bleeding edge (pre-release versions) by cloning this repository and running `pip install . --upgrade` in the main directory

Getting Started

- *Overview: Why climpred?*
- *Scope of climpred*
- *Quick Start*
- *Examples*

2.1 Overview: Why climpred?

There are many packages out there related to computing metrics on initialized geoscience predictions. However, we didn't find any one package that unified all our needs.

Output from earth system prediction hindcast (also called re-forecast) experiments is difficult to work with. A typical output file could contain the dimensions `initialization`, `lead time`, `ensemble member`, `latitude`, `longitude`, `depth`. `climpred` leverages the labeled dimensions of `xarray` to handle the headache of bookkeeping for you. We offer *`HindcastEnsemble`* and *`PerfectModelEnsemble`* objects that carry products to verify against (e.g., control runs, reconstructions, uninitialized ensembles) along with your decadal prediction output.

When computing lead-dependent skill scores, `climpred` handles all of the lag-correlating for you, properly aligning the multiple time dimensions between the hindcast and verification datasets. We offer a suite of vectorized deterministic and probabilistic metrics that can be applied to time series and grids. It's as easy as adding your decadal prediction output to an object and running compute: `HindcastEnsemble.verify(metric='rmse')`.

2.2 Scope of climpred

climpred aims to be the primary package used to analyze output from initialized dynamical forecast models, ranging from short-term weather forecasts to decadal climate forecasts. The code base will be driven entirely by the geoscientific prediction community through open source development. It leverages xarray to keep track of core prediction ensemble dimensions (e.g., ensemble member, initialization date, and lead time) and dask to perform out-of-memory computations on large datasets.

The primary goal of climpred is to offer a comprehensive set of analysis tools for assessing the forecasts relative to a validation product (e.g., observations, reanalysis products, control runs, baseline forecasts). This will range from simple deterministic and probabilistic verification metrics—such as mean absolute error and various skill scores—to more advanced analysis methods, such as relative entropy and mutual information. climpred expects users to handle their domain-specific post-processing of model output, so that the package can focus on the actual analysis of forecasts.

Finally, the climpred documentation will serve as a repository of unified analysis methods through jupyter notebook examples, and will also collect relevant references and literature.

2.3 Quick Start

The easiest way to get up and running is to load in one of our example datasets (or load in some data of your own) and to convert them to either a HindcastEnsemble or PerfectModelEnsemble object.

climpred provides example datasets from the MPI-ESM-LR decadal prediction ensemble and the CESM decadal prediction ensemble. See our examples to see some analysis cases.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import xarray as xr

from climpred import HindcastEnsemble
from climpred.tutorial import load_dataset
import climpred
```

You can view the datasets available to be loaded with the load_datasets() command without passing any arguments:

[2]:
```
load_dataset()
```

```
'MPI-control-1D': area averages for the MPI control run of SST/SSS.
'MPI-control-3D': lat/lon/time for the MPI control run of SST/SSS.
'MPI-PM-DP-1D': perfect model decadal prediction ensemble area averages of SST/SSS/
↪AMO.
'MPI-PM-DP-3D': perfect model decadal prediction ensemble lat/lon/time of SST/SSS/AMO.
'CESM-DP-SST': hindcast decadal prediction ensemble of global mean SSTs.
'CESM-DP-SSS': hindcast decadal prediction ensemble of global mean SSS.
'CESM-DP-SST-3D': hindcast decadal prediction ensemble of eastern Pacific SSTs.
'CESM-LE': uninitialized ensemble of global mean SSTs.
'MPIESM_miklip_baseline1-hind-SST-global': hindcast initialized ensemble of global␣
↪mean SSTs
'MPIESM_miklip_baseline1-hist-SST-global': uninitialized ensemble of global mean SSTs
'MPIESM_miklip_baseline1-assim-SST-global': assimilation in MPI-ESM of global mean␣
↪SSTs
'ERSST': observations of global mean SSTs.
'FOSI-SST': reconstruction of global mean SSTs.
'FOSI-SSS': reconstruction of global mean SSS.
'FOSI-SST-3D': reconstruction of eastern Pacific SSTs
```

(continues on next page)

(continued from previous page)

```
'GMAO-GEOS-RMM1': daily RMM1 from the GMAO-GEOS-V2p1 model for SubX
'RMM-INTERANN-OBS': observed RMM with interannual variablity included
```

From here, loading a dataset is easy. Note that you need to be connected to the internet for this to work – the datasets are being pulled from the climpred-data repository. Once loaded, it is cached on your computer so you can reload extremely quickly. These datasets are very small (< 1MB each) so they won't take up much space.

[3]:
```
hind = climpred.tutorial.load_dataset('CESM-DP-SST')
# Add lead attribute units.
hind["lead"].attrs["units"] = "years"
obs = climpred.tutorial.load_dataset('ERSST')
```

Make sure your prediction ensemble's dimension labeling conforms to `climpred`'s standards. In other words, you need an `init`, `lead`, and (optional) `member` dimension. Make sure that your `init` and `lead` dimensions align. *E.g.*, a November 1st, 1954 initialization should be labeled as `init=1954` so that the lead=1 forecast is 1955.

[4]:
```
print(hind)
```

```
<xarray.Dataset>
Dimensions:  (init: 64, lead: 10, member: 10)
Coordinates:
  * lead     (lead) int32 1 2 3 4 5 6 7 8 9 10
  * member   (member) int32 1 2 3 4 5 6 7 8 9 10
  * init     (init) float32 1954.0 1955.0 1956.0 1957.0 ... 2015.0 2016.0 2017.0
Data variables:
    SST      (init, lead, member) float64 ...
```

We'll quickly process the data to create anomalies. CESM-DPLE's drift-correction occurs over 1964-2014, so we'll remove that from the observations.

[5]:
```
# subtract climatology
obs = obs - obs.sel(time=slice(1964, 2014)).mean()
```

We'll also remove a linear trend so that it doesn't artificially boost our predictability.

[6]:
```
hind = climpred.stats.rm_trend(hind, dim='init')
obs = climpred.stats.rm_trend(obs, dim='time')
```

We have to add the lead attributes back on, because `xarray` sometimes drops attributes. This is a bug we're aware of that we are working on fixing for `climpred`.

[7]:
```
# Add lead attribute units.
hind["lead"].attrs["units"] = "years"
```

We can now create a HindcastEnsemble object and add our observations and name them `'Obs'`.

[8]:
```
hindcast = HindcastEnsemble(hind)
hindcast = hindcast.add_observations(obs, 'Obs')
print(hindcast)
```

```
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST      (init, lead, member) float64 0.005165 0.03014 ... 0.1842 0.1812
Obs:
    SST      (time) float32 -0.061960407 -0.023283795 ... 0.072058104 0.165859
Uninitialized:
    None
```

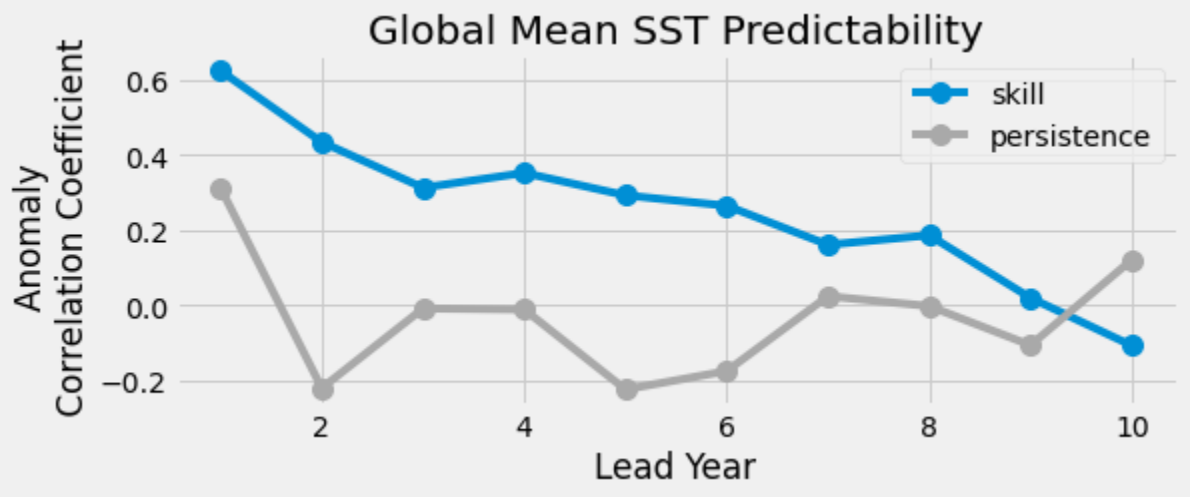
```
/Users/ribr5703/miniconda3/envs/climpred-dev/lib/python3.6/site-packages/climpred/
↪utils.py:141: UserWarning: Assuming annual resolution due to numeric inits. Change␣
↪init to a datetime if it is another resolution.
  'Assuming annual resolution due to numeric inits. '
```

Now we'll quickly calculate skill and persistence. We have a variety of possible metrics to use.

[9]:
```
result = hindcast.verify(metric='acc', reference='persistence')
skill = result.sel(skill='init')
persistence = result.sel(skill='persistence')
print(skill)
```

```
<xarray.Dataset>
Dimensions:  (lead: 10)
Coordinates:
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
    skill    <U11 'init'
Data variables:
    SST      (lead) float64 0.6253 0.4349 0.3131 ... 0.1861 0.01881 -0.1084
```

[10]:
```
plt.style.use('fivethirtyeight')
f, ax = plt.subplots(figsize=(8, 3))
skill.SST.plot(marker='o', markersize=10, label='skill')
persistence.SST.plot(marker='o', markersize=10, label='persistence',
                     color='#a9a9a9')
plt.legend()
ax.set(title='Global Mean SST Predictability',
       ylabel='Anomaly \n Correlation Coefficient',
       xlabel='Lead Year')
plt.show()
```



We can also check error in our forecasts.

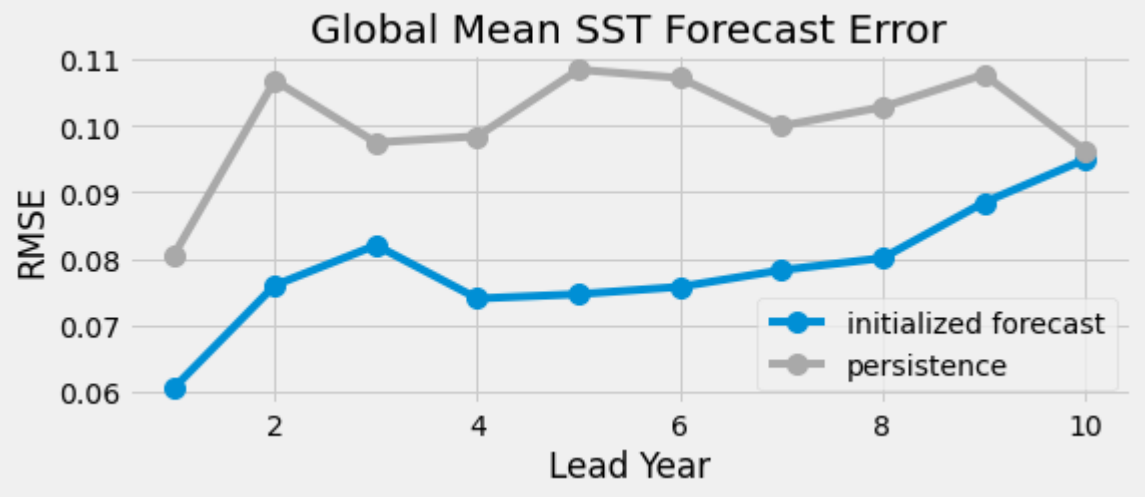
[12]:
```
result = hindcast.verify(metric='rmse', reference='persistence')
skill = result.sel(skill='init')
persistence = result.sel(skill='persistence')
```

[13]:
```
plt.style.use('fivethirtyeight')
f, ax = plt.subplots(figsize=(8, 3))
```

(continues on next page)

(continued from previous page)

```
skill.SST.plot(marker='o', markersize=10, label='initialized forecast')
persistence.SST.plot(marker='o', markersize=10, label='persistence',
                     color='#a9a9a9')
plt.legend()
ax.set(title='Global Mean SST Forecast Error',
       ylabel='RMSE',
       xlabel='Lead Year')
plt.show()
```



2.4 Examples

2.4.1 Dask

Using `dask` with `climpred`

This demo demonstrates `climpred`'s capabilities with `dask` https://docs.dask.org/en/latest/array.html. This enables enables out-of-memory and parallel computation for large datasets with `climpred`.

[1]:
```
import warnings

%matplotlib inline
import numpy as np
import xarray as xr
import dask
import climpred

warnings.filterwarnings("ignore")
```

- Load a `Client` to use `dask.distributed`: https://stackoverflow.com/questions/51099685/best-practices-in-setting-number-of-dask-workers
- (Optionally) Use the `dask` dashboard to visualize performance: https://github.com/dask/dask-labextension

[11]:
```
from dask.distributed import Client
import multiprocessing
ncpu = multiprocessing.cpu_count()
```

(continues on next page)

(continued from previous page)

```
processes = False
nworker = 8
threads = ncpu // nworker
print(
    f"Number of CPUs: {ncpu}, number of threads: {threads}, number of workers:
↪{nworker}, processes: {processes}",
)
client = Client(
    processes=processes,
    threads_per_worker=threads,
    n_workers=nworker,
    memory_limit="64GB",
)
client
```

```
Number of CPUs: 48, number of threads: 6, number of workers: 8, processes: False
```

[11]:
```
<Client: 'inproc://10.50.36.67/15942/1' processes=8 threads=48, memory=512.00 GB>
```

Load data

[12]:
```
# generic
ny, nx = 256, 220
nl, ni, nm = 20, 12, 10
ds = xr.DataArray(np.random.random((nl, ni, nm, ny, nx)), dims=('lead', 'init',
↪'member', 'y', 'x'))
ds['init'] = np.arange(3000, 3300, 300 // ni)
ds['lead'] = np.arange(1,1+ds.lead.size)
control = xr.DataArray(np.random.random((300, ny, nx)),dims=('time', 'y', 'x'))
control['time'] = np.arange(3000, 3300)
```

compute skill with `compute_perfect_model`

[13]:
```
kw = {'comparison':'m2e', 'metric':'rmse'}
```

compute skill without `dask`

[14]:
```
%time s = climpred.prediction.compute_perfect_model(ds, control, **kw)
```

```
CPU times: user 10.9 s, sys: 10.9 s, total: 21.8 s
Wall time: 19.9 s
```

- 2 core Mac Book Pro 2018: CPU times: user 11.5 s, sys: 6.88 s, total: 18.4 s Wall time: 19.6 s
- 24 core mistral node: CPU times: user 9.22 s, sys: 10.3 s, total: 19.6 s Wall time: 19.5 s

compute skill with `dask`

In order to use `dask` efficient, we need to chunk the data appropriately. Processing chunks of data lazily with `dask` creates a tiny overhead per dask, therefore chunking mostly makes sense when applying it to large data.

[26]:
```
chunked_dim = 'y'
chunks = {chunked_dim:ds[chunked_dim].size // nworker}
ds = ds.chunk(chunks)
# if memory allows
ds = ds.persist()
ds.data
```

[26]:
```
dask.array<rechunk-merge, shape=(20, 12, 10, 256, 220), dtype=float64, chunksize=(20, 12, 10, 32, 220), chunktype=numpy.ndarray>
```

[27]:
```
%%time
s_chunked = climpred.prediction.compute_perfect_model(ds, control, **kw)
assert dask.is_dask_collection(s_chunked)
s_chunked = s_chunked.compute()
```

```
CPU times: user 25.5 s, sys: 1min 21s, total: 1min 46s
Wall time: 5.42 s
```

- 2 core Mac Book Pro 2018: CPU times: user 2min 35s, sys: 1min 4s, total: 3min 40s Wall time: 2min 10s
- 24 core mistral node: CPU times: user 26.2 s, sys: 1min 37s, total: 2min 3s Wall time: 5.38 s

[28]:
```
try:
    xr.testing.assert_allclose(s,s_chunked,atol=1e-6)
except AssertionError:
    for v in s.data_vars:
        (s-s_chunked)[v].plot(robust=True, col='lead')
```

- The results `s` and `s_chunked` are identical as requested.
- Chunking reduces Wall time from 20s to 5s on supercomputer.

---

bootstrap skill with `bootstrap_perfect_model`

This speedup translates into `bootstrap_perfect_model`, where bootstrapped resamplings of intializialized, uninitialized and persistence skill are computed and then translated into p values and confidence intervals.

[29]:
```
kwp = kw.copy()
kwp['iterations'] = 4
```

bootstrap skill without `dask`

[30]:
```
ds = ds.compute()
control = control.compute()
```

[31]:
```
%time s_p = climpred.bootstrap.bootstrap_perfect_model(ds, control, **kwp)
```

```
CPU times: user 1min 51s, sys: 1min 54s, total: 3min 45s
Wall time: 3min 25s
```

- 2 core Mac Book Pro 2018: CPU times: user 2min 3s, sys: 1min 22s, total: 3min 26s Wall time: 3min 43s
- 24 core mistral node: CPU times: user 1min 51s, sys: 1min 54s, total: 3min 45s Wall time: 3min 25s

bootstrap skill with `dask`

When `ds` is chunked, `bootstrap_perfect_model` performs all skill calculations on resampled inputs in parallel.

[32]:
```
chunked_dim = 'y'
chunks = {chunked_dim:ds[chunked_dim].size // nworker}
ds = ds.chunk(chunks)
# if memory allows
ds = ds.persist()
ds.data
```

[32]:
```
dask.array<xarray-<this-array>, shape=(20, 12, 10, 256, 220), dtype=float64,␣
↪chunksize=(20, 12, 10, 32, 220), chunktype=numpy.ndarray>
```

[33]:
```
%time s_p_chunked = climpred.bootstrap.bootstrap_perfect_model(ds, control, **kwp)
```

```
CPU times: user 2min 55s, sys: 8min 8s, total: 11min 3s
Wall time: 1min 53s
```

- 2 core Mac Book Pro 2018: CPU times: user 2min 35s, sys: 1min 4s, total: 3min 40s Wall time: 2min 10s
- 24 core mistral node: CPU times: user 2min 55s, sys: 8min 8s, total: 11min 3s Wall time: 1min 53s

[ ]:
```
```

2.4.2 Pre-Processing

Setting up your own output

This demo demonstrates how you can setup your raw model output with `climpred.preprocessing` to match `climpred`'s expectations.

[1]:
```
from dask.distributed import Client
import multiprocessing
ncpu = multiprocessing.cpu_count()
threads = 6
nworker = ncpu//threads
print(f'Number of CPUs: {ncpu}, number of threads: {threads}, number of workers:
↪{nworker}')
```

```
Number of CPUs: 48, number of threads: 6, number of workers: 8
```

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import climpred
```

[3]:
```
from climpred.preprocessing.shared import load_hindcast, set_integer_time_axis
from climpred.preprocessing.mpi import get_path
```

Assuming your raw model output is stored in multiple files per member and initialization, `load_hindcast` is a nice wrapper function based on `get_path` designed for the output format of `MPI-ESM` to aggregated all hindcast output into one file as expected by `climpred`.

The basic idea is to look over the output of all members and concatinate, then loop over all initializations and concatinate. Before concatination, it is important to make the `time` dimension identical in all input datasets for concatination.

To reduce the data size, use the `preprocess` function provided to `xr.open_mfdataset` wisely in combination with `set_integer_axis`, e.g. additionally extracting only a certain region, time-step, time-aggregation or only few variables for a multi-variable input file as in MPI-ESM standard output.

[4]:
```
# check the code of load_hindcast
# load_hindcast??
```

[5]:
```
v = "global_primary_production"

def preprocess_1var(ds, v=v):
    """Only leave one variable `v` in dataset """
    return ds[v].to_dataset(name=v).squeeze()
```

[6]:
```
# lead_offset because yearmean output
%time ds = load_hindcast(inits=range(1961, 1965), members=range(1, 3),␣
↪preprocess=preprocess_1var, get_path=get_path)
```

```
Processing init 1961 ...
Processing init 1962 ...
Processing init 1963 ...
Processing init 1964 ...
CPU times: user 5.07 s, sys: 2.06 s, total: 7.13 s
Wall time: 5.19 s
```

[7]:
```
# what we need for climpred
ds.coords
```

[7]:
```
Coordinates:
    depth    float64 0.0
    lat      float64 0.0
    lon      float64 0.0
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
  * member   (member) int64 1 2
  * init     (init) int64 1961 1962 1963 1964
```

[8]:
```
ds[v].data
```

[8]:
```
dask.array<concatenate, shape=(4, 2, 10), dtype=float32, chunksize=(1, 1, 1),␣
↪chunktype=numpy.ndarray>
```

[9]:
```
# loading the data into memory
# if not rechunk
%time ds = ds.load()
```

```
CPU times: user 216 ms, sys: 44 ms, total: 260 ms
Wall time: 220 ms
```

[10]:
```
# you may actually want to use `compute_hindcast` to calculate skill from hindcast.
# for this you also need an `observation` to compare to
# here `compute_perfect_model` compares one member to the ensemble mean of the remain␣
↪members in turn
climpred.prediction.compute_perfect_model(ds, ds.rename({'lead':'time'}))
```

[10]:
```
<xarray.Dataset>
Dimensions:                    (lead: 10)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
    depth                       float64 0.0
    lon                         float64 0.0
    lat                         float64 0.0
  * lead                        (lead) int64 1 2 3 4 5 6 7 8 9 10
Data variables:
    global_primary_production   (lead) float64 -0.1276 0.593 ... 0.09196 0.5038
Attributes:
    prediction_skill:               calculated by climpred https://climpred.re...
    skill_calculated_by_function:   compute_perfect_model
    number_of_initializations:      4
    number_of_members:              2
    metric:                         pearson_r
    comparison:                     m2e
    dim:                            ['init', 'member']
    units:                          None
    created:                        2020-02-07 18:27:28
```

`intake-esm` for cmorized output

In case you have access to cmorized output of CMIP experiments, consider using `intake-esm`. With the `preprocess` function you can align the `time` dimension of all input files. Finally, `rename_to_climpred_dims` only renames.

[11]:
```
from climpred.preprocessing.shared import rename_to_climpred_dims, set_integer_time_
↪axis
```

[12]:
```
# make to have intake-esm installed
import intake # this is enough for intake-esm to work
```

[13]:
```
# https://github.com/NCAR/intake-esm-datastore/
col_url = "/home/mpim/m300524/intake-esm-datastore/catalogs/mistral-cmip6.json"
col = intake.open_esm_datastore(col_url)
```

[14]:
```
col.df.columns
```

[14]:
```
Index(['activity_id', 'institution_id', 'source_id', 'experiment_id',
       'member_id', 'table_id', 'variable_id', 'grid_label', 'dcpp_init_year',
       'version', 'time_range', 'path'],
      dtype='object')
```

[15]:
```
# load 2 members for 2 inits for one variable from one model
query = dict(experiment_id=[
    'dcppA-hindcast'], table_id='Amon', member_id=['r1i1p1f1', 'r2i1p1f1'], dcpp_init_
↪year=[1970, 1971],
    variable_id='tas', source_id='MPI-ESM1-2-HR')
cat = col.search(**query)
cdf_kwargs = {'chunks': {'time': 12}, 'decode_times': False}
```

[16]:
```
cat.df.head()
```

[16]:
```
  activity_id institution_id      source_id   experiment_id member_id  \
0        DCPP          MPI-M  MPI-ESM1-2-HR  dcppA-hindcast  r1i1p1f1
1        DCPP          MPI-M  MPI-ESM1-2-HR  dcppA-hindcast  r1i1p1f1
```

(continues on next page)

(continued from previous page)

```
2        DCPP          MPI-M  MPI-ESM1-2-HR  dcppA-hindcast  r2i1p1f1
3        DCPP          MPI-M  MPI-ESM1-2-HR  dcppA-hindcast  r2i1p1f1

  table_id variable_id grid_label  dcpp_init_year    version     time_range  \
0     Amon         tas         gn          1971.0  v20190906  197111-198112
1     Amon         tas         gn          1970.0  v20190906  197011-198012
2     Amon         tas         gn          1971.0  v20190906  197111-198112
3     Amon         tas         gn          1970.0  v20190906  197011-198012

                                                 path
0  /work/ik1017/CMIP6/data/CMIP6/DCPP/MPI-M/MPI-E...
1  /work/ik1017/CMIP6/data/CMIP6/DCPP/MPI-M/MPI-E...
2  /work/ik1017/CMIP6/data/CMIP6/DCPP/MPI-M/MPI-E...
3  /work/ik1017/CMIP6/data/CMIP6/DCPP/MPI-M/MPI-E...
```

[17]:
```
def preprocess(ds):
    # extract tiny spatial and temporal subset to make this fast
    ds = ds.isel(lon=[50, 51, 52], lat=[50, 51, 52],
                 time=np.arange(12 * 2))
    # make time dim identical
    ds = set_integer_time_axis(ds,time_dim='time')
    return ds
```

[18]:
```
dset_dict = cat.to_dataset_dict(
    cdf_kwargs=cdf_kwargs, preprocess=preprocess)
```

```
Progress: || 100.0%

--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'

--> There are 1 group(s)
```

[19]:
```
# get first dict value
_, ds = dset_dict.popitem()
ds.coords
```

[19]:
```
Coordinates:
    height          float64 ...
  * dcpp_init_year  (dcpp_init_year) float64 1.97e+03 1.971e+03
  * lat             (lat) float64 -42.55 -41.61 -40.68
  * time            (time) int64 1 2 3 4 5 6 7 8 9 ... 17 18 19 20 21 22 23 24
  * lon             (lon) float64 46.88 47.81 48.75
  * member_id       (member_id) <U8 'r1i1p1f1' 'r2i1p1f1'
```

[20]:
```
# rename to comply with climpred's required dimension names
ds = rename_to_climpred_dims(ds)
```

[21]:
```
# what we need for climpred
ds.coords
```

[21]:
```
Coordinates:
    height   float64 ...
  * init     (init) float64 1.97e+03 1.971e+03
  * lat      (lat) float64 -42.55 -41.61 -40.68
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10 ... 15 16 17 18 19 20 21 22 23 24
```

(continues on next page)

(continued from previous page)

```
  * lon      (lon) float64 46.88 47.81 48.75
  * member   (member) <U8 'r1i1p1f1' 'r2i1p1f1'
```

[22]:
```
ds['tas'].data
```

[22]:
```
dask.array<concatenate, shape=(2, 2, 24, 3, 3), dtype=float32, chunksize=(1, 1, 12, 3,
↪ 3), chunktype=numpy.ndarray>
```

[23]:
```
# loading the data into memory
# if not rechunk
# this is here quite fast before we only select 9 grid cells
%time ds = ds.load()
```

```
CPU times: user 218 ms, sys: 74 ms, total: 292 ms
Wall time: 237 ms
```

[24]:
```
# you may actually want to use `compute_hindcast` to calculate skill from hindcast.
# for this you also need an `observation` to compare to
# here `compute_perfect_model` compares one member to the ensemble mean of the remain␣
↪members in turn
climpred.prediction.compute_perfect_model(ds, ds.rename({'lead':'time'}))
```

```
/work/mh0727/m300524/miniconda3/envs/climpred-dev/lib/python3.6/site-packages/
↪xskillscore/core/np_deterministic.py:182: RuntimeWarning: invalid value encountered␣
↪in true_divide
  r = r_num / r_den
```

[24]:
```
<xarray.Dataset>
Dimensions:    (bnds: 2, lat: 3, lead: 24, lon: 3)
Coordinates:
    height     float64 2.0
  * lat        (lat) float64 -42.55 -41.61 -40.68
  * lead       (lead) int64 1 2 3 4 5 6 7 8 9 10 ... 16 17 18 19 20 21 22 23 24
  * lon        (lon) float64 46.88 47.81 48.75
Dimensions without coordinates: bnds
Data variables:
    lat_bnds   (lead, lat, bnds) float64 nan nan nan nan nan ... nan nan nan nan
    time_bnds  (lead, bnds) float64 nan nan nan nan nan ... nan nan nan nan nan
    lon_bnds   (lead, lon, bnds) float64 nan nan nan nan nan ... nan nan nan nan
    tas        (lead, lat, lon) float32 0.933643 0.88438606 ... -0.8581106
Attributes:
    nominal_resolution:           100 km
    institution:                  Max Planck Institute for Meteorology, Hamb...
    experiment_id:                dcppA-hindcast
    variable_id:                  tas
    experiment:                   hindcast initialized based on observations...
    tracking_id:                  hdl:21.14100/f41d2fe5-bb1c-49d0-8afa-0cb9e...
    table_info:                   Creation Date:(09 May 2019) MD5:e6ef8ececc...
    references:                   MPI-ESM: Mauritsen, T. et al. (2019), Deve...
    frequency:                    mon
    source_id:                    MPI-ESM1-2-HR
    physics_index:                1
    initialization_index:         1
    product:                      model-output
    source_type:                  AOGCM
    title:                        MPI-ESM1-2-HR output prepared for CMIP6
    institution_id:               MPI-M
```

(continues on next page)

(continued from previous page)

```
    project_id:                     CMIP6
    Conventions:                    CF-1.7 CMIP-6.2
    intake_esm_varname:             tas
    activity_id:                    DCPP
    branch_method:                  standard lagged initialization
    table_id:                       Amon
    data_specs_version:             01.00.30
    grid_label:                     gn
    forcing_index:                  1
    external_variables:             areacella
    cmor_version:                   3.5.0
    realm:                          atmos
    history:                        2019-09-06T14:20:04Z ; CMOR rewrote data t...
    grid:                           spectral T127; 384 x 192 longitude/latitude
    license:                        CMIP6 model data produced by MPI-M is lice...
    source:                         MPI-ESM1.2-HR (2017): \naerosol: none, pre...
    mip_era:                        CMIP6
    contact:                        cmip6-mpi-esm@dkrz.de
    prediction_skill:               calculated by climpred https://climpred.re...
    skill_calculated_by_function:   compute_perfect_model
    number_of_initializations:      2
    number_of_members:              2
    metric:                         pearson_r
    comparison:                     m2e
    dim:                            ['init', 'member']
    units:                          None
    created:                        2020-02-07 18:27:51
```

2.4.3 Subseasonal

Calculate the skill of a MJO Index as a function of lead time

In this example, we demonstrate:

1. How to remotely access data from the Subseasonal Experiment (SubX) hindcast database and set it up to be used in `climpred`.
2. How to calculate the Anomaly Correlation Coefficient (ACC) using daily data with `climpred`
3. How to calculate and plot historical forecast skill of the real-time multivariate MJO (RMM) indices as function of lead time.

The Subseasonal Experiment (SubX)

Further information on SubX is available from Pegion et al. 2019 and the SubX project website

The SubX public database is hosted on the International Research Institute for Climate and Society (IRI) data server http://iridl.ldeo.columbia.edu/SOURCES/.Models/.SubX/

Since the SubX data server is accessed via this notebook, the time for the notebook to run may is several minutes and will vary depending on the speed that data can be downloaded. This is a large dataset, so please be patient. If you prefer to download SubX data locally, scripts are available from https://github.com/kpegion/SubX.

Definitions

RMM Two indices (RMM1 and RMM2) are used to represent the MJO. Together they define the MJO based on 8 phases and can represent both the phase and amplitude of the MJO (Wheeler and Hendon 2004). This example uses the observed RMM1 provided by Matthew Wheeler at the Center for Australian Weather and Climate Research. It is the version of the indices in which interannual variability has not been removed.

Skill of RMM Traditionally, the skill of the RMM is calculated as a bivariate correlation encompassing the skill of the two indices together (Rashid et al. 2010; Gottschalck et al. 2010). Currently, `climpred` does not have the functionality to calculate the bivariate correlation, thus the anomaly correlation coefficient for RMM1 index is calculated here as a demonstration. The bivariate correlation metric will be added in a future version of `climpred`

[1]:
```
import warnings

import matplotlib.pyplot as plt
plt.style.use('ggplot')
plt.style.use('seaborn-talk')

import xarray as xr
import pandas as pd
import numpy as np

from climpred import HindcastEnsemble
import climpred
```

[2]:
```
warnings.filterwarnings("ignore")
```

Function to set 360 calendar to 360_day calendar and decond cf times

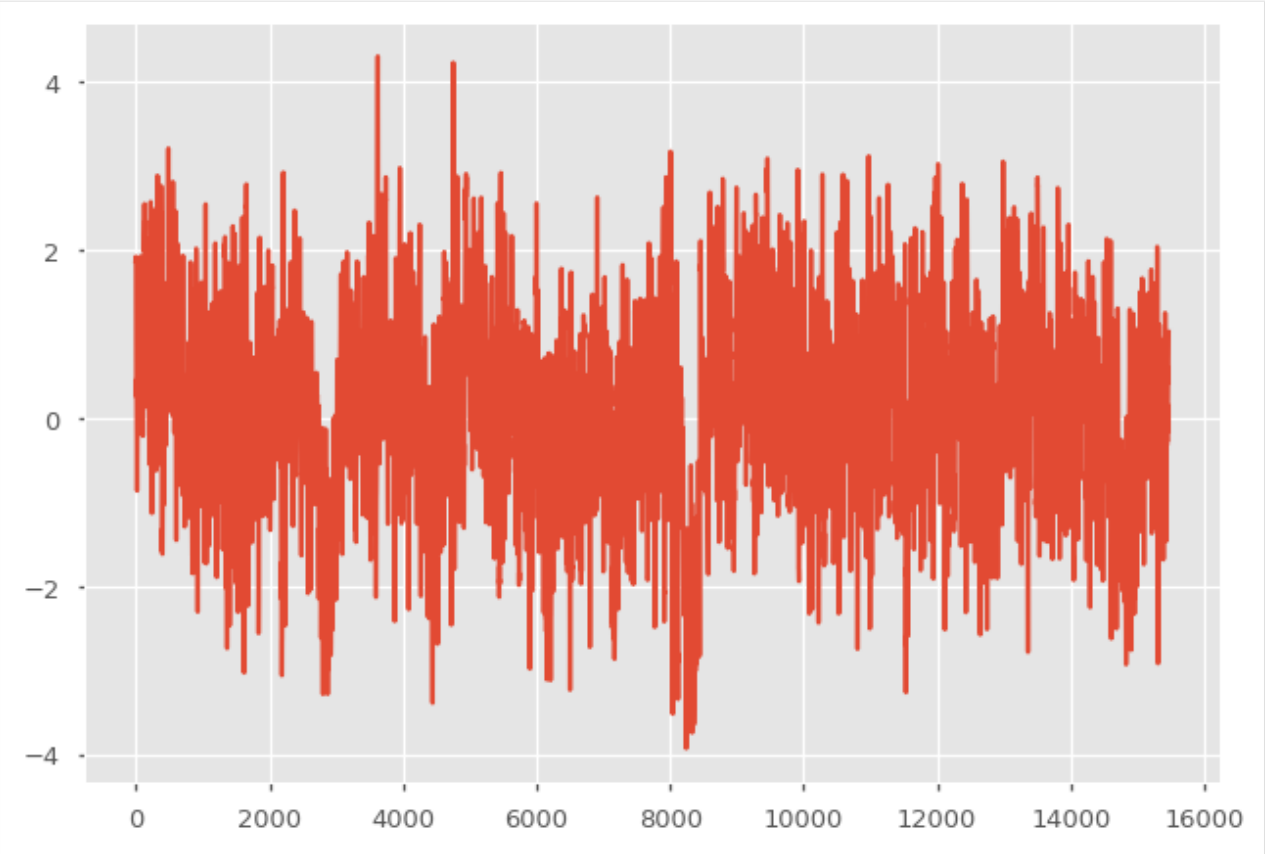
[3]:
```
def decode_cf(ds, time_var):
    if ds[time_var].attrs['calendar'] == '360':
        ds[time_var].attrs['calendar'] = '360_day'
    ds = xr.decode_cf(ds, decode_times=True)
    return ds
```

Read the observed RMM Indices

[4]:
```
obsds = climpred.tutorial.load_dataset('RMM-INTERANN-OBS')['rmm1'].to_dataset()
obsds = obsds.dropna('time') # Get rid of missing times.
```

[5]:
```
plt.plot(obsds['rmm1'])
```

[5]:
```
[<matplotlib.lines.Line2D at 0x1263a64e0>]
```



Read the SubX RMM1 data for the GMAO-GEOS_V2p1 model from the SubX data server. It is important to note that the SubX data contains weekly initialized forecasts where the `init` day varies by model. SubX data may have all NaNs for initial dates in which a model does not make a forecast, thus we apply `dropna` over the `S=init` dimension when `how=all` data for a given `S=init` is missing. This can be slow, but allows the rest of the calculations to go more quickly.

Note that we ran the `dropna` operation offline and then uploaded the post-processed SubX dataset to the `climpred-data` repo for the purposes of this demo. This is how you can do this manually:

```
url = 'http://iridl.ldeo.columbia.edu/SOURCES/.Models/.SubX/.GMAO/.GEOS_V2p1/.
↪hindcast/.RMM/.RMM1/dods/'
fcstds = xr.open_dataset(url, decode_times=False, chunks={'S': 1, 'L': 45}).
↪dropna(dim='S',how='all')
```

[6]:
```
fcstds = climpred.tutorial.load_dataset('GMAO-GEOS-RMM1', decode_times=False)
print(fcstds)
```

```
<xarray.Dataset>
Dimensions:  (L: 45, M: 4, S: 510)
Coordinates:
  * S        (S) float32 14245.0 14250.0 14255.0 ... 20439.0 20444.0 20449.0
  * M        (M) float32 1.0 2.0 3.0 4.0
  * L        (L) float32 0.5 1.5 2.5 3.5 4.5 5.5 ... 40.5 41.5 42.5 43.5 44.5
Data variables:
    RMM1     (S, M, L) float32 ...
Attributes:
    Conventions:  IRIDL
```

The SubX data dimensions correspond to the following `climpred` dimension definitions: `X=lon`,`L=lead`,`Y=lat`,`M=member`, `S=init`. We will rename the dimensions to their `climpred` names.

[7]:
```
fcstds=fcstds.rename({'S': 'init','L': 'lead','M': 'member', 'RMM1' : 'rmm1'})
```

Let's make sure that the `lead` dimension is set properly for `climpred`. SubX data stores `leads` as 0.5, 1.5, 2.5, etc, which correspond to 0, 1, 2, ... days since initialization. We will change the `lead` to be integers starting with zero.

[8]:
```
fcstds['lead']=(fcstds['lead']-0.5).astype('int')
```

Now we need to make sure that the `init` dimension is set properly for `climpred`. For daily data, the `init` dimension must be a `xr.CFTimeIndex` or a `pd.DatetimeIndex`. We convert the `init` values to `pd.DatetimeIndex`.

[9]:
```
fcstds = decode_cf(fcstds, 'init')
```

`climpred` also requires that `lead` dimension has an attribute called `units` indicating what time units the `lead` is assocated with. Options are: `years`,`seasons`,`months`,`weeks`,`pentads`,`days`. For the SubX data, the `lead` `units` are `days`.

[10]:
```
fcstds['lead'].attrs={'units': 'days'}
```

Create the `climpred HindcastEnsemble` object and add the observations.

[11]:
```
hindcast = HindcastEnsemble(fcstds)
```

[12]:
```
hindcast = hindcast.add_observations(obsds, 'obs')
```

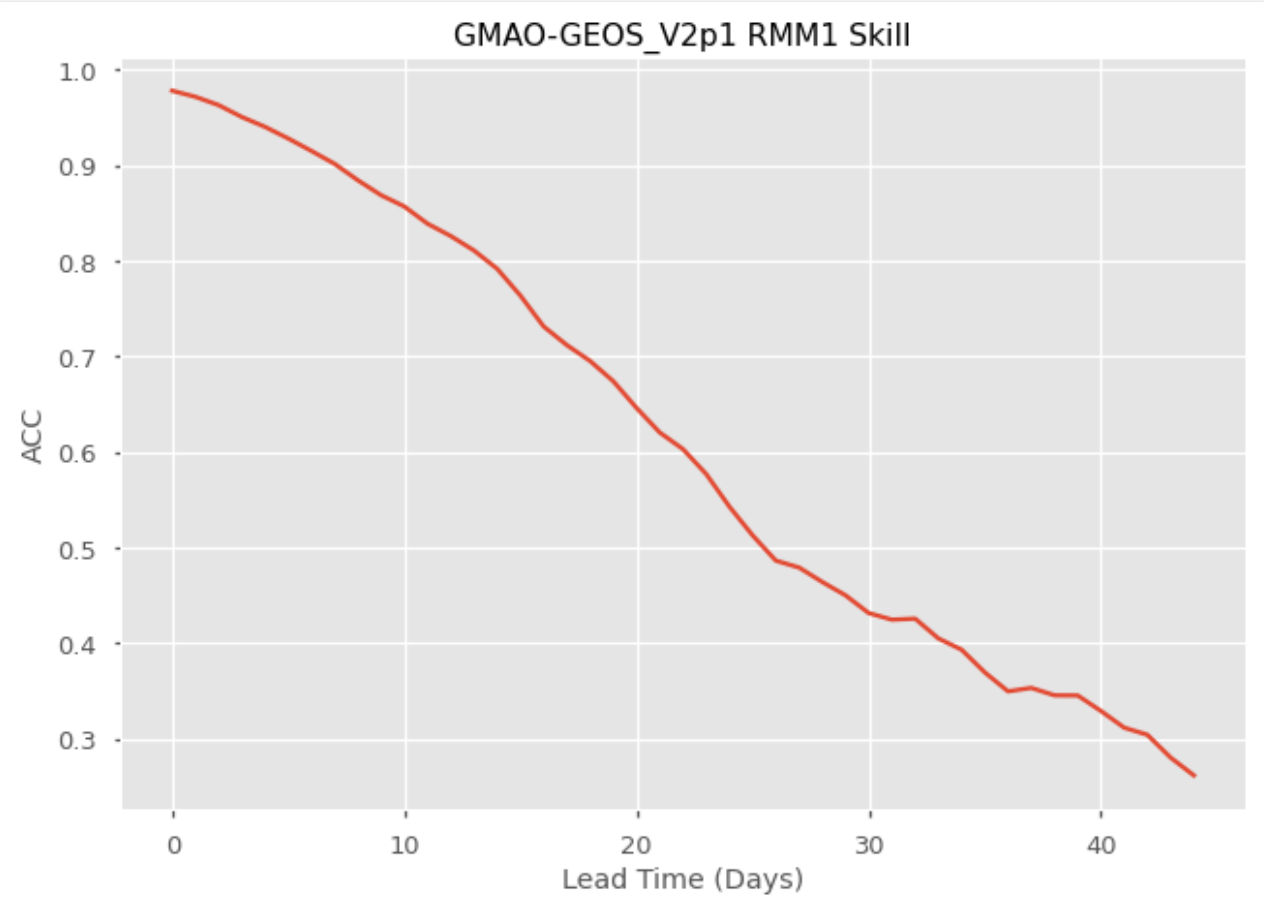
Calculate the Anomaly Correlation Coefficient (ACC)

[13]:
```
skill = hindcast.verify(metric='acc', alignment='maximize')
```

Plot the skill as a function of lead time

[14]:
```
plt.plot(skill['rmm1'])
plt.title('GMAO-GEOS_V2p1 RMM1 Skill')
plt.xlabel('Lead Time (Days)')
plt.ylabel('ACC')
```

[14]:
```
Text(0, 0.5, 'ACC')
```



References

1. Pegion, K., B.P. Kirtman, E. Becker, D.C. Collins, E. LaJoie, R. Burgman, R. Bell, T. DelSole, D. Min, Y. Zhu, W. Li, E. Sinsky, H. Guan, J. Gottschalck, E.J. Metzger, N.P. Barton, D. Achuthavarier, J. Marshak, R.D. Koster, H. Lin, N. Gagnon, M. Bell, M.K. Tippett, A.W. Robertson, S. Sun, S.G. Benjamin, B.W. Green, R. Bleck, and H. Kim, 2019: The Subseasonal Experiment (SubX): A Multimodel Subseasonal Prediction Experiment. Bull. Amer. Meteor. Soc., 100, 2043–2060, https://doi.org/10.1175/BAMS-D-18-0270.1

2. Kirtman, B. P., Pegion, K., DelSole, T., Tippett, M., Robertson, A. W., Bell, M., … Green, B. W. (2017). The Subseasonal Experiment (SubX) [Data set]. IRI Data Library. https://doi.org/10.7916/D8PG249H

3. Wheeler, M. C., & (null), H. H. (2004). An all-season real-time multivariate MJO index: Development of an index for monitoring and prediction. Monthly Weather Review, 132(8), 1917–1932. http://doi.org/10.1175/1520-0493(2004)132

4. Rashid, H. A., Hendon, H. H., Wheeler, M. C., & Alves, O. (2010). Prediction of the Madden–Julian oscillation with the POAMA dynamical prediction system. Climate Dynamics, 36(3-4), 649–661. http://doi.org/10.1007/s00382-010-0754-x

5. Gottschalck, J., Wheeler, M., Weickmann, K., Vitart, F., Savage, N., Lin, H., et al. (2010). A Framework for Assessing Operational Madden–Julian Oscillation Forecasts: A CLIVAR MJO Working Group Project. Bulletin of the American Meteorological Society, 91(9), 1247–1258. http://doi.org/10.1175/2010BAMS2816.1

Calculate the skill of a MJO Index as a function of lead time for Weekly Data

In this example, we demonstrate:

1. How to remotely access data from the Subseasonal Experiment (SubX) hindcast database and set it up to be used in `climpred`.
2. How to calculate the Anomaly Correlation Coefficient (ACC) using weekly data with `climpred`
3. How to calculate and plot historical forecast skill of the real-time multivariate MJO (RMM) indices as function of lead time.

The Subseasonal Experiment (SubX)

Further information on SubX is available from Pegion et al. 2019 and the SubX project website

The SubX public database is hosted on the International Research Institute for Climate and Society (IRI) data server http://iridl.ldeo.columbia.edu/SOURCES/.Models/.SubX/

Since the SubX data server is accessed via this notebook, the time for the notebook to run may is several minutes and will vary depending on the speed that data can be downloaded. This is a large dataset, so please be patient. If you prefer to download SubX data locally, scripts are available from https://github.com/kpegion/SubX.

Definitions

RMM Two indices (RMM1 and RMM2) are used to represent the MJO. Together they define the MJO based on 8 phases and can represent both the phase and amplitude of the MJO (Wheeler and Hendon 2004). This example uses the observed RMM1 provided by Matthew Wheeler at the Center for Australian Weather and Climate Research. It is the version of the indices in which interannual variability has not been removed.

Skill of RMM Traditionally, the skill of the RMM is calculated as a bivariate correlation encompassing the skill of the two indices together (Rashid et al. 2010; Gottschalck et al 2010). Currently, `climpred` does not have the functionality to calculate the bivariate correlation, thus the anomaly correlation coefficient for RMM1 index is calculated here as a demonstration. The bivariate correlation metric will be added in a future version of `climpred`

[1]:
```
import warnings

import matplotlib.pyplot as plt
plt.style.use('ggplot')
plt.style.use('seaborn-talk')

import xarray as xr
import pandas as pd
import numpy as np

from climpred import HindcastEnsemble
import climpred
```

[2]:
```
warnings.filterwarnings("ignore")
```

Function to set 360 calendar to 360_day calendar and decode cf times

[3]:
```
def decode_cf(ds, time_var):
    if ds[time_var].attrs['calendar'] == '360':
        ds[time_var].attrs['calendar'] = '360_day'
    ds = xr.decode_cf(ds, decode_times=True)
    return ds
```

Read the observed RMM Indices

[4]:
```
obsds = climpred.tutorial.load_dataset('RMM-INTERANN-OBS')['rmm1'].to_dataset()
obsds
```

[4]:
```
<xarray.Dataset>
Dimensions:  (time: 15613)
Coordinates:
  * time     (time) datetime64[ns] 1974-06-03 1974-06-04 ... 2017-07-24
Data variables:
    rmm1     (time) float64 ...
```

Read the SubX RMM1 data for the GMAO-GEOS_V2p1 model from the SubX data server. It is important to note that the SubX data contains weekly initialized forecasts where the `init` day varies by model. SubX data may have all NaNs for initial dates in which a model does not make a forecast, thus we apply `dropna` over the `S=init` dimension when `how=all` data for a given `S=init` is missing. This can be slow, but allows the rest of the calculations to go more quickly.

Note that we ran the `dropna` operation offline and then uploaded the post-processed SubX dataset to the `climpred-data` repo for the purposes of this demo. This is how you can do this manually:

```
url = 'http://iridl.ldeo.columbia.edu/SOURCES/.Models/.SubX/.GMAO/.GEOS_V2p1/.
↪hindcast/.RMM/.RMM1/dods/'
fcstds = xr.open_dataset(url, decode_times=False, chunks={'S': 1, 'L': 45}).
↪dropna(dim='S',how='all')
```

[5]:
```
fcstds = climpred.tutorial.load_dataset('GMAO-GEOS-RMM1', decode_times=False)
print(fcstds)
```

```
<xarray.Dataset>
Dimensions:  (L: 45, M: 4, S: 510)
Coordinates:
  * S        (S) float32 14245.0 14250.0 14255.0 ... 20439.0 20444.0 20449.0
  * M        (M) float32 1.0 2.0 3.0 4.0
  * L        (L) float32 0.5 1.5 2.5 3.5 4.5 5.5 ... 40.5 41.5 42.5 43.5 44.5
Data variables:
    RMM1     (S, M, L) float32 ...
Attributes:
    Conventions:  IRIDL
```

The SubX data dimensions correspond to the following `climpred` dimension definitions: `X=lon`,`L=lead`,`Y=lat`,`M=member`, `S=init`. We will rename the dimensions to their `climpred` names.

[6]:
```
fcstds=fcstds.rename({'S': 'init','L': 'lead','M': 'member', 'RMM1' : 'rmm1'})
```

Let's make sure that the `lead` dimension is set properly for `climpred`. SubX data stores `leads` as 0.5, 1.5, 2.5, etc, which correspond to 0, 1, 2, ... days since initialization. We will change the `lead` to be integers starting with zero.

[7]:
```
fcstds['lead']=(fcstds['lead']-0.5).astype('int')
```

Now we need to make sure that the `init` dimension is set properly for `climpred`. For daily data, the `init` dimension must be a `xr.cfdateTimeIndex` or a `pd.datetimeIndex`. We convert the `init` values to `pd.datatimeIndex`.

[8]:
```
fcstds=decode_cf(fcstds,'init')
fcstds['init']=pd.to_datetime(fcstds.init.values.astype(str))
fcstds['init']=pd.to_datetime(fcstds['init'].dt.strftime('%Y%m%d 00:00'))
```

Make Weekly Averages

[9]:
```
fcstweekly=fcstds.rolling(lead=7,center=False).mean().dropna(dim='lead')
obsweekly=obsds.rolling(time=7,center=False).mean().dropna(dim='time')
print(fcstweekly)
print(obsweekly)
```

```
<xarray.Dataset>
Dimensions:  (init: 510, lead: 39, member: 4)
Coordinates:
  * init     (init) datetime64[ns] 1999-01-01 1999-01-06 ... 2015-12-27
  * member   (member) float32 1.0 2.0 3.0 4.0
  * lead     (lead) int64 6 7 8 9 10 11 12 13 14 ... 36 37 38 39 40 41 42 43 44
Data variables:
    rmm1     (init, member, lead) float32 0.13466184 0.10946906 ... 0.076061934
<xarray.Dataset>
Dimensions:  (time: 15456)
Coordinates:
  * time     (time) datetime64[ns] 1974-06-09 1974-06-10 ... 2017-07-24
Data variables:
    rmm1     (time) float64 1.336 1.107 0.9046 0.695 ... 0.6265 0.673 0.7352
```

Create a new `xr.DataArray` for the weekly fcst data

[10]:
```
nleads=fcstweekly['lead'][::7].size
fcstweeklyda=xr.DataArray(fcstweekly['rmm1'][:,:,::7],
                          coords={'init' : fcstweekly['init'],
                                  'member': fcstweekly['member'],
                                  'lead': np.arange(1,nleads+1),
                                  },
                          dims=['init', 'member','lead'])
fcstweeklyda.name = 'rmm1'
```

`climpred` requires that `lead` dimension has an attribute called `units` indicating what time units the `lead` is associated with. Options are: `years`, `seasons`, `months`, `weeks`, `pentads`, `days`. The `lead` `units` are `weeks`.

[11]:
```
fcstweeklyda['lead'].attrs={'units': 'weeks'}
```

Create the `climpred HindcastEnsemble` object and add the observations.

[12]:
```
hindcast = HindcastEnsemble(fcstweeklyda)
hindcast=hindcast.add_observations(obsweekly, 'observations')
```

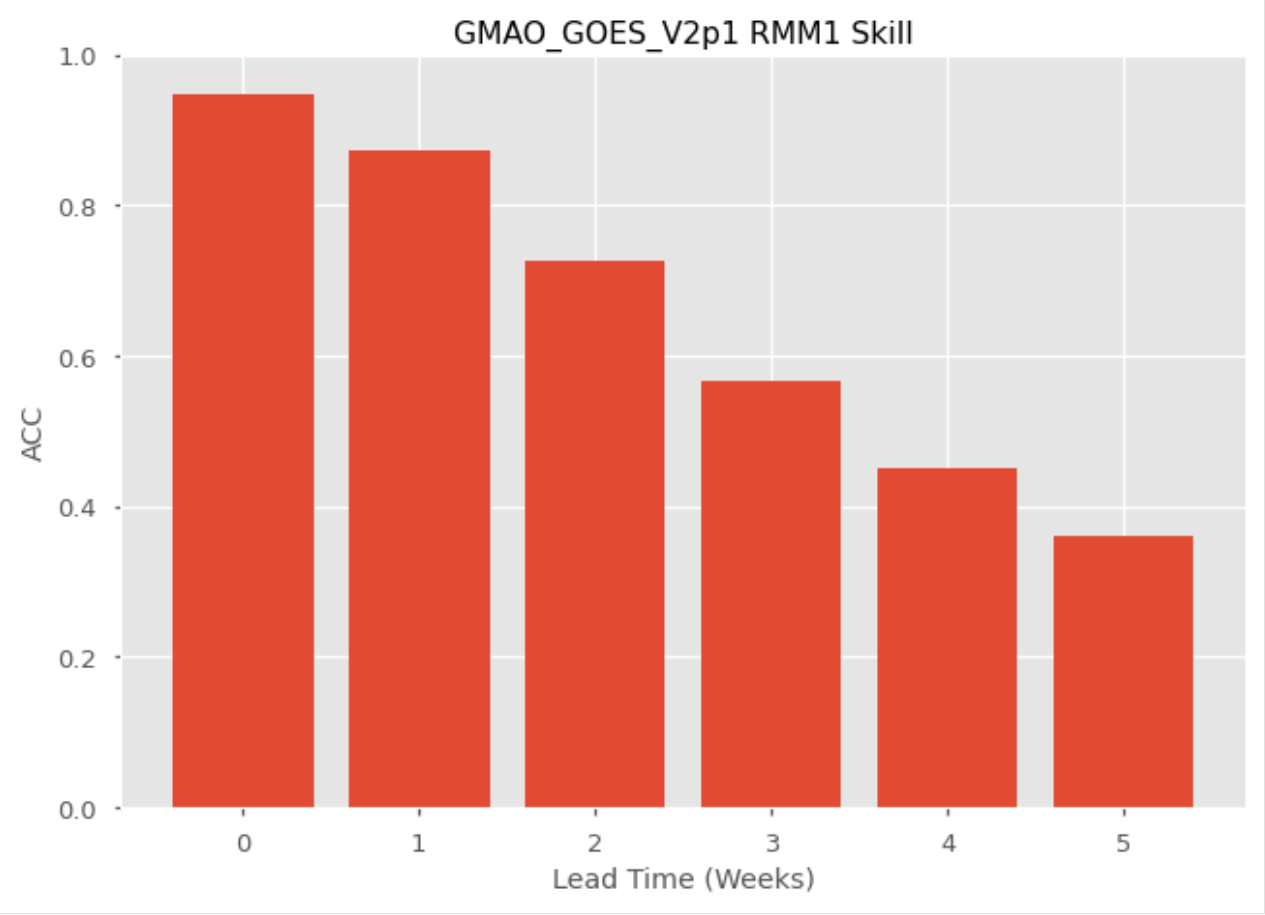
Calculate the Anomaly Correlation Coefficient (ACC)

[13]:
```
skill = hindcast.verify(metric='acc', alignment='maximize')
```

Plot the skill as a function of lead time

[14]:
```
x=np.arange(fcstweeklyda['lead'].size)
plt.bar(x,skill['rmm1'])
plt.title('GMAO_GOES_V2p1 RMM1 Skill')
plt.xlabel('Lead Time (Weeks)')
plt.ylabel('ACC')
plt.ylim(0.0, 1.0)
```

[14]:
```
(0.0, 1.0)
```



References

1. Pegion, K., B.P. Kirtman, E. Becker, D.C. Collins, E. LaJoie, R. Burgman, R. Bell, T. DelSole, D. Min, Y. Zhu, W. Li, E. Sinsky, H. Guan, J. Gottschalck, E.J. Metzger, N.P. Barton, D. Achuthavarier, J. Marshak, R.D. Koster, H. Lin, N. Gagnon, M. Bell, M.K. Tippett, A.W. Robertson, S. Sun, S.G. Benjamin, B.W. Green, R. Bleck, and H. Kim, 2019: The Subseasonal Experiment (SubX): A Multimodel Subseasonal Prediction Experiment. Bull. Amer. Meteor. Soc., 100, 2043–2060, https://doi.org/10.1175/BAMS-D-18-0270.1

2. Kirtman, B. P., Pegion, K., DelSole, T., Tippett, M., Robertson, A. W., Bell, M., … Green, B. W. (2017). The Subseasonal Experiment (SubX) [Data set]. IRI Data Library. https://doi.org/10.7916/D8PG249H

3. Wheeler, M. C., & Hendon, H. H. (2004). An all-season real-time multivariate MJO index: Development of an index for monitoring and prediction. Monthly Weather Review, 132(8), 1917–1932. http://doi.org/10.1175/1520-0493(2004)132<1917:AARMMI>2.0.CO;2

4. Rashid, H. A., Hendon, H. H., Wheeler, M. C., & Alves, O. (2010). Prediction of the Madden–Julian oscillation with the POAMA dynamical prediction system. Climate Dynamics, 36(3-4), 649–661. http://doi.org/10.1007/

s00382-010-0754-x

5. Gottschalck, J., Wheeler, M., Weickmann, K., Vitart, F., Savage, N., Lin, H., et al. (2010). A Framework for Assessing Operational Madden–Julian Oscillation Forecasts: A CLIVAR MJO Working Group Project. Bulletin of the American Meteorological Society, 91(9), 1247–1258. http://doi.org/10.1175/2010BAMS2816.1

2.4.4 Monthly and Seasonal

Calculate ENSO Skill as a Function of Initial Month vs. Lead Time

In this example, we demonstrate:

1. How to remotely access data from the North American Multi-model Ensemble (NMME) hindcast database and set it up to be used in `climpred`.
2. How to calculate the Anomaly Correlation Coefficient (ACC) using monthly data
3. How to calculate and plot historical forecast skill of the Nino3.4 index as function of initialization month and lead time.

The North American Multi-model Ensemble (NMME)

Further information on NMME is available from Kirtman et al. 2014 and the NMME project website

The NMME public database is hosted on the International Research Institute for Climate and Society (IRI) data server http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/

Since the NMME data server is accessed via this notebook, the time for the notebook to run may take a few minutes and vary depending on the speed that data is downloaded.

Definitions

Anomalies Departure from normal, where normal is defined as the climatological value based on the average value for each month over all years.

Nino3.4 An index used to represent the evolution of the El Nino-Southern Oscillation (ENSO). Calculated as the average sea surface temperature (SST) anomalies in the region 5S-5N; 190-240

[1]:
```
import warnings

import matplotlib.pyplot as plt
import xarray as xr
import pandas as pd
import numpy as np
from tqdm.auto import tqdm

from climpred import HindcastEnsemble
import climpred
```

[2]:
```
warnings.filterwarnings("ignore")
```

Function to set 360 calendar to 360_day calendar and decond cf times

[3]:
```
def decode_cf(ds, time_var):
    if ds[time_var].attrs['calendar'] == '360':
        ds[time_var].attrs['calendar'] = '360_day'
    ds = xr.decode_cf(ds, decode_times=True)
    return ds
```

Load the monthly sea surface temperature (SST) hindcast data for the NCEP-CFSv2 model from the NMME data server. This is a large dataset, so we allow dask to chunk the data as it chooses.

[4]:
```
url = 'http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/NCEP-CFSv2/.HINDCAST/.
↪MONTHLY/.sst/dods'
fcstds = decode_cf(xr.open_dataset(url, decode_times=False,
                                   chunks={'S': 'auto', 'L': 'auto', 'M':'auto'}),'S')

fcstds
```

[4]:
```
<xarray.Dataset>
Dimensions:  (L: 10, M: 24, S: 348, X: 360, Y: 181)
Coordinates:
  * S        (S) object 1982-01-01 00:00:00 ... 2010-12-01 00:00:00
  * M        (M) float32 1.0 2.0 3.0 4.0 5.0 6.0 ... 20.0 21.0 22.0 23.0 24.0
  * X        (X) float32 0.0 1.0 2.0 3.0 4.0 ... 355.0 356.0 357.0 358.0 359.0
  * L        (L) float32 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
  * Y        (Y) float32 -90.0 -89.0 -88.0 -87.0 -86.0 ... 87.0 88.0 89.0 90.0
Data variables:
    sst      (S, L, M, Y, X) float32 dask.array<chunksize=(6, 5, 8, 181, 360),␣
↪meta=np.ndarray>
Attributes:
    Conventions:  IRIDL
```

The NMME data dimensions correspond to the following `climpred` dimension definitions: `X=lon`,`L=lead`,`Y=lat`,`M=member`, `S=init`. We will rename the dimensions to their `climpred` names.

[5]:
```
fcstds=fcstds.rename({'S': 'init','L': 'lead','M': 'member', 'X': 'lon', 'Y': 'lat'})
```

Let's make sure that the `lead` dimension is set properly for `climpred`. NMME data stores `leads` as 0.5, 1.5, 2.5, etc, which correspond to 0, 1, 2, ... months since initialization. We will change the `lead` to be integers starting with zero. `climpred` also requires that `lead` dimension has an attribute called `units` indicating what time units the `lead` is assocated with. Options are: `years`,`seasons`,`months`,`weeks`,`pentads`,`days`. For the monthly NMME data, the `lead` `units` are `months`.

[6]:
```
fcstds['lead']=(fcstds['lead']-0.5).astype('int')
fcstds['lead'].attrs={'units': 'months'}
```

Now we need to make sure that the `init` dimension is set properly for `climpred`. For monthly data, the `init` dimension must be a `xr.cfdateTimeIndex` or a `pd.datetimeIndex`. We convert the `init` values to `pd.datatimeIndex`.

[7]:
```
fcstds['init']=pd.to_datetime(fcstds.init.values.astype(str))
fcstds['init']=pd.to_datetime(fcstds['init'].dt.strftime('%Y%m01 00:00'))
```

Next, we want to get the verification SST data from the data server

[8]:
```
obsurl='http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/.OIv2_SST/.sst/dods'
verifds = decode_cf(xr.open_dataset(obsurl, decode_times=False),'T')
verifds
```

[8]:
```
<xarray.Dataset>
Dimensions:  (T: 405, X: 360, Y: 181)
Coordinates:
  * Y        (Y) float32 -90.0 -89.0 -88.0 -87.0 -86.0 ... 87.0 88.0 89.0 90.0
  * X        (X) float32 0.0 1.0 2.0 3.0 4.0 ... 355.0 356.0 357.0 358.0 359.0
  * T        (T) object 1982-01-16 00:00:00 ... 2015-09-16 00:00:00
Data variables:
    sst      (T, Y, X) float32 ...
Attributes:
    Conventions:  IRIDL
```

Rename the dimensions to correspond to climpred dimensions

[9]:
```
verifds=verifds.rename({'T': 'time','X': 'lon', 'Y': 'lat'})
```

Convert the `time` data to be of type `pd.datetimeIndex`

[10]:
```
verifds['time']=pd.to_datetime(verifds.time.values.astype(str))
verifds['time']=pd.to_datetime(verifds['time'].dt.strftime('%Y%m01 00:00'))
```

Subset the data to 1982-2010

[11]:
```
fcstds=fcstds.sel(init=slice('1982-01-01','2010-12-01'))
verifds=verifds.sel(time=slice('1982-01-01','2010-12-01'))
```

Calculate the Nino3.4 index for forecast and verification.

[12]:
```
fcstnino34=fcstds.sel(lat=slice(-5,5),lon=slice(190,240)).mean(['lat','lon'])
verifnino34=verifds.sel(lat=slice(-5,5),lon=slice(190,240)).mean(['lat','lon'])

fcstclimo = fcstnino34.groupby('init.month').mean('init')
fcst = (fcstnino34.groupby('init.month') - fcstclimo)

verifclimo = verifnino34.groupby('time.month').mean('time')
verif = (verifnino34.groupby('time.month') - verifclimo)
```

Because will will calculate the anomaly correlation coefficient over all `time` for verification and `init` for the hindcasts, we need to rechunk the data so that these dimensions are in same chunk

[13]:
```
fcst=fcst.chunk({'init':-1})
verif=verif.chunk({'time':-1})
```

Use the `climpred HindcastEnsemble` to calculate the anomaly correlation coefficient (ACC) as a function of initial month and `lead`

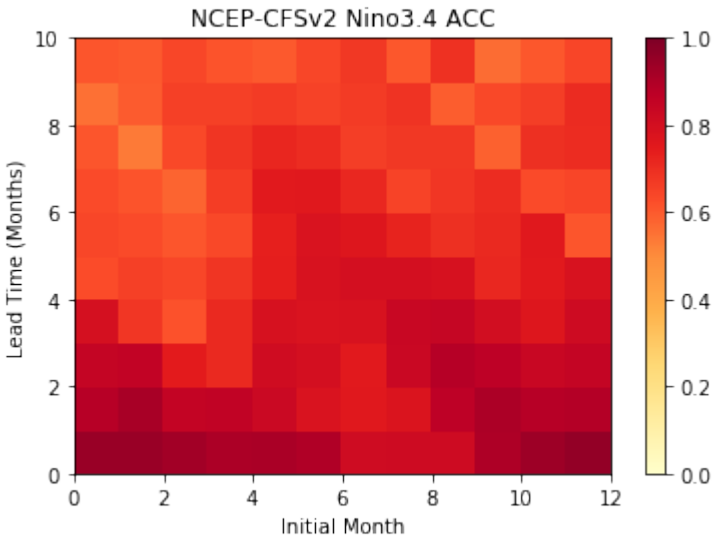
[14]:
```
skill=np.zeros((fcst['lead'].size, 12))
for im in tqdm(np.arange(0,12)):
    hindcast = HindcastEnsemble(fcst.sel(init=fcst['init.month']==im+1))
    hindcast = hindcast.add_observations(verif, 'observations')
    skillds = hindcast.verify(metric='acc')
    skill[:,im]=skillds['sst'].values
```

```
HBox(children=(FloatProgress(value=0.0, max=12.0), HTML(value='')))
```

Plot the ACC as function of Initial Month and lead-time

[15]:
```
plt.pcolormesh(skill,cmap=plt.cm.YlOrRd,vmin=0.0,vmax=1.0)
plt.colorbar()
plt.title('NCEP-CFSv2 Nino3.4 ACC')
plt.xlabel('Initial Month')
plt.ylabel('Lead Time (Months)')
```

[15]:
```
Text(0, 0.5, 'Lead Time (Months)')
```



References

1. Kirtman, B.P., D. Min, J.M. Infanti, J.L. Kinter, D.A. Paolino, Q. Zhang, H. van den Dool, S. Saha, M.P. Mendez, E. Becker, P. Peng, P. Tripp, J. Huang, D.G. DeWitt, M.K. Tippett, A.G. Barnston, S. Li, A. Rosati, S.D. Schubert, M. Rienecker, M. Suarez, Z.E. Li, J. Marshak, Y. Lim, J. Tribbia, K. Pegion, W.J. Merryfield, B. Denis, and E.F. Wood, 2014: The North American Multimodel Ensemble: Phase-1 Seasonal-to-Interannual Prediction; Phase-2 toward Developing Intraseasonal Prediction. Bull. Amer. Meteor. Soc., 95, 585–601, https://doi.org/10.1175/BAMS-D-12-00050.1

Calculate Seasonal ENSO Skill

In this example, we demonstrate:

1. How to remotely access data from the North American Multi-model Ensemble (NMME) hindcast database and set it up to be used in `climpred`
2. How to calculate the Anomaly Correlation Coefficient (ACC) using seasonal data

The North American Multi-model Ensemble (NMME)

Further information on NMME is available from Kirtman et al. 2014 and the NMME project website

The NMME public database is hosted on the International Research Institute for Climate and Society (IRI) data server http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/

Definitions

Anomalies Departure from normal, where normal is defined as the climatological value based on the average value for each month over all years.

Nino3.4 An index used to represent the evolution of the El Nino-Southern Oscillation (ENSO). Calculated as the average sea surface temperature (SST) anomalies in the region 5S-5N; 190-240

[1]:
```
import warnings

import matplotlib.pyplot as plt
import xarray as xr
import pandas as pd
import numpy as np

from climpred import HindcastEnsemble
import climpred
```

[2]:
```
warnings.filterwarnings("ignore")
```

[3]:
```
def decode_cf(ds, time_var):
    if ds[time_var].attrs['calendar'] == '360':
        ds[time_var].attrs['calendar'] = '360_day'
    ds = xr.decode_cf(ds, decode_times=True)
    return ds
```

Load the monthly sea surface temperature (SST) hindcast data for the NCEP-CFSv2 model from the data server

[4]:
```
# Get NMME data for NCEP-CFSv2, SST
url = 'http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/NCEP-CFSv2/.HINDCAST/.
↪MONTHLY/.sst/dods'
fcstds = decode_cf(xr.open_dataset(url, decode_times=False, chunks={'S': 1, 'L': 12}),
↪ 'S')
fcstds
```

[4]:
```
<xarray.Dataset>
Dimensions:  (L: 10, M: 24, S: 348, X: 360, Y: 181)
Coordinates:
  * S        (S) object 1982-01-01 00:00:00 ... 2010-12-01 00:00:00
  * M        (M) float32 1.0 2.0 3.0 4.0 5.0 6.0 ... 20.0 21.0 22.0 23.0 24.0
  * X        (X) float32 0.0 1.0 2.0 3.0 4.0 ... 355.0 356.0 357.0 358.0 359.0
  * L        (L) float32 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
  * Y        (Y) float32 -90.0 -89.0 -88.0 -87.0 -86.0 ... 87.0 88.0 89.0 90.0
Data variables:
    sst      (S, L, M, Y, X) float32 dask.array<chunksize=(1, 10, 24, 181, 360),␣
↪meta=np.ndarray>
Attributes:
    Conventions:  IRIDL
```

The NMME data dimensions correspond to the following `climpred` dimension definitions: `X=lon`,`L=lead`,`Y=lat`,`M=member`, `S=init`. We will rename the dimensions to their `climpred` names.

[5]:
```
fcstds=fcstds.rename({'S': 'init','L': 'lead','M': 'member', 'X': 'lon', 'Y': 'lat'})
```

Let's make sure that the `lead` dimension is set properly for `climpred`. NMME data stores `leads` as 0.5, 1.5, 2.5, etc, which correspond to 0, 1, 2, ... months since initialization. We will change the `lead` to be integers starting with zero.

[6]:
```
fcstds['lead']=(fcstds['lead']-0.5).astype('int')
```

Now we need to make sure that the `init` dimension is set properly for `climpred`. For monthly data, the `init` dimension must be a `xr.cfdateTimeIndex` or a `pd.datetimeIndex`. We convert the `init` values to `pd.datatimeIndex`.

[7]:
```
fcstds['init']=pd.to_datetime(fcstds.init.values.astype(str))
fcstds['init']=pd.to_datetime(fcstds['init'].dt.strftime('%Y%m01 00:00'))
```

Next, we want to get the verification SST data from the data server

[8]:
```
obsurl='http://iridl.ldeo.columbia.edu/SOURCES/.Models/.NMME/.OIv2_SST/.sst/dods'
verifds = decode_cf(xr.open_dataset(obsurl, decode_times=False),'T')
verifds
```

[8]:
```
<xarray.Dataset>
Dimensions:  (T: 405, X: 360, Y: 181)
Coordinates:
  * Y        (Y) float32 -90.0 -89.0 -88.0 -87.0 -86.0 ... 87.0 88.0 89.0 90.0
  * X        (X) float32 0.0 1.0 2.0 3.0 4.0 ... 355.0 356.0 357.0 358.0 359.0
  * T        (T) object 1982-01-16 00:00:00 ... 2015-09-16 00:00:00
Data variables:
    sst      (T, Y, X) float32 ...
Attributes:
    Conventions:  IRIDL
```

Rename the dimensions to correspond to `climpred` dimensions

[9]:
```
verifds=verifds.rename({'T': 'time','X': 'lon', 'Y': 'lat'})
```

Convert the `time` data to be of type `pd.datetimeIndex`

[10]:
```
verifds['time']=pd.to_datetime(verifds.time.values.astype(str))
verifds['time']=pd.to_datetime(verifds['time'].dt.strftime('%Y%m01 00:00'))
verifds
```

[10]:
```
<xarray.Dataset>
Dimensions:  (lat: 181, lon: 360, time: 405)
Coordinates:
  * lat      (lat) float32 -90.0 -89.0 -88.0 -87.0 -86.0 ... 87.0 88.0 89.0 90.0
  * lon      (lon) float32 0.0 1.0 2.0 3.0 4.0 ... 355.0 356.0 357.0 358.0 359.0
  * time     (time) datetime64[ns] 1982-01-01 1982-02-01 ... 2015-09-01
Data variables:
    sst      (time, lat, lon) float32 ...
Attributes:
    Conventions:  IRIDL
```

Subset the data to 1982-2010

[11]:
```
verifds=verifds.sel(time=slice('1982-01-01','2010-12-01'))
fcstds=fcstds.sel(init=slice('1982-01-01','2010-12-01'))
```

Calculate the Nino3.4 index for forecast and verification

[12]:
```
fcstnino34=fcstds.sel(lat=slice(-5,5),lon=slice(190,240)).mean(['lat','lon'])
verifnino34=verifds.sel(lat=slice(-5,5),lon=slice(190,240)).mean(['lat','lon'])

fcstclimo = fcstnino34.groupby('init.month').mean('init')
```

(continues on next page)

(continued from previous page)

```
fcstanoms = (fcstnino34.groupby('init.month') - fcstclimo)

verifclimo = verifnino34.groupby('time.month').mean('time')
verifanoms = (verifnino34.groupby('time.month') - verifclimo)

print(fcstanoms)
print(verifanoms)
```

```
<xarray.Dataset>
Dimensions:  (init: 348, lead: 10, member: 24)
Coordinates:
  * lead     (lead) int64 0 1 2 3 4 5 6 7 8 9
  * member   (member) float32 1.0 2.0 3.0 4.0 5.0 ... 20.0 21.0 22.0 23.0 24.0
  * init     (init) datetime64[ns] 1982-01-01 1982-02-01 ... 2010-12-01
    month    (init) int64 1 2 3 4 5 6 7 8 9 10 11 ... 2 3 4 5 6 7 8 9 10 11 12
Data variables:
    sst      (init, lead, member) float32 dask.array<chunksize=(1, 10, 24), meta=np.
↪ndarray>
<xarray.Dataset>
Dimensions:  (time: 348)
Coordinates:
  * time     (time) datetime64[ns] 1982-01-01 1982-02-01 ... 2010-12-01
    month    (time) int64 1 2 3 4 5 6 7 8 9 10 11 ... 2 3 4 5 6 7 8 9 10 11 12
Data variables:
    sst      (time) float32 0.14492226 -0.044160843 ... -1.5685654 -1.6083965
```

Make Seasonal Averages with center=True and drop NaNs. This means that the first value

[13]:
```
fcstnino34seas=fcstanoms.rolling(lead=3, center=True).mean().dropna(dim='lead')
verifnino34seas=verifanoms.rolling(time=3, center=True).mean().dropna(dim='time')
```

Create new `xr.DataArray` with seasonal data

[14]:
```
nleads=fcstnino34seas['lead'][::3].size
fcst=xr.DataArray(fcstnino34seas['sst'][:,::3,:],
                          coords={'init' : fcstnino34seas['init'],
                                  'lead': np.arange(0,nleads),
                                  'member': fcstanoms['member'],
                                  },
                          dims=['init','lead','member'])
fcst.name = 'sst'
```

Assign the `units` attribute of `seasons` to the `lead` dimension

[15]:
```
fcst['lead'].attrs={'units': 'seasons'}
```

Create a `climpred HindcastEnsemble` object

[16]:
```
hindcast = HindcastEnsemble(fcst)
hindcast = hindcast.add_observations(verifnino34seas, 'observations')
```

Compute the Anomaly Correlation Coefficient (ACC) 0, 1, 2, and 3 season lead-times

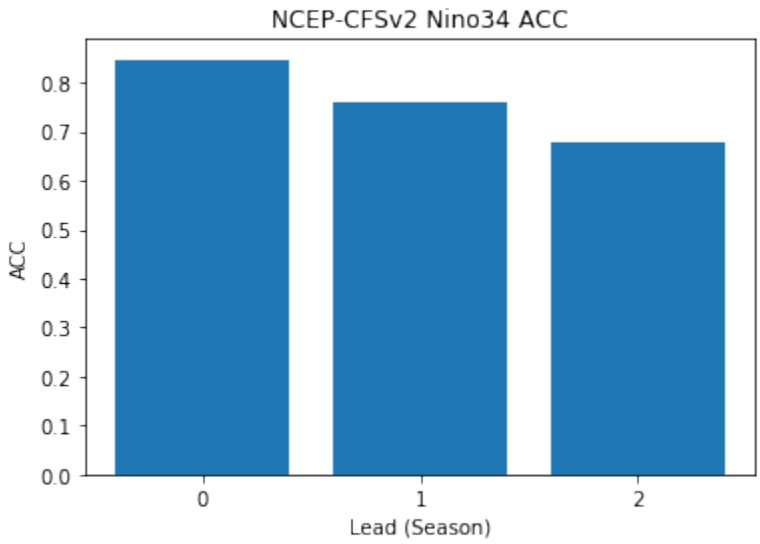
[17]:
```
skillds = hindcast.verify(metric='acc')
print(skillds)
```

```
<xarray.Dataset>
Dimensions:  (lead: 3)
Coordinates:
  * lead     (lead) int64 0 1 2
Data variables:
    sst      (lead) float64 0.847 0.7614 0.6779
Attributes:
    prediction_skill:              calculated by climpred https://climpred.re...
    skill_calculated_by_function:  compute_hindcast
    number_of_initializations:     348
    number_of_members:             24
    metric:                        pearson_r
    comparison:                    e2o
    dim:                           time
    units:                         None
    created:                       2020-01-21 11:41:35
```

Make bar plot of Nino3.4 skill for 0,1, and 2 season lead times

[18]:
```
x=np.arange(0,nleads,1.0).astype(int)
plt.bar(x,skillds['sst'])
plt.xticks(x)
plt.title('NCEP-CFSv2 Nino34 ACC')
plt.xlabel('Lead (Season)')
plt.ylabel('ACC')
```

[18]:
```
Text(0, 0.5, 'ACC')
```



References

1. Kirtman, B.P., D. Min, J.M. Infanti, J.L. Kinter, D.A. Paolino, Q. Zhang, H. van den Dool, S. Saha, M.P. Mendez, E. Becker, P. Peng, P. Tripp, J. Huang, D.G. DeWitt, M.K. Tippett, A.G. Barnston, S. Li, A. Rosati, S.D. Schubert, M. Rienecker, M. Suarez, Z.E. Li, J. Marshak, Y. Lim, J. Tribbia, K. Pegion, W.J. Merryfield, B. Denis, and E.F. Wood, 2014: The North American Multimodel Ensemble: Phase-1 Seasonal-to-Interannual Prediction; Phase-2 toward Developing Intraseasonal Prediction. Bull. Amer. Meteor. Soc., 95, 585–601, https://doi.org/10.1175/BAMS-D-12-00050.1

2.4.5 Decadal

Demo of Perfect Model Predictability Functions

This demo demonstrates `climpred`'s capabilities for a perfect-model framework ensemble simulation.

What's a perfect-model framework simulation?

A perfect-model framework uses a set of ensemble simulations that are based on a General Circulation Model (GCM) or Earth System Model (ESM) alone. There is *no* use of any reanalysis, reconstruction, or data product to initialize the decadal prediction ensemble. An arbitrary number of `members` are initialized from perturbed initial conditions (the "`ensemble`"), and the control simulation can be viewed as just another member.

How to compare predictability skill score: As no observational data interferes with the random climate evolution of the model, we cannot use an observation-based reference for computing skill scores. Therefore, we can compare the members with one another (`m2m`), against the ensemble mean (`m2e`), or against the control (`m2c`). We can also compare the ensemble mean to the control (`e2c`). See the comparisons page for more information.

When to use perfect-model frameworks:

- You don't have a sufficiently long observational record to use as a reference.
- You want to avoid biases between model climatology and reanalysis climatology.
- You want to avoid sensitive reactions of biogeochemical cycles to disruptive changes in ocean physics due to assimilation.
- You want to delve into process understanding of predictability in a model without outside artifacts.

[1]:
```
import warnings

import cartopy.crs as ccrs
import cartopy.feature as cfeature
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import climpred
```

[2]:
```
warnings.filterwarnings("ignore")
```

Load sample data

Here we use a subset of ensembles and members from the MPI-ESM-LR (CMIP6 version) esmControl simulation of an early state. This corresponds to `vga0214` from year 3000 to 3300.

1-dimensional output

Our 1D sample output contains datasets of time series of certain spatially averaged `area` ('global', 'North_Atlantic') and temporally averaged `period` ('ym', 'DJF', ...) for some `lead` years (1, ..., 20).

`ds`: The ensemble dataset of all `members` (1, ..., 10), `inits` (initialization years: 3014, 3023, ..., 3257), `areas`, `periods`, and `lead` years.

`control`: The control dataset with the same areas and periods, as well as the years 3000 to 3299.

[28]:
```
ds = climpred.tutorial.load_dataset('MPI-PM-DP-1D').isel(area=1,period=-1)['tos']
control = climpred.tutorial.load_dataset('MPI-control-1D').isel(area=1,period=-1)['tos
↪']
```

(continues on next page)

(continued from previous page)

[29]:
```
ds['lead'].attrs = {'units': 'years'}
```

[30]:
```
# Add to climpred PerfectModelEnsemble object.
pm = climpred.PerfectModelEnsemble(ds)
pm = pm.add_control(control)
print(pm)
```

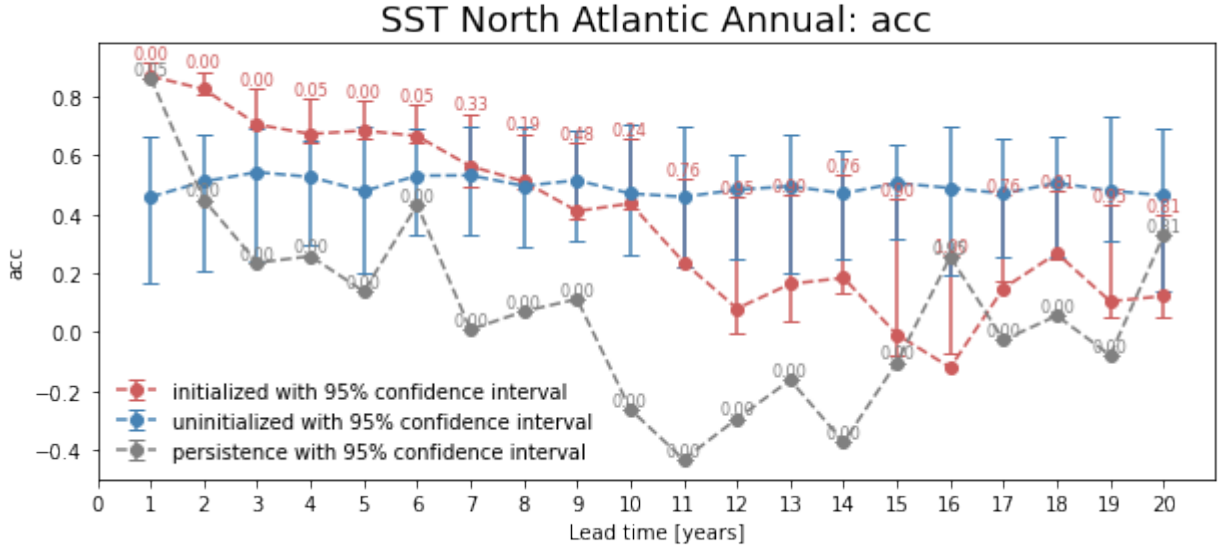
```
<climpred.PerfectModelEnsemble>
Initialized Ensemble:
    tos      (lead, init, member) float32 ...
Control:
    tos      (time) float32 ...
Uninitialized:
    None
```

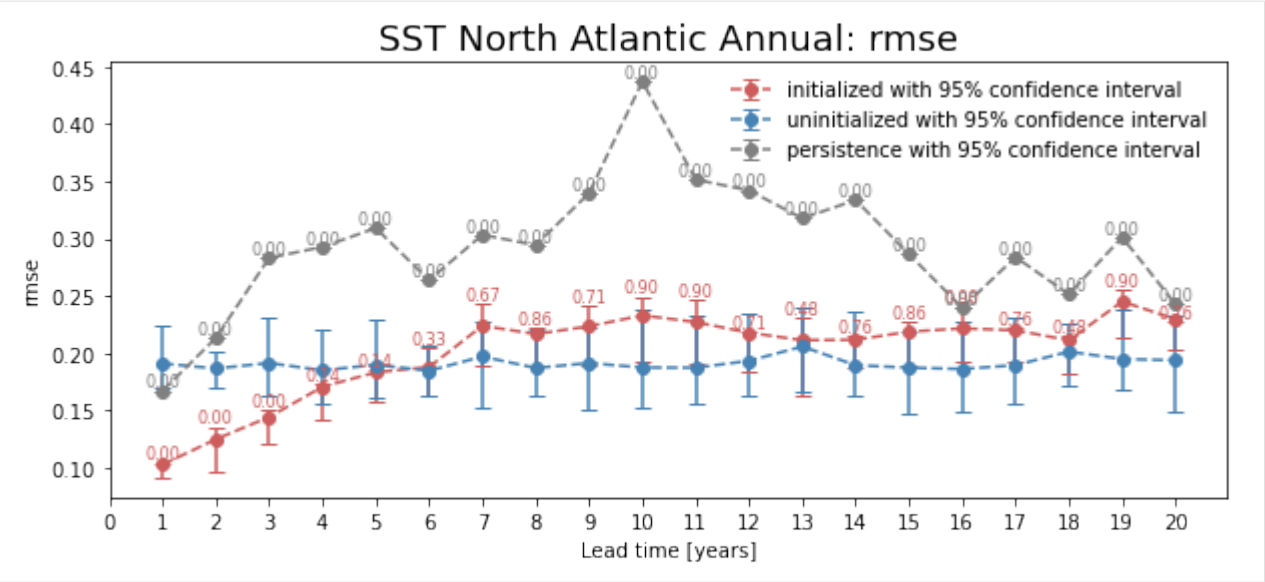
We'll sub-select annual means ('ym') of sea surface temperature ('tos') in the North Atlantic.

Bootstrapping with Replacement

Here, we bootstrap the ensemble with replacement [Goddard et al. 2013] to compare the initialized ensemble to an "uninitialized" counterpart and a persistence forecast. The visualization is based on those used in [Li et al. 2016]. The p-value demonstrates the probability that the uninitialized or persistence beats the initialized forecast based on N bootstrapping with replacement.

[31]:
```
for metric in ['acc', 'rmse']:
    bootstrapped = pm.bootstrap(metric=metric, comparison='m2e', iterations=21,␣
↪sig=95)
    climpred.graphics.plot_bootstrapped_skill_over_leadyear(bootstrapped)
    plt.title(' '.join(['SST', 'North Atlantic', 'Annual:', metric]),fontsize=18)
    plt.ylabel(metric)
    plt.show()
```

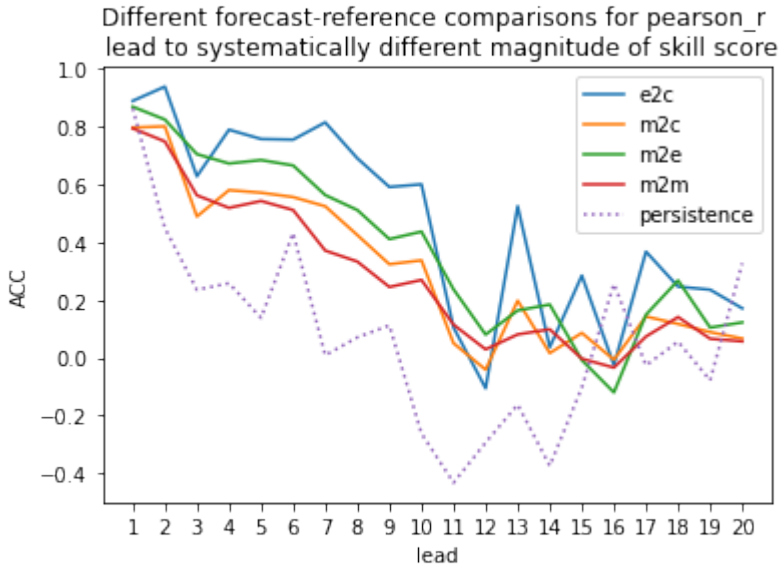




Computing Skill with Different Comparison Methods

Here, we use `compute_perfect_model` to compute the Anomaly Correlation Coefficient (ACC) with different comparison methods. This generates different ACC values by design. See the comparisons page for a description of the various ways to compute skill scores for a perfect-model framework.

[32]:
```
for c in ['e2c','m2c','m2e','m2m']:
    pm.compute_metric(metric='acc', comparison=c)['tos'].plot(label=c)
# Persistence computation for a baseline.
pm.compute_persistence(metric='acc')['tos'].plot(label='persistence', ls=':')
plt.ylabel('ACC')
plt.xticks(np.arange(1,21))
plt.legend()
plt.title('Different forecast-reference comparisons for pearson_r \n lead to␣
↪systematically different magnitude of skill score')
plt.show()
```



3-dimensional output (maps)

We also have some sample output that contains gridded time series on the curvilinear MPI grid. Our compute functions (`compute_perfect_model`, `compute_persistence`) are indifferent to any dimensions that exist in addition to `init`, `member`, and `lead`. In other words, the functions are set up to make these computations on a grid, if one includes `lat`, `lon`, `lev`, `depth`, etc.

`ds3d`: The ensemble dataset of `members` (1, 2, 3, 4), `inits` (initialization years: 3014, 3061, 3175, 3237), and `lead` years (1, 2, 3, 4, 5).

`control3d`: The control dataset spanning (3000, . . . , 3049).

Note: These are very small subsets of the actual MPI simulations so that we could host the sample output maps on Github.

[33]:
```
# Sea surface temperature
ds3d = climpred.tutorial.load_dataset('MPI-PM-DP-3D') \
                        .sel(init=3014) \
                        .expand_dims('init')['tos']
control3d = climpred.tutorial.load_dataset('MPI-control-3D')['tos']
```

[34]:
```
ds3d['lead'].attrs = {'units': 'years'}
```

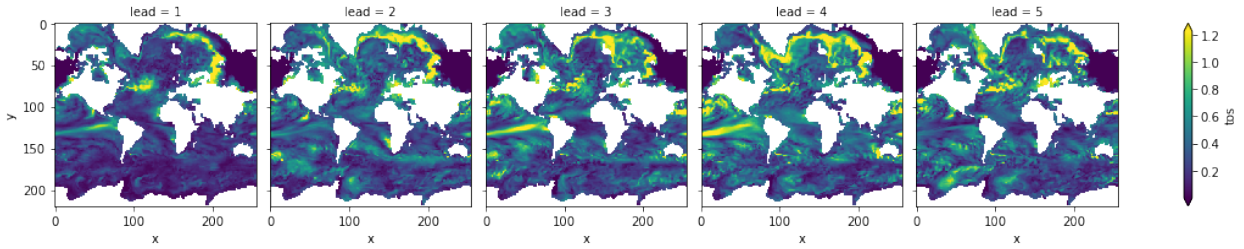
[35]:
```
# Create climpred PerfectModelEnsemble object.
pm = climpred.PerfectModelEnsemble(ds3d)
pm = pm.add_control(control3d)
print(pm)
```

```
<climpred.PerfectModelEnsemble>
Initialized Ensemble:
    tos      (init, lead, member, y, x) float32 nan nan nan nan ... nan nan nan
Control:
    tos      (time, y, x) float32 ...
Uninitialized:
    None
```

Maps of Skill by Lead Year

[36]:
```
pm.compute_metric(metric='rmse', comparison='m2e')['tos'] \
  .T.plot(col='lead', robust=True, yincrease=False)
```

[36]:
```
<xarray.plot.facetgrid.FacetGrid at 0x11d8fffd0>
```



References

1. Bushuk, Mitchell, Rym Msadek, Michael Winton, Gabriel Vecchi, Xiaosong Yang, Anthony Rosati, and Rich Gudgel. "Regional Arctic Sea–Ice Prediction: Potential versus Operational Seasonal Forecast Skill." Climate Dynamics, June 9, 2018. https://doi.org/10/gd7hfq.
2. Collins, Matthew, and Sinha Bablu. "Predictability of Decadal Variations in the Thermohaline Circulation and Climate." Geophysical Research Letters 30, no. 6 (March 22, 2003). https://doi.org/10/cts3cr.
3. Goddard, Lisa, et al. "A verification framework for interannual-to-decadal predictions experiments." Climate Dynamics 40.1-2 (2013): 245-272.
4. Griffies, S. M., and K. Bryan. "A Predictability Study of Simulated North Atlantic Multidecadal Variability." Climate Dynamics 13, no. 7–8 (August 1, 1997): 459–87. https://doi.org/10/ch4kc4.
5. Hawkins, Ed, Steffen Tietsche, Jonathan J. Day, Nathanael Melia, Keith Haines, and Sarah Keeley. "Aspects of Designing and Evaluating Seasonal-to-Interannual Arctic Sea-Ice Prediction Systems." Quarterly Journal of the Royal Meteorological Society 142, no. 695 (January 1, 2016): 672–83. https://doi.org/10/gfb3pn.
6. Li, Hongmei, Tatiana Ilyina, Wolfgang A. Müller, and Frank Sienz. "Decadal Predictions of the North Atlantic CO2 Uptake." Nature Communications 7 (March 30, 2016): 11076. https://doi.org/10/f8wkrs.
7. Pohlmann, Holger, Michael Botzet, Mojib Latif, Andreas Roesch, Martin Wild, and Peter Tschuck. "Estimating the Decadal Predictability of a Coupled AOGCM." Journal of Climate 17, no. 22 (November 1, 2004): 4463–72. https://doi.org/10/d2qf62.

[ ]:
```
```

Hindcast Predictions of Equatorial Pacific SSTs

In this example, we evaluate hindcasts (retrospective forecasts) of sea surface temperatures in the eastern equatorial Pacific from CESM-DPLE. These hindcasts are evaluated against a forced ocean–sea ice simulation that initializes the model.

See the quick start for an analysis of time series (rather than maps) from a hindcast prediction ensemble.

[1]:
```
import warnings

import cartopy.crs as ccrs
```

(continues on next page)

(continued from previous page)

```
import cartopy.feature as cfeature
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

import climpred
from climpred import HindcastEnsemble
```

[2]:
```
warnings.filterwarnings("ignore")
```

We'll load in a small region of the eastern equatorial Pacific for this analysis example.

[3]:
```
climpred.tutorial.load_dataset()
```

```
'MPI-control-1D': area averages for the MPI control run of SST/SSS.
'MPI-control-3D': lat/lon/time for the MPI control run of SST/SSS.
'MPI-PM-DP-1D': perfect model decadal prediction ensemble area averages of SST/SSS/
↪AMO.
'MPI-PM-DP-3D': perfect model decadal prediction ensemble lat/lon/time of SST/SSS/AMO.
'CESM-DP-SST': hindcast decadal prediction ensemble of global mean SSTs.
'CESM-DP-SSS': hindcast decadal prediction ensemble of global mean SSS.
'CESM-DP-SST-3D': hindcast decadal prediction ensemble of eastern Pacific SSTs.
'CESM-LE': uninitialized ensemble of global mean SSTs.
'MPIESM_miklip_baseline1-hind-SST-global': hindcast initialized ensemble of global␣
↪mean SSTs
'MPIESM_miklip_baseline1-hist-SST-global': uninitialized ensemble of global mean SSTs
'MPIESM_miklip_baseline1-assim-SST-global': assimilation in MPI-ESM of global mean␣
↪SSTs
'ERSST': observations of global mean SSTs.
'FOSI-SST': reconstruction of global mean SSTs.
'FOSI-SSS': reconstruction of global mean SSS.
'FOSI-SST-3D': reconstruction of eastern Pacific SSTs
'GMAO-GEOS-RMM1': daily RMM1 from the GMAO-GEOS-V2p1 model for SubX
'RMM-INTERANN-OBS': observed RMM with interannual variablity included
```

[4]:
```
hind = climpred.tutorial.load_dataset('CESM-DP-SST-3D')['SST']
recon = climpred.tutorial.load_dataset('FOSI-SST-3D')['SST']
print(hind)
```

```
<xarray.DataArray 'SST' (init: 64, lead: 10, nlat: 37, nlon: 26)>
[615680 values with dtype=float32]
Coordinates:
    TLAT     (nlat, nlon) float64 ...
    TLONG    (nlat, nlon) float64 ...
  * init     (init) float32 1954.0 1955.0 1956.0 1957.0 ... 2015.0 2016.0 2017.0
  * lead     (lead) int32 1 2 3 4 5 6 7 8 9 10
    TAREA    (nlat, nlon) float64 ...
Dimensions without coordinates: nlat, nlon
```

These two example products cover a small portion of the eastern equatorial Pacific.

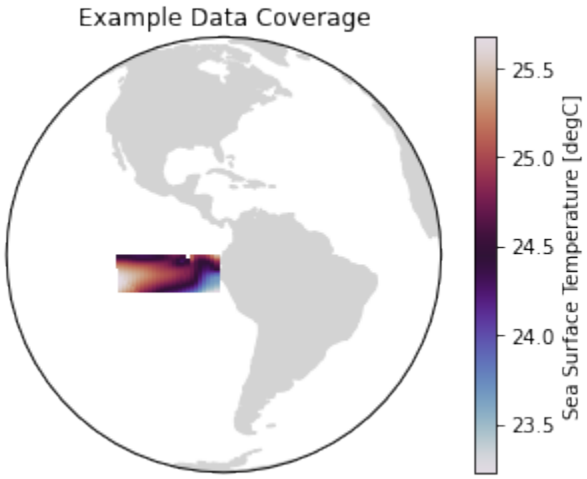
[5]:
```
ax = plt.axes(projection=ccrs.Orthographic(-80, 0))
p = ax.pcolormesh(recon.TLONG, recon.TLAT, recon.mean('time'),
                  transform=ccrs.PlateCarree(), cmap='twilight')
ax.add_feature(cfeature.LAND, color='#d3d3d3')
ax.set_global()
```

(continues on next page)

(continued from previous page)

```
plt.colorbar(p, label='Sea Surface Temperature [degC]')
ax.set(title='Example Data Coverage')
```

[5]:
```
[Text(0.5, 1.0, 'Example Data Coverage')]
```



We first need to remove the same climatology that was used to drift-correct the CESM-DPLE. Then we'll create a detrended version of our two products to assess detrended predictability.

[6]:
```
# Remove 1964-2014 climatology.
recon = recon - recon.sel(time=slice(1964, 2014)).mean('time')
```

We'll also detrend the reconstruction over its `time` dimension and initialized forecast ensemble over `init`.

[7]:
```
hind = climpred.stats.rm_trend(hind, dim='init')
recon = climpred.stats.rm_trend(recon, dim='time')
```

`climpred` requires that `lead` dimension has an attribute called `units` indicating what time units the `lead` is assocated with. Options are: `years`, `seasons`, `months`, `weeks`, `pentads`, `days`. For the this data, the `lead units` are `years`.

[8]:
```
hind['lead'].attrs['units'] = 'years'
```

Although functions can be called directly in `climpred`, we suggest that you use our classes (`HindcastEnsemble` and `PerfectModelEnsemble`) to make analysis code cleaner.

[9]:
```
hindcast = HindcastEnsemble(hind)
hindcast = hindcast.add_observations(recon, 'Reconstruction')
print(hindcast)
```

```
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST       (init, lead, nlat, nlon) float32 -0.29811984 ... 0.5265896
Reconstruction:
    SST       (time, nlat, nlon) float32 0.2235269 0.22273289 ... 1.3010706
Uninitialized:
    None
```

Anomaly Correlation Coefficient of SSTs

We can now compute the ACC over all leads and all grid cells.

[10]:
```
predictability = hindcast.verify(metric='acc')
print(predictability)
```

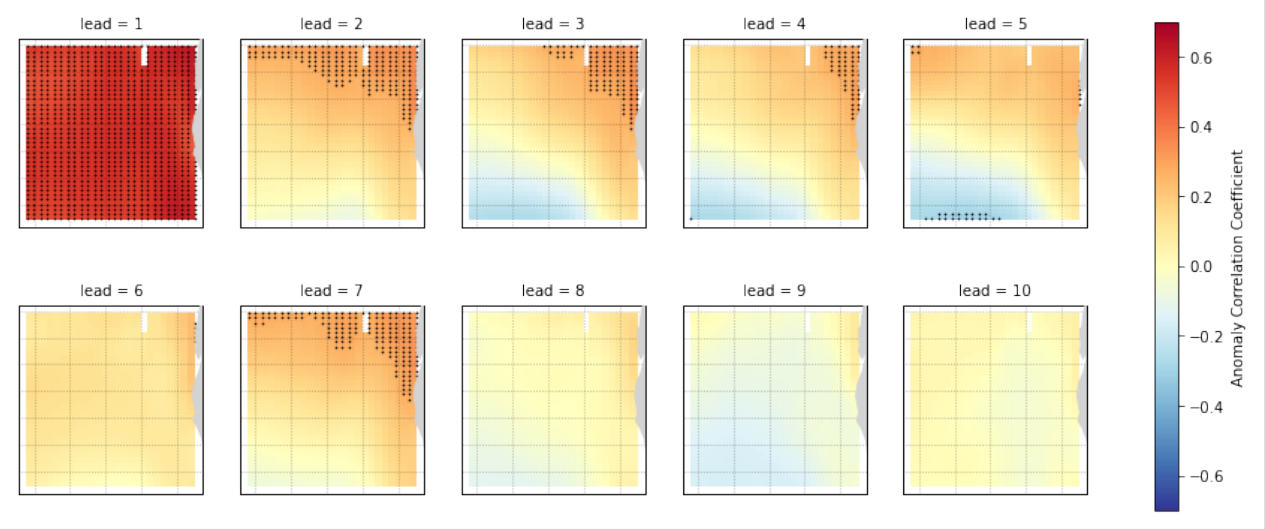
```
<xarray.Dataset>
Dimensions:  (lead: 10, nlat: 37, nlon: 26, skill: 1)
Coordinates:
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
    TAREA    (nlat, nlon) float64 3.661e+13 3.661e+13 ... 3.714e+13 3.714e+13
    TLONG    (nlat, nlon) float64 250.8 251.9 253.1 254.2 ... 276.7 277.8 278.9
  * nlat     (nlat) int64 0 1 2 3 4 5 6 7 8 9 ... 27 28 29 30 31 32 33 34 35 36
  * nlon     (nlon) int64 0 1 2 3 4 5 6 7 8 9 ... 16 17 18 19 20 21 22 23 24 25
  * skill    (skill) <U4 'init'
Data variables:
    SST      (lead, nlat, nlon) float32 0.5079418 0.5048633 ... 0.088374
```

We use the `pval` keyword to get associated p-values for our ACCs. We can then mask our final maps based on $\alpha = 0.05$.

[11]:
```
significance = hindcast.verify(metric='p_pval')
significance.coords['TLAT'] = hind['TLAT']
predictability.coords['TLAT'] = hind['TLAT']
```

[12]:
```
# Mask latitude and longitude by significance for stippling.
siglat = significance.TLAT.where(significance.SST <= 0.05)
siglon = significance.TLONG.where(significance.SST <= 0.05)
```

[13]:
```
p = predictability.SST.plot.pcolormesh(x='TLONG', y='TLAT',
                                       transform=ccrs.PlateCarree(),
                                       col='lead', col_wrap=5,
                                       subplot_kws={'projection': ccrs.PlateCarree(),
                                                    'aspect': 3},
                                       cbar_kwargs={'label': 'Anomaly Correlation␣
↪Coefficient'},
                                       vmin=-0.7, vmax=0.7,
                                       cmap='RdYlBu_r')
for i, ax in enumerate(p.axes.flat):
    ax.add_feature(cfeature.LAND, color='#d3d3d3', zorder=4)
    ax.gridlines(alpha=0.3, color='k', linestyle=':')
    # Add significance stippling
    ax.scatter(siglon.isel(lead=i),
               siglat.isel(lead=i),
               color='k',
               marker='.',
               s=1.5,
               transform=ccrs.PlateCarree())
```

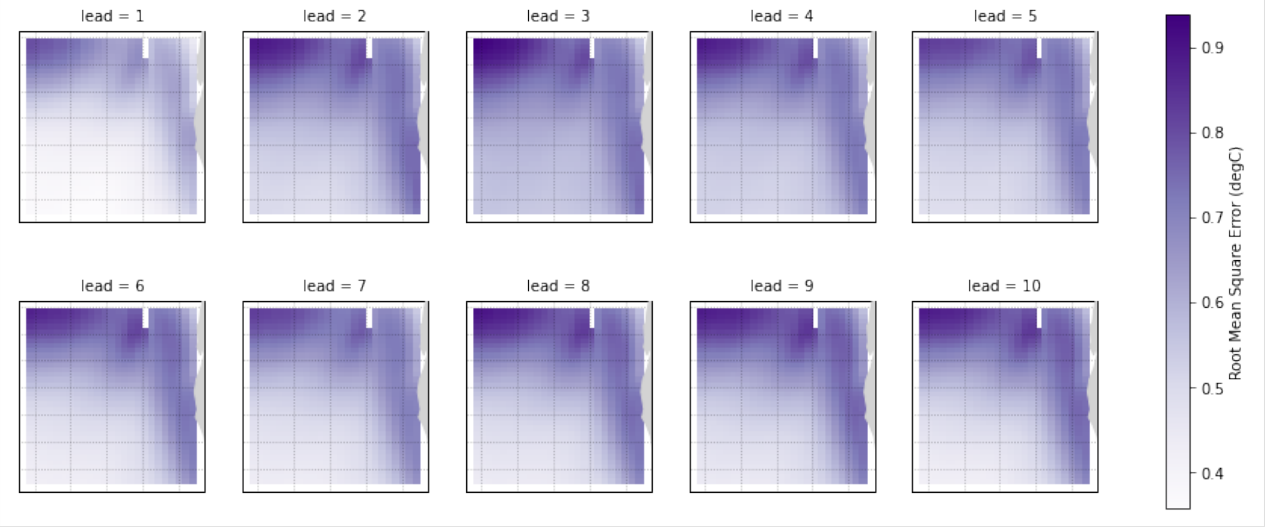


Root Mean Square Error of SSTs

We can also check error in our forecasts, just by changing the metric keyword.

[14]:
```
rmse = hindcast.verify(metric='rmse')
rmse.coords['TLAT'] = hind['TLAT']
```

[15]:
```
p = rmse.SST.plot.pcolormesh(x='TLONG', y='TLAT',
                             transform=ccrs.PlateCarree(),
                             col='lead', col_wrap=5,
                             subplot_kws={'projection': ccrs.PlateCarree(),
                                          'aspect': 3},
                             cbar_kwargs={'label': 'Root Mean Square Error (degC)'},
                             cmap='Purples')
for ax in p.axes.flat:
    ax.add_feature(cfeature.LAND, color='#d3d3d3', zorder=4)
    ax.gridlines(alpha=0.3, color='k', linestyle=':')
```



Diagnosing Potential Predictability

This demo demonstrates `climpred`'s capabilities to diagnose areas containing potentially predictable variations from a control/verification product alone without requiring multi-member, multi-initialization simulations. This notebook identifies the slow components of internal variability that indicate potential predictability. Here, we showcase a set of methods to show regions indicating probabilities for decadal predictability.

[1]:
```
import warnings
%matplotlib inline
import climpred
warnings.filterwarnings("ignore")
```

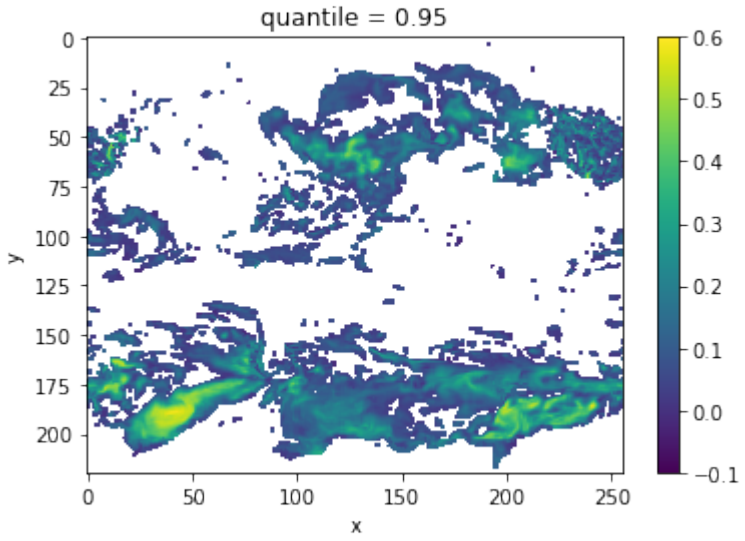
[2]:
```
# Sea surface temperature
varname='tos'
control3d = climpred.tutorial.load_dataset('MPI-control-3D')[varname].load()
```

Diagnostic Potential Predictability (DPP)

We can first use the [Resplandy 2015] and [Seferian 2018] method for computing the unbiased DPP by not chunking the time dimension.

[3]:
```
# calculate DPP with m=10
DPP10 = climpred.stats.dpp(control3d, m=10, chunk=False)
# calculate a threshold by random shuffling (based on bootstrapping with replacement␣
↪at 95% significance level)
threshold = climpred.bootstrap.dpp_threshold(control3d,
                                             m=10,
                                             chunk=False,
                                             iterations=10,
                                             sig=95)
# plot grid cells where DPP above threshold
DPP10.where(DPP10 > threshold).plot(yincrease=False, vmin=-0.1, vmax=0.6, cmap=
↪'viridis')
```

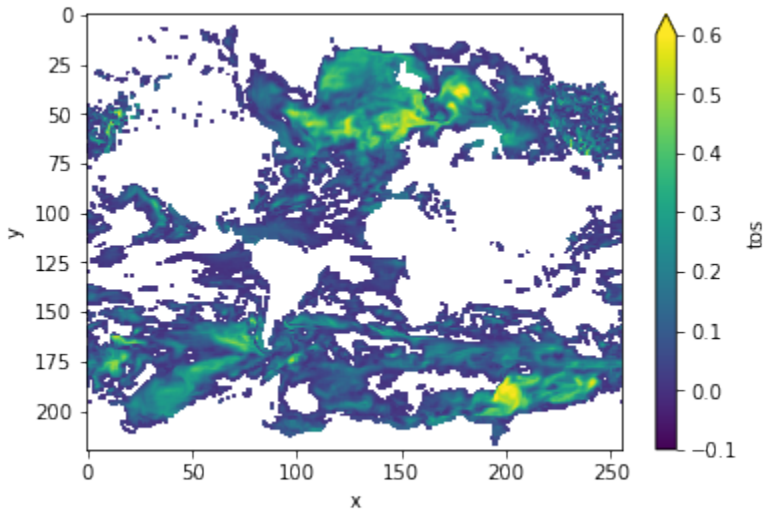
[3]:
```
<matplotlib.collections.QuadMesh at 0x11b7e8a58>
```



Now, we can turn on chunking (the default for this function) to use the [Boer 2004] method.

[4]:
```
# chunk = True signals the Boer 2004 method
DPP10 = climpred.stats.dpp(control3d, m=10, chunk=True)
threshold = climpred.bootstrap.dpp_threshold(control3d,
                                             m=10,
                                             chunk=True,
                                             iterations=10,
                                             sig=95)
DPP10.where(DPP10>0).plot(yincrease=False, vmin=-0.1, vmax=0.6, cmap='viridis')
```

[4]:
```
<matplotlib.collections.QuadMesh at 0x11b6e22b0>
```

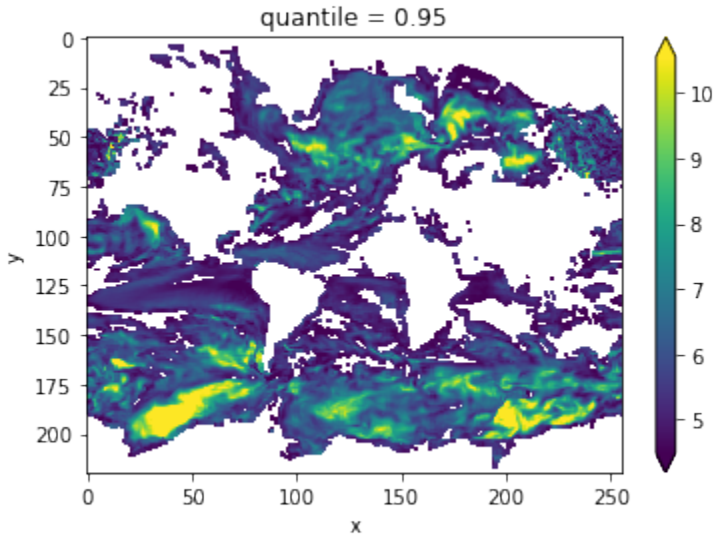


Variance-Weighted Mean Period

A periodogram is computed based on a control simulation to extract the mean period of variations, which are weighted by the respective variance. Regions with a high mean period value indicate low-frequency variations with are potentially predictable [Branstator2010].

[5]:
```
vwmp = climpred.stats.varweighted_mean_period(control3d, dim='time')
threshold = climpred.bootstrap.varweighted_mean_period_threshold(control3d,
                                                                 iterations=10)
vwmp.where(vwmp > threshold).plot(yincrease=False, robust=True)
```

[5]:
```
<matplotlib.collections.QuadMesh at 0x11b594358>
```

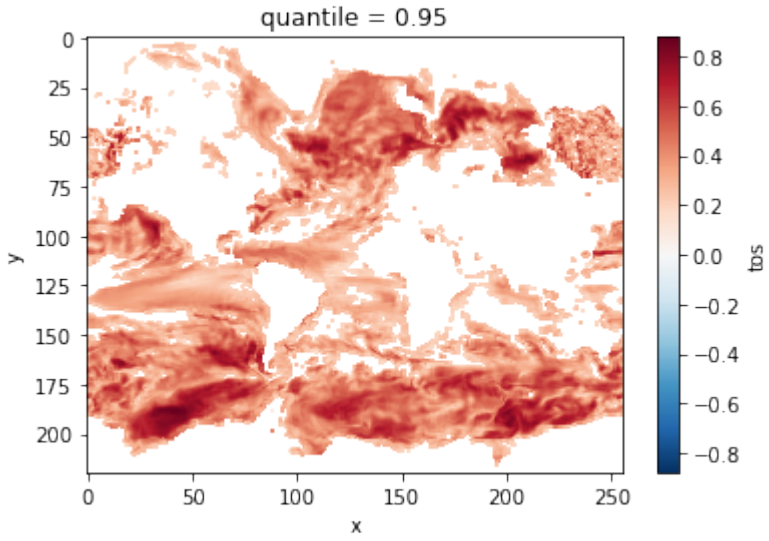


Lag-1 Autocorrelation

The lag-1 autocorrelation also indicates where slower modes of variability occur by identifying regions with high temporal correlation [vonStorch1999].

[6]:
```
# use climpred.bootstrap._bootstrap_func to wrap any stats function
threshold = climpred.bootstrap._bootstrap_func(climpred.stats.autocorr,control3d,'time
↪',iterations=10)
corr_ef = climpred.stats.autocorr(control3d, dim='time')
corr_ef.where(corr_ef>threshold).plot(yincrease=False, robust=False)
```

[6]:
```
<matplotlib.collections.QuadMesh at 0x11eba6278>
```

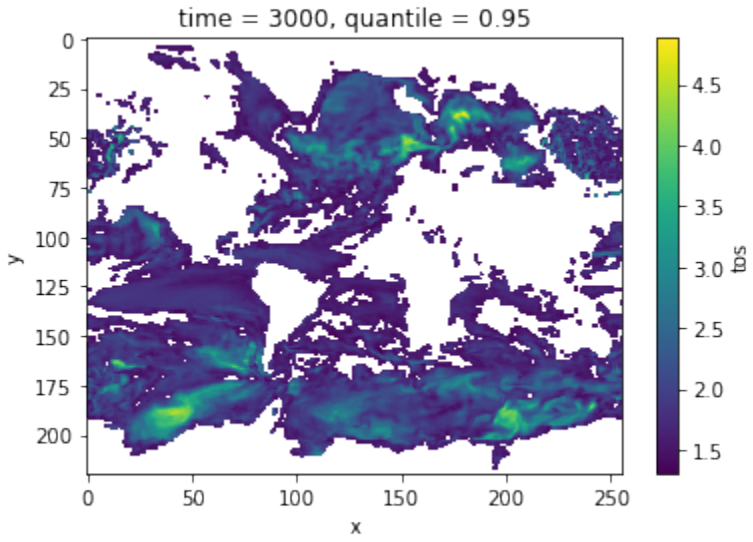


Decorrelation time

Taking the lagged correlation further over all lags, the decorrelation time shows the time after which the autocorrelation fell beyond its e-folding [vonStorch1999]

[7]:
```
threshold = climpred.bootstrap._bootstrap_func(climpred.stats.decorrelation_time,
↪control3d,'time',iterations=10)
decorr_time = climpred.stats.decorrelation_time(control3d)
decorr_time.where(decorr_time>threshold).plot(yincrease=False, robust=False)
```

[7]:
```
<matplotlib.collections.QuadMesh at 0x11eff20b8>
```



Verify diagnostic potential predictability in predictability simulations

Do we find predictability in the areas highlighted above also in perfect-model experiments?

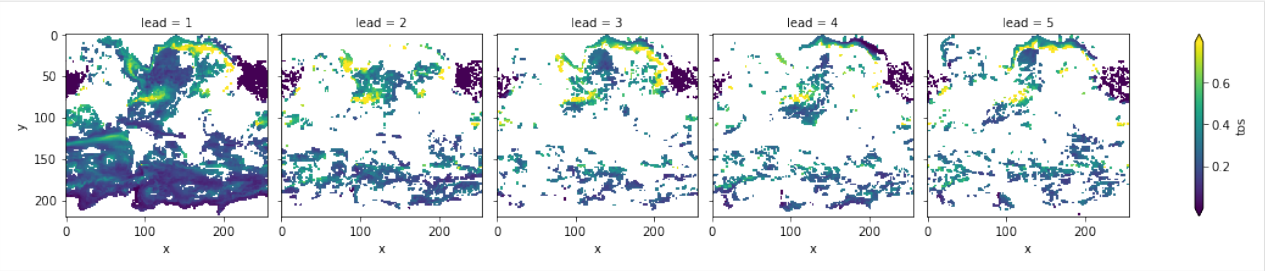
[8]:
```
ds3d = climpred.tutorial.load_dataset('MPI-PM-DP-3D')[varname].load()
ds3d['lead'].attrs['units'] = 'years'
```

[9]:
```
bootstrap_skill = climpred.bootstrap.bootstrap_perfect_model(ds3d,
                                                             control3d,
                                                             metric='rmse',
                                                             comparison='m2e',
                                                             iterations=10)
```

[10]:
```
init_skill = bootstrap_skill.sel(results='skill',kind='init')
# p value: probability that random uninitialized forecasts perform better than␣
↪initialized
p = bootstrap_skill.sel(results='p',kind='uninit')
```

[11]:
```
init_skill.where(p<=.05).plot(col='lead', robust=True, yincrease=False)
```

[11]:
```
<xarray.plot.facetgrid.FacetGrid at 0x11e837e48>
```



The metric `rmse` is negatively oriented, e.g. higher values show large disprepancy between members and hence less skill.

As suggested by DPP, the variance-weighted mean period and autocorrelation, also in slight perturbed initial values ensembles there is predictability in the North Atlantic, North Pacific and Southern Ocean in sea-surface temperatures.

References

1. Boer, Georges J. "Long time-scale potential predictability in an ensemble of coupled climate models." Climate dynamics 23.1 (2004): 29-44.
2. Resplandy, Laure, R. Séférian, and L. Bopp. "Natural variability of CO2 and O2 fluxes: What can we learn from centuries-long climate models simulations?." Journal of Geophysical Research: Oceans 120.1 (2015): 384-404.
3. Séférian, Roland, Sarah Berthet, and Matthieu Chevallier. "Assessing the Decadal Predictability of Land and Ocean Carbon Uptake." Geophysical Research Letters, March 15, 2018. https://doi.org/10/gdb424.
4. Branstator, Grant, and Haiyan Teng. "Two Limits of Initial-Value Decadal Predictability in a CGCM." Journal of Climate 23, no. 23 (August 27, 2010): 6292–6311. https://doi.org/10/bwq92h.
5. Storch, H. v, and Francis W. Zwiers. Statistical Analysis in Climate Research. Cambridge; New York: Cambridge University Press, 1999.

Temporal and spatial smoothing

This demo demonstrates `climpred`'s capabilities to postprocess decadal prediction output before skill verification. Here, we showcase a set of methods to smooth out noise in the spatial and temporal domain.

[1]:
```
import warnings
%matplotlib inline
import climpred
warnings.filterwarnings("ignore")
```

[2]:
```
# Sea surface temperature
varname='tos'
ds3d = climpred.tutorial.load_dataset('MPI-PM-DP-3D')[varname]
control3d = climpred.tutorial.load_dataset('MPI-control-3D')[varname]
```

`climpred` requires that `lead` dimension has an attribute called `units` indicating what time units the `lead` is assocated with. Options are: `years`, `seasons`, `months`, `weeks`, `pentads`, `days`. For the this data, the `lead units` are `years`.

[3]:
```
ds3d['lead'].attrs={'units': 'years'}
```

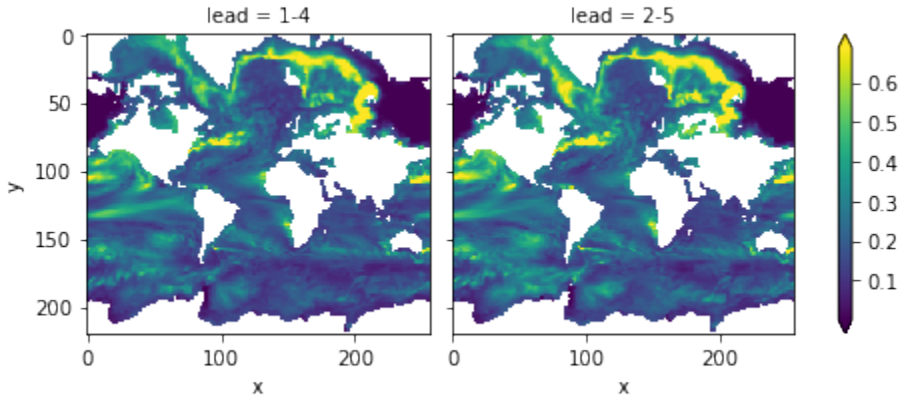
Temporal smoothing

In order to reduce temporal noise, decadal predictions are recommended to take multi-year averages [Goddard2013].

[4]:
```
ds3d_ts = climpred.smoothing.temporal_smoothing(ds3d,smooth_kws={'lead':4})
control3d_ts = climpred.smoothing.temporal_smoothing(control3d, smooth_kws={'time':4})
```

[5]:
```
climpred.prediction.compute_perfect_model(ds3d_ts,
                                          control3d_ts,
                                          metric='rmse',
                                          comparison='m2e') \
                    .plot(col='lead', robust=True, yincrease=False)
```

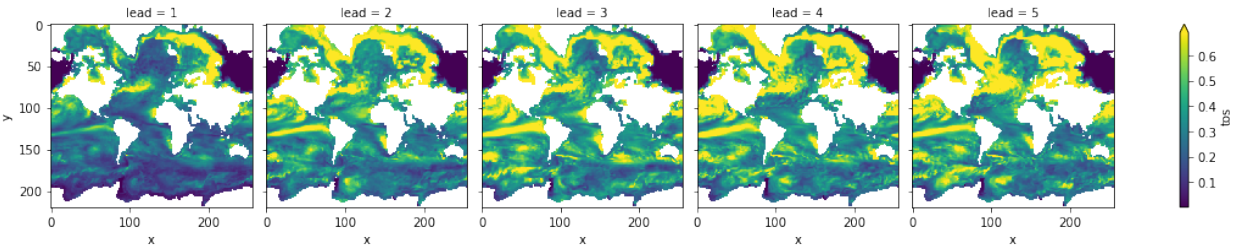
[5]:
```
<xarray.plot.facetgrid.FacetGrid at 0x12524c358>
```



Compare to without smoothing:

[6]:
```
climpred.prediction.compute_perfect_model(ds3d,
                                          control3d,
                                          metric='rmse',
                                          comparison='m2e') \
                    .plot(col='lead', vmax=.69, yincrease=False)
```

[6]:
```
<xarray.plot.facetgrid.FacetGrid at 0x124d6f518>
```



Note: When using `temporal_smoothing` on compute_hindcast, set rename_dim=False and after calculating the skill `_reset_temporal_axis` to get proper labeling of the lead dimension.

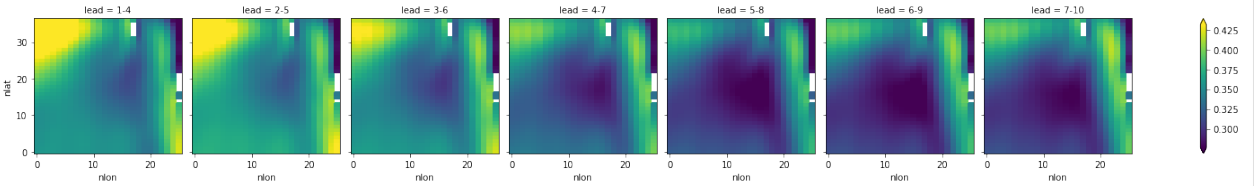
[7]:
```
hind = climpred.tutorial.load_dataset('CESM-DP-SST-3D').load()['SST']
reconstruction = climpred.tutorial.load_dataset('FOSI-SST-3D').load()['SST']
# get anomaly reconstruction
reconstruction = reconstruction - reconstruction.mean('time')
```

[8]:
```
hind_ts = climpred.smoothing.temporal_smoothing(hind,smooth_kws={'lead':4},rename_
↪dim=False)
reconstruction_ts = climpred.smoothing.temporal_smoothing(reconstruction, smooth_kws={
↪'time':4},rename_dim=False)
```

[9]:
```
# Lose units attribute along the way.
hind_ts["lead"].attrs["units"] = "years"
```

[10]:
```
s = climpred.prediction.compute_hindcast(hind_ts,
                                         reconstruction_ts,
                                         metric='rmse',
                                         comparison='e2r')
s = climpred.smoothing._reset_temporal_axis(s,smooth_kws={'lead':4})
s.plot(col='lead', robust=True)
```

[10]:
```
<xarray.plot.facetgrid.FacetGrid at 0x12c404198>
```



Spatial smoothing

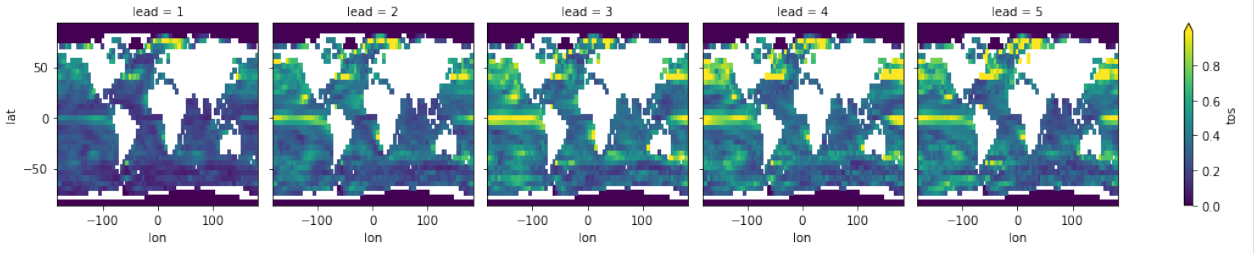
In order to reduce spatial noise, global decadal predictions are recommended to get regridded to a 5 degree longitude x 5 degree latitude grid as recommended [Goddard2013].

[11]:
```
ds3d_ss = climpred.smoothing.spatial_smoothing_xesmf(ds3d,d_lon_lat_kws={'lon':5, 'lat
↪':5})
control3d_ss = climpred.smoothing.spatial_smoothing_xesmf(control3d, d_lon_lat_kws={
↪'lon':5,'lat':5})
```

```
Create weight file: bilinear_220x256_36x73.nc
Reuse existing file: bilinear_220x256_36x73.nc
```

[12]:
```
climpred.prediction.compute_perfect_model(ds3d_ss,
                                          control3d_ss,
                                          metric='rmse',
                                          comparison='m2e') \
                    .plot(col='lead', robust=True, yincrease=True)
```

[12]:
```
<xarray.plot.facetgrid.FacetGrid at 0x12d54ae80>
```



Alternatively, also `climpred.smoothing.spatial_smoothing_xrcoarsen` aggregates gridcells like `xr_coarsen`.

`smooth_goddard2013` creates 4-year means and 5x5 degree regridding as suggested in [Goddard2013].

[13]:
```
climpred.smoothing.smooth_goddard_2013(ds3d).coords
```

```
Reuse existing file: bilinear_220x256_36x73.nc
```

[13]:
```
Coordinates:
  * lead     (lead) <U3 '1-4' '2-5'
  * init     (init) int64 3014 3061 3175 3237
  * member   (member) int64 1 2 3 4
  * lon      (lon) float64 -180.0 -175.0 -170.0 -165.0 ... 170.0 175.0 180.0
  * lat      (lat) float64 -83.97 -78.97 -73.97 -68.97 ... 81.03 86.03 91.03
```

References

1. Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

Significance Testing

This demo shows how to handle significance testing from a functional perspective of `climpred`. In the future, we will have a robust significance testing framework inplemented with `HindcastEnsemble` and `PerfectModelEnsemble` objects.

[ ]:
```
import xarray as xr
from climpred.tutorial import load_dataset
from climpred.stats import rm_poly
from climpred.prediction import compute_hindcast, compute_perfect_model
from climpred.bootstrap import bootstrap_hindcast, bootstrap_perfect_model
import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings("ignore")
```

[2]:
```
# load data
v = "SST"
hind = load_dataset("CESM-DP-SST")[v]
hind.lead.attrs["units"] = "years"

hist = load_dataset("CESM-LE")[v]
hist = hist - hist.mean()

obs = load_dataset("ERSST")[v]
obs = obs - obs.mean()
```

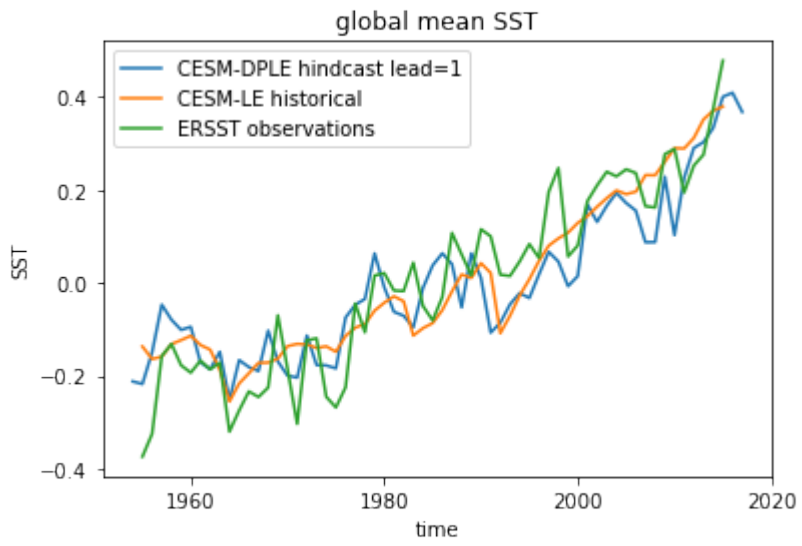
[3]:
```
# align times
hind["init"] = xr.cftime_range(
    start=str(hind.init.min().astype("int").values), periods=hind.init.size, freq="YS"
)
hist["time"] = xr.cftime_range(
    start=str(hist.time.min().astype("int").values), periods=hist.time.size, freq="YS"
)
obs["time"] = xr.cftime_range(
```

(continues on next page)

(continued from previous page)

```
    start=str(obs.time.min().astype("int").values), periods=obs.time.size, freq="YS"
)
```

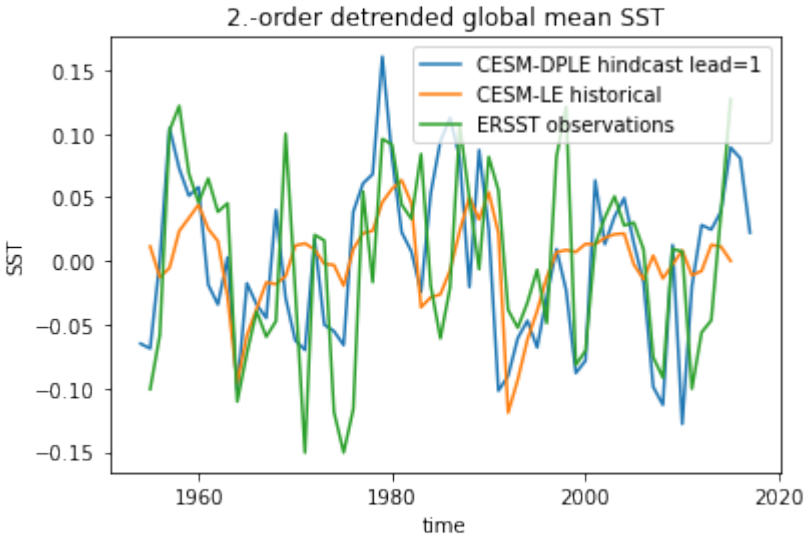
[4]:
```
# view the data
hind.sel(lead=1).mean("member").plot(label="CESM-DPLE hindcast lead=1")
hist.mean("member").plot(label="CESM-LE historical")
obs.plot(label="ERSST observations")
plt.legend()
plt.title(f"global mean {v}")
plt.show()
```



Here we see the strong trend due to climate change. This trend is not linear but rather quadratic. Because we often aim to prediction natural variations and not specifically the external forcing in initialized predictions, we remove the 2nd-order trend from each dataset along a time axis.

[5]:
```
order = 2
hind = rm_poly(hind, dim="init", order=order)
hist = rm_poly(hist, dim="time", order=order)
obs = rm_poly(obs, dim="time", order=order)
# lead attrs is lost in rm_poly
hind.lead.attrs["units"] = "years"
```

[6]:
```
hind.sel(lead=1).mean("member").plot(label="CESM-DPLE hindcast lead=1")
hist.mean("member").plot(label="CESM-LE historical")
obs.plot(label="ERSST observations")
plt.legend()
plt.title(f"{order}.-order detrended global mean {v}")
plt.show()
```



p value for temporal correlations

For correlation metrics the associated p-value checks whether the correlation is significantly different from zero incorporating reduced degrees of freedom due to temporal autocorrelation.

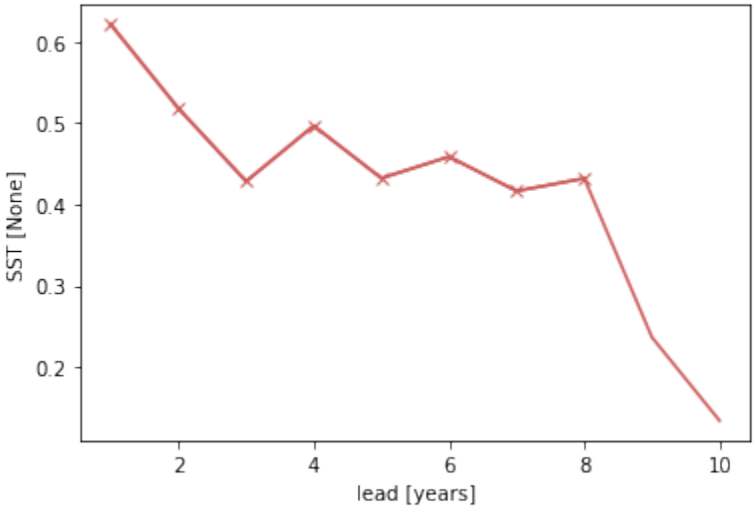
[7]:
```
# level that initialized ensembles are significantly better than other forecast skills
sig = 0.05
```

[8]:
```
acc = compute_hindcast(hind, obs, metric="pearson_r", comparison="e2r")

acc_p_value = compute_hindcast(hind, obs, metric="pearson_r_eff_p_value", comparison=
↪"e2r")
```

[9]:
```
init_color = "indianred"
acc.plot(c=init_color)
acc.where(acc_p_value <= sig).plot(marker="x", c=init_color)
```

[9]:
```
[<matplotlib.lines.Line2D at 0x1288a69b0>]
```



Bootstrapping with replacement

Bootstrapping significance relies on resampling the underlying data with replacement for a large number of iterations as proposed by the decadal prediction framework of Goddard et al. 2013. We just use 20 iterations here to demonstrate the functionality.

[10]:
```
%%time
bootstrapped_acc = bootstrap_hindcast(
    hind, hist, obs, metric="pearson_r", comparison="e2r", iterations=500, sig=95
)
```

```
CPU times: user 1.7 s, sys: 46.4 ms, total: 1.75 s
Wall time: 1.77 s
```

[11]:
```
bootstrapped_acc.coords
```

[11]:
```
Coordinates:
  * kind     (kind) object 'init' 'pers' 'uninit'
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
  * results  (results) <U7 'skill' 'p' 'low_ci' 'high_ci'
```

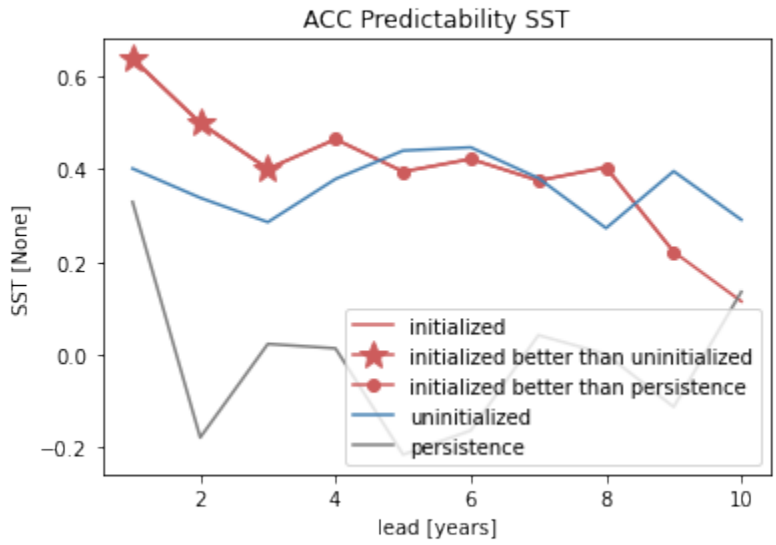
`bootstrap_acc` contains for the three different kinds of predictions: - `init` for the initialized hindcast `hind` and describes skill due to initialization and external forcing - `uninit` for the uninitialized historical `hist` and approximates skill from external forcing - `pers` for the reference forecast computed by `baseline_compute`, which defaults to `compute_persistence`

for different results: - `skill`: skill values - `p`: p value - `low_ci` and `high_ci`: high and low ends of confidence intervals based on significance threshold `sig`

[12]:
```
init_skill = bootstrapped_acc.sel(results="skill", kind="init")
init_better_than_uninit = init_skill.where(
    bootstrapped_acc.sel(results="p", kind="uninit") <= sig
)
init_better_than_persistence = init_skill.where(
    bootstrapped_acc.sel(results="p", kind="pers") <= sig
)
```

[13]:
```
# create a plot by hand
bootstrapped_acc.sel(results="skill", kind="init").plot(
    c=init_color, label="initialized"
)
init_better_than_uninit.plot(
    c=init_color,
    marker="*",
    markersize=15,
    label="initialized better than uninitialized",
)
init_better_than_persistence.plot(
    c=init_color, marker="o", label="initialized better than persistence"
)
bootstrapped_acc.sel(results="skill", kind="uninit").plot(
    c="steelblue", label="uninitialized"
)
bootstrapped_acc.sel(results="skill", kind="pers").plot(c="gray", label="persistence")
plt.title(f"ACC Predictability {v}")
plt.legend(loc="lower right")
```

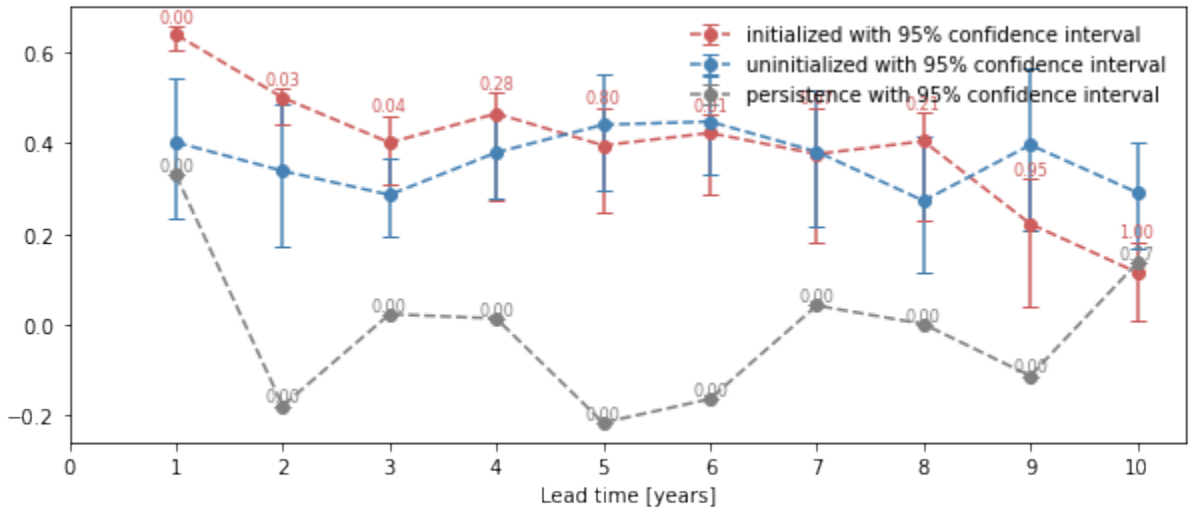
[13]:
```
<matplotlib.legend.Legend at 0x12a1f4320>
```



[14]:
```
# use climpred convenience plotting function
from climpred.graphics import plot_bootstrapped_skill_over_leadyear

plot_bootstrapped_skill_over_leadyear(bootstrapped_acc)
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1264bdc88>
```



Field significance

Using `esmtools.testing.multipletests` to control the false discovery rate (FDR) from the above obtained p-values in geospatial data.

[15]:
```
v = "tos"
ds3d = load_dataset("MPI-PM-DP-3D")[v]
ds3d.lead.attrs["unit"] = "years"
control3d = load_dataset("MPI-control-3D")[v]
```

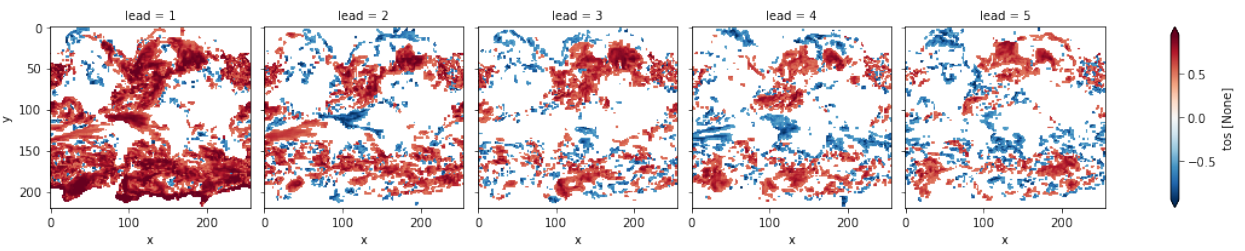
p value for temporal correlations

[16]:
```
# ACC skill
acc3d = compute_perfect_model(ds3d, control3d, metric="pearson_r", comparison="m2e")

# ACC_p_value skill
acc_p_3d = compute_perfect_model(
    ds3d, control3d, metric="pearson_r_p_value", comparison="m2e"
)
```

[17]:
```
# mask init skill where not significant
acc3d.where(acc_p_3d <= sig).plot(col="lead", robust=True, yincrease=False)
```

[17]:
```
<xarray.plot.facetgrid.FacetGrid at 0x128845b00>
```

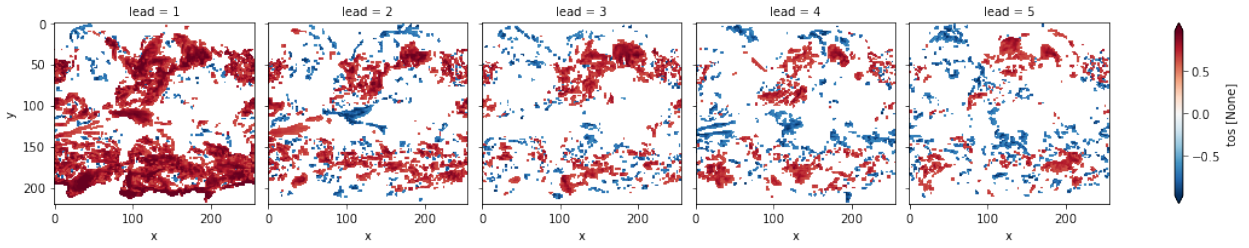


[18]:
```
# apply FDR Benjamini-Hochberg
# relies on esmtools https://github.com/bradyrx/esmtools
from esmtools.testing import multipletests

_, acc_p_3d_fdr_corr = multipletests(acc_p_3d, method="fdr_bh", alpha=sig)
```

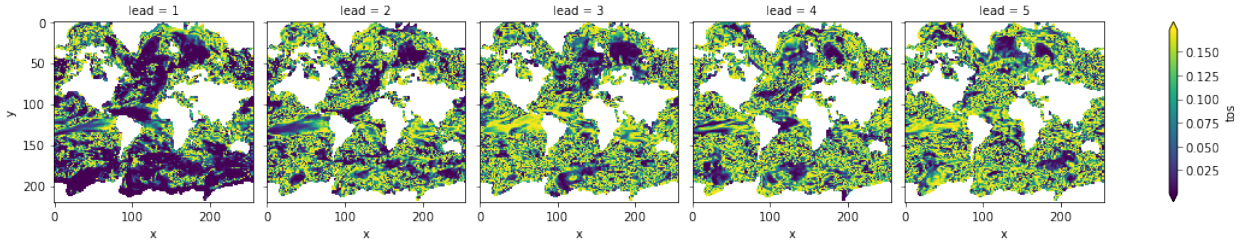
[19]:
```
# mask init skill where not significant on corrected p-values
acc3d.where(acc_p_3d_fdr_corr <= sig).plot(col="lead", robust=True, yincrease=False)
```

[19]:
```
<xarray.plot.facetgrid.FacetGrid at 0x122e5d2e8>
```



[20]:
```
# difference due to FDR Benjamini-Hochberg
(acc_p_3d_fdr_corr - acc_p_3d).plot(col="lead", robust=True, yincrease=False)
```

[20]:
```
<xarray.plot.facetgrid.FacetGrid at 0x123048dd8>
```



FDR Benjamini-Hochberg increases the p-value and therefore reduces the number of significant grid cells.

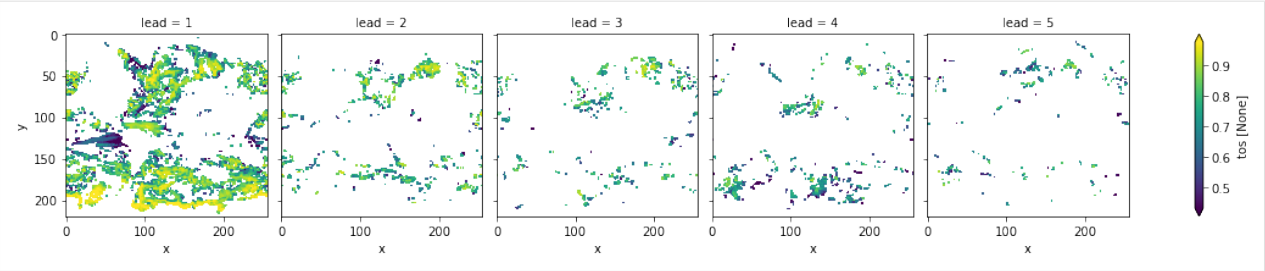
Bootstrapping with replacement

[37]:
```
%%time
bootstrapped_acc_3d = bootstrap_perfect_model(
    ds3d, control3d, metric="pearson_r", comparison="m2e", iterations=10
)
```

```
CPU times: user 22.2 s, sys: 12.7 s, total: 35 s
Wall time: 42.5 s
```

[46]:
```
# mask init skill where not significant
bootstrapped_acc_3d.sel(kind="init", results="skill").where(
    bootstrapped_acc_3d.sel(kind="uninit", results="p") <= sig
).plot(col="lead", robust=True, yincrease=False)
```

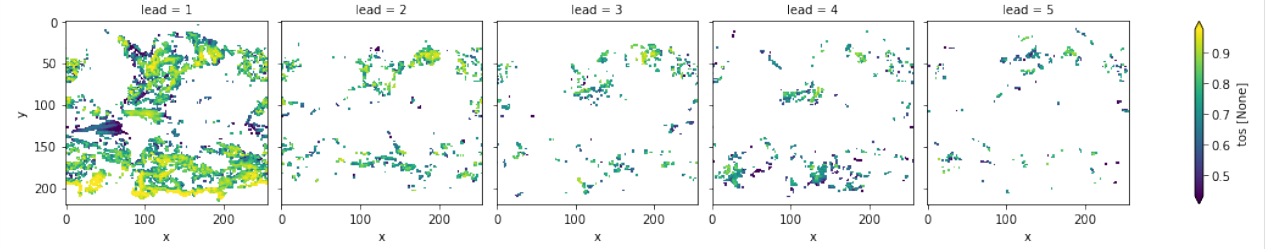
[46]:
```
<xarray.plot.facetgrid.FacetGrid at 0x141955400>
```



[66]:
```
# apply FDR Benjamini-Hochberg
_, bootstrapped_acc_p_3d_fdr_corr = multipletests(
    bootstrapped_acc_3d.sel(kind="uninit", results="p"), method="fdr_bh", alpha=sig
)
```

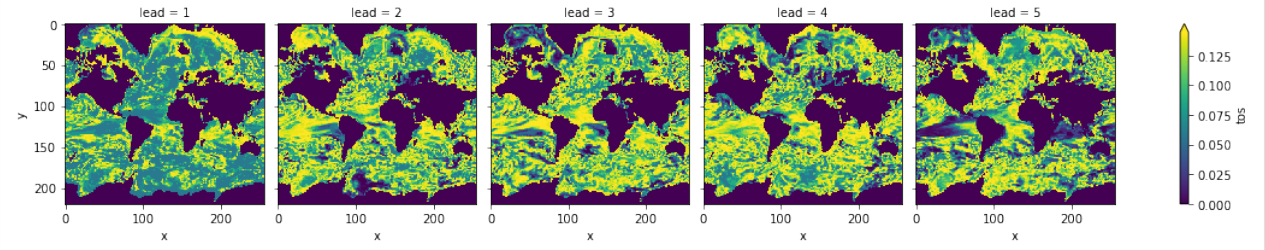
[75]:
```
# mask init skill where not significant on corrected p-values
bootstrapped_acc_3d.sel(kind="init", results="skill").where(
    bootstrapped_acc_p_3d_fdr_corr <= sig*2
).plot(col="lead", robust=True, yincrease=False)
```

[75]:
```
<xarray.plot.facetgrid.FacetGrid at 0x157e438d0>
```



[76]:
```
# difference due to FDR Benjamini-Hochberg
(
    bootstrapped_acc_p_3d_fdr_corr - bootstrapped_acc_3d.sel(kind="uninit", results="p
↪")
).plot(col="lead", robust=True, yincrease=False)
```

[76]:
```
<xarray.plot.facetgrid.FacetGrid at 0x158b34048>
```



FDR Benjamini-Hochberg increases the p-value and therefore reduces the number of significant grid cells.

User Guide

- *Setting Up Your Dataset*
- *PredictionEnsemble Objects*
- *Verification Alignment*
- *Metrics*

- *Comparisons*
- *Significance Testing*
- *Prediction Terminology*
- *Reference Forecasts*

2.5 Setting Up Your Dataset

`climpred` relies on a consistent naming system for `xarray` dimensions. This allows things to run more easily under-the-hood.

Prediction ensembles are expected at the minimum to contain dimensions `init` and `lead`. `init` is the initialization dimension, that relays the time steps at which the ensemble was initialized. `init` must be of type ``int`, ``pd.DatetimeIndex``, or `xr.cftimeIndex`. If `init` is of type `int`, it is assumed to be annual data. A user warning is issues when this assumption is made. `lead` is the lead time of the forecasts from initialization. The units for the `lead` dimension must be specified in as an attribute. Valid options are `years, seasons, months, weeks, pentads, days`. Another crucial dimension is `member`, which holds the various ensemble members. Any additional dimensions will be passed through `climpred` without issue: these could be things like `lat`, `lon`, `depth`, etc.

Check out the demo to setup a `climpred`-ready prediction ensemble from your own data or via intake-esm from CMIP DCPP.

Verification products are expected to contain the `time` dimension at the minimum. For best use of `climpred`, their `time` dimension should cover the full length of `init` and the same calendar type as the accompanying prediction ensemble, if possible. The `time` dimension must be of type `int`, `pd.DatetimeIndex` or `xr.cftimeIndex`. `time` dimension of type int is assumed to be annual data. A user warning is issued when this assumption is made. These products can also include additional dimensions, such as `lat`, `lon`, `depth`, etc.

See the below table for a summary of dimensions used in `climpred`, and data types that `climpred` supports for them.

Short Name	Types	Long Name	Attribute(s)
`lead`	`int`	lead timestep after initialization, [`init`]	units (str) [years, seasons, months, weeks, pentads, days]
`init`	`int`, `pd.DatetimeIndex`, `xr.CFTimeIndex`	initialization: start date of experiment	None
`member`	`int`, `str`	ensemble member	None

2.6 PredictionEnsemble Objects

One of the major features of `climpred` is our objects that are based upon the `PredictionEnsemble` class. We supply users with a `HindcastEnsemble` and `PerfectModelEnsemble` object. We encourage users to take advantage of these high-level objects, which wrap all of our core functions. These objects don't comprehensively cover all functions yet, but eventually we'll deprecate direct access to the function calls in favor of the lightweight objects.

Briefly, we consider a `HindcastEnsemble` to be one that is initialized from some observational-like product (e.g., assimilated data, reanalysis products, or a model reconstruction). Thus, this object is built around comparing the initialized ensemble to various observational products. In contrast, a `PerfectModelEnsemble` is one that is initialized off of a model control simulation. These forecasting systems are not meant to be compared directly to

real-world observations. Instead, they provide a contained model environment with which to theoretically study the limits of predictability. You can read more about the terminology used in `climpred` here.

Let's create a demo object to explore some of the functionality and why they are much smoother to use than direct function calls.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import xarray as xr

from climpred import HindcastEnsemble
from climpred.tutorial import load_dataset
import climpred
```

We can pull in some sample data that is packaged with `climpred`.

[2]:
```
load_dataset()
```

```
'MPI-control-1D': area averages for the MPI control run of SST/SSS.
'MPI-control-3D': lat/lon/time for the MPI control run of SST/SSS.
'MPI-PM-DP-1D': perfect model decadal prediction ensemble area averages of SST/SSS/
↪AMO.
'MPI-PM-DP-3D': perfect model decadal prediction ensemble lat/lon/time of SST/SSS/AMO.
'CESM-DP-SST': hindcast decadal prediction ensemble of global mean SSTs.
'CESM-DP-SSS': hindcast decadal prediction ensemble of global mean SSS.
'CESM-DP-SST-3D': hindcast decadal prediction ensemble of eastern Pacific SSTs.
'CESM-LE': uninitialized ensemble of global mean SSTs.
'MPIESM_miklip_baseline1-hind-SST-global': hindcast initialized ensemble of global␣
↪mean SSTs
'MPIESM_miklip_baseline1-hist-SST-global': uninitialized ensemble of global mean SSTs
'MPIESM_miklip_baseline1-assim-SST-global': assimilation in MPI-ESM of global mean␣
↪SSTs
'ERSST': observations of global mean SSTs.
'FOSI-SST': reconstruction of global mean SSTs.
'FOSI-SSS': reconstruction of global mean SSS.
'FOSI-SST-3D': reconstruction of eastern Pacific SSTs
'GMAO-GEOS-RMM1': daily RMM1 from the GMAO-GEOS-V2p1 model for SubX
'RMM-INTERANN-OBS': observed RMM with interannual variablity included
```

2.6.1 HindcastEnsemble

We'll start out with a `HindcastEnsemble` demo, followed by a `PerfectModelEnsemble` case.

[3]:
```
hind = climpred.tutorial.load_dataset('CESM-DP-SST') # CESM-DPLE hindcast ensemble␣
↪output.
obs = climpred.tutorial.load_dataset('ERSST') # ERSST observations.
recon = climpred.tutorial.load_dataset('FOSI-SST') # Reconstruction simulation that␣
↪initialized CESM-DPLE.
```

We need to add a "units" attribute to the hindcast ensemble so that `climpred` knows how to interpret the lead units.

[4]:
```
hind["lead"].attrs["units"] = "years"
```

CESM-DPLE was drift-corrected prior to uploading the output, so we just need to subtract the climatology over the same period for our other products before building the object.

[5]:
```
obs = obs - obs.sel(time=slice(1964, 2014)).mean('time')
recon = recon - recon.sel(time=slice(1964, 2014)).mean('time')
```

Now we instantiate the `HindcastEnsemble` object and append all of our products to it.

[6]:
```
hindcast = HindcastEnsemble(hind) # Instantiate object by passing in our initialized␣
↪ensemble.
print(hindcast)
```

```
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST      (init, lead, member) float64 ...
Verification Data:
    None
Uninitialized:
    None
```

```
/Users/ribr5703/miniconda3/envs/climpred-dev/lib/python3.6/site-packages/climpred/
↪utils.py:141: UserWarning: Assuming annual resolution due to numeric inits. Change␣
↪init to a datetime if it is another resolution.
  'Assuming annual resolution due to numeric inits. '
```

Now we just use the `add_` methods to attach other objects. See the API here. **Note that we strive to make our conventions follow those of "xarray"'s**. For example, we don't allow inplace operations. One has to run `hindcast = hindcast.add_observations(...)` to modify the object upon later calls rather than just `hindcast.add_observations(...)`.

[7]:
```
hindcast = hindcast.add_observations(recon, 'reconstruction')
hindcast = hindcast.add_observations(obs, 'ERSST')
```

[8]:
```
print(hindcast)
```

```
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST      (init, lead, member) float64 ...
reconstruction:
    SST      (time) float64 -0.05064 -0.0868 -0.1396 ... 0.3023 0.3718 0.292
ERSST:
    SST      (time) float32 -0.40146065 -0.35238647 ... 0.34601402 0.45021248
Uninitialized:
    None
```

You can apply most standard `xarray` functions directly to our objects! `climpred` will loop through the objects and apply the function to all applicable `xarray.Datasets` within the object. If you reference a dimension that doesn't exist for the given `xarray.Dataset`, it will ignore it. This is useful, since the initialized ensemble is expected to have dimension `init`, while other products have dimension `time` (see more here).

Let's start by taking the ensemble mean of the initialized ensemble so our metric computations don't have to take the extra time on that later. I'm just going to use deterministic metrics here, so we don't need the individual ensemble members. Note that above our initialized ensemble had a `member` dimension, and now it is reduced.

[9]:
```
hindcast = hindcast.mean('member')
print(hindcast)
```

```
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST      (init, lead) float64 -0.2121 -0.1637 -0.1206 ... 0.7286 0.7532
reconstruction:
```

(continues on next page)

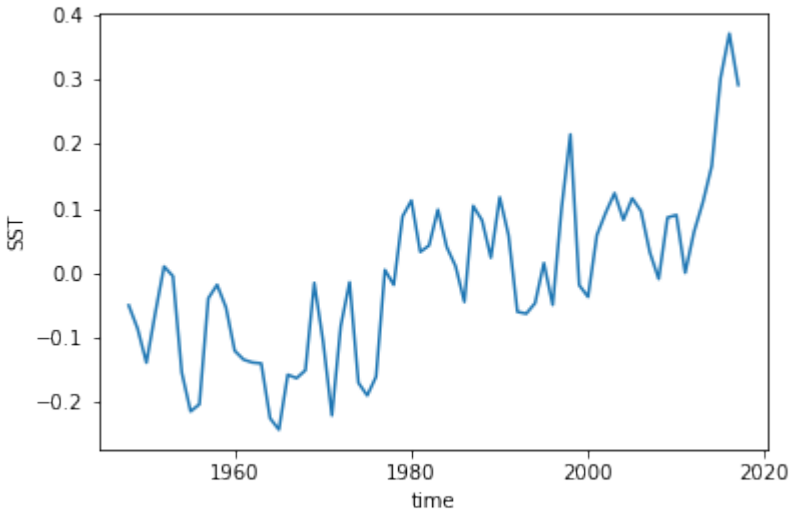
(continued from previous page)

```
    SST      (time) float64 -0.05064 -0.0868 -0.1396 ... 0.3023 0.3718 0.292
ERSST:
    SST      (time) float32 -0.40146065 -0.35238647 ... 0.34601402 0.45021248
Uninitialized:
    None
```

We still have a trend in all of our products, so we could also detrend them as well.

[10]:
```
hindcast.get_observations('reconstruction').SST.plot()
```

[10]:
```
[<matplotlib.lines.Line2D at 0x124152f60>]
```



[11]:
```
from scipy.signal import detrend
```

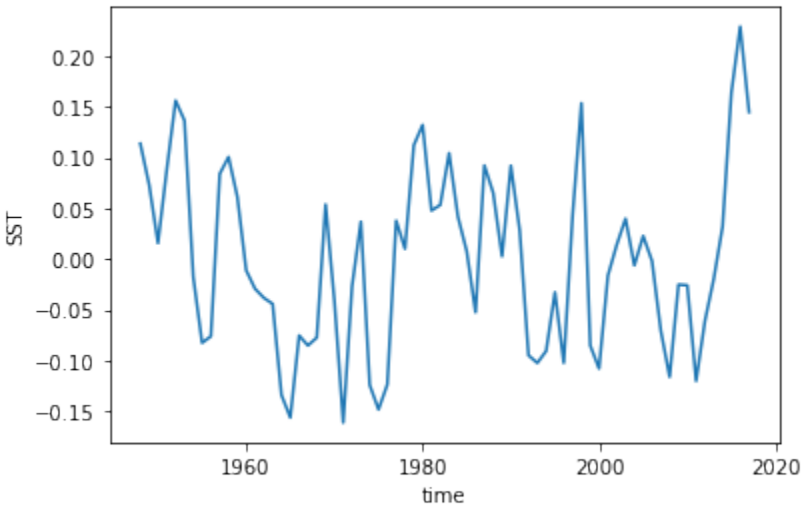
I'm going to transpose this first since my initialized ensemble has dimensions ordered `(init, lead)` and `scipy.signal.detrend` is applied over the last axis. I'd like to detrend over the `init` dimension rather than `lead` dimension.

[12]:
```
hindcast = hindcast.transpose().apply(detrend)
```

And it looks like everything got detrended by a linear fit! That wasn't too hard.

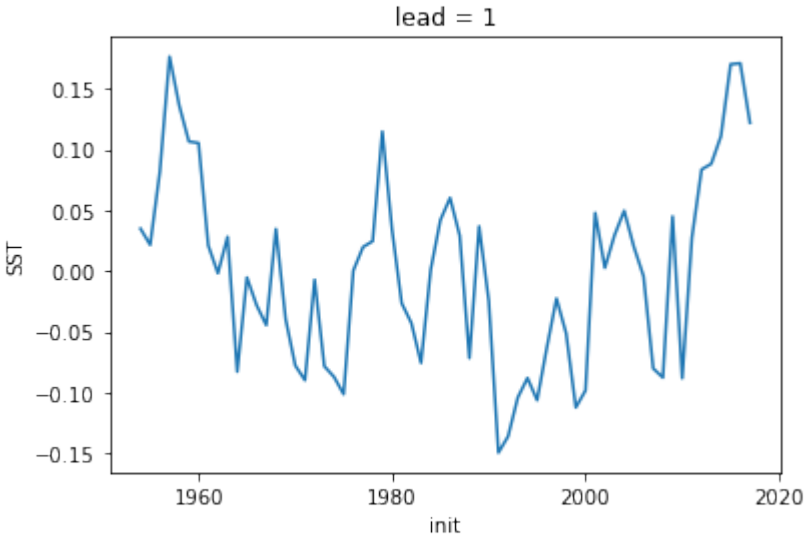
[13]:
```
hindcast.get_observations('reconstruction').SST.plot()
```

[13]:
```
[<matplotlib.lines.Line2D at 0x124618518>]
```



[14]:
```
hindcast.get_initialized().isel(lead=0).SST.plot()
```

[14]:
```
[<matplotlib.lines.Line2D at 0x1243d3668>]
```



Now that we've done our pre-processing, let's quickly compute some metrics. Check the metrics page here for all the keywords you can use. The API is currently pretty simple for the `HindcastEnsemble`. You can essentially compute standard skill metrics and a reference persistence forecast.

If you just pass a metric, it'll compute the skill metric against all observations and return a dictionary with keys of the names the user entered when adding them.

[15]:
```
hindcast.verify(metric='mse')
```

[15]:
```
{'reconstruction': <xarray.Dataset>
 Dimensions:  (lead: 10, skill: 1)
 Coordinates:
   * lead     (lead) int32 1 2 3 4 5 6 7 8 9 10
   * skill    (skill) <U4 'init'
 Data variables:
```

(continues on next page)

(continued from previous page)

```
    SST      (lead) float64 0.004468 0.007589 0.008514 ... 0.01054 0.01261,
'ERSST': <xarray.Dataset>
Dimensions:  (lead: 10, skill: 1)
Coordinates:
  * lead     (lead) int32 1 2 3 4 5 6 7 8 9 10
  * skill    (skill) <U4 'init'
Data variables:
    SST      (lead) float64 0.003911 0.005762 0.006707 ... 0.008239 0.009568}
```

One can also directly call individual observations to compare to. Here we leverage `xarray`'s plotting method to compute Mean Absolute Error and the Anomaly Correlation Coefficient for both our observational products, as well as the equivalent metrics computed for persistence forecasts for each of those metrics.

[16]:
```
import numpy as np

plt.style.use('ggplot')
plt.style.use('seaborn-talk')

RECON_COLOR = '#1b9e77'
OBS_COLOR = '#7570b3'

f, axs = plt.subplots(nrows=2, figsize=(8, 8), sharex=True)

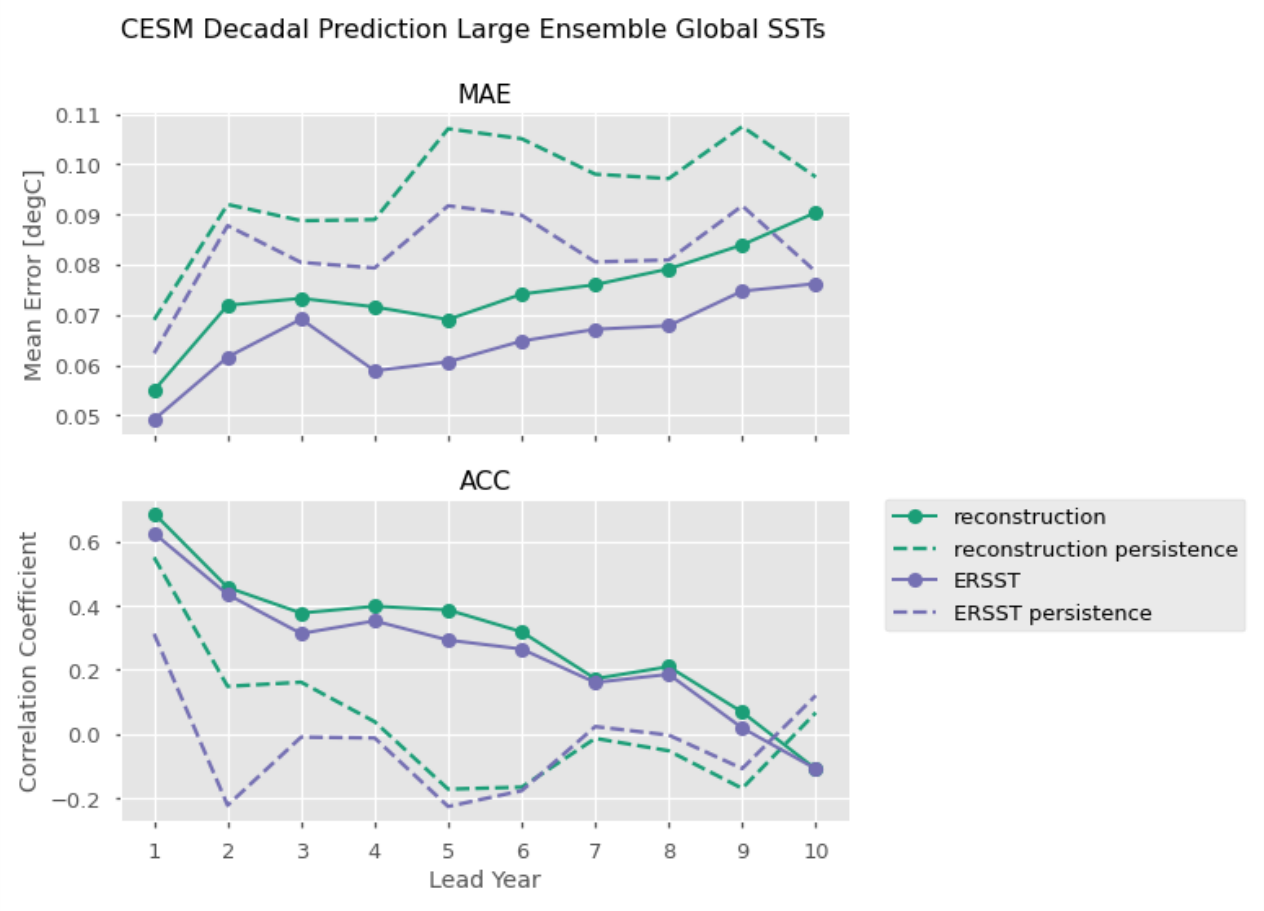
for ax, metric in zip(axs.ravel(), ['mae', 'acc']):
    handles = []
    for product, color in zip(['reconstruction', 'ERSST'], [RECON_COLOR, OBS_COLOR]):
        result = hindcast.verify(product, metric=metric, reference='persistence')
        p1, = result.sel(skill='init').SST.plot(ax=ax,
                                        marker='o',
                                        color=color,
                                        label=product,
                                        linewidth=2)
        p2, = result.sel(skill='persistence').SST.plot(ax=ax,
                                              color=color,
                                              linestyle='--',
                                              label=product + ' persistence')
        handles.append(p1)
        handles.append(p2)
    ax.set_title(metric.upper())


axs[0].set_ylabel('Mean Error [degC]')
axs[1].set_ylabel('Correlation Coefficient')
axs[0].set_xlabel('')
axs[1].set_xlabel('Lead Year')
axs[1].set_xticks(np.arange(10)+1)

# matplotlib/xarray returning weirdness for the legend handles.
handles = [i.get_label() for i in handles]

# a little trick to put the legend on the outside.
plt.legend(handles, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.0)

plt.suptitle('CESM Decadal Prediction Large Ensemble Global SSTs', fontsize=16)
plt.show()
```



2.6.2 PerfectModelEnsemble

We'll now play around a bit with the `PerfectModelEnsemble` object, using sample data from the MPI perfect model configuration.

[17]:
```
from climpred import PerfectModelEnsemble
```

[18]:
```
ds = climpred.tutorial.load_dataset('MPI-PM-DP-1D') # initialized ensemble from MPI
control = climpred.tutorial.load_dataset('MPI-control-1D') # base control run that␣
↪initialized it
```

[19]:
```
ds["lead"].attrs["units"] = "years"
```

[20]:
```
print(ds)
```

```
<xarray.Dataset>
Dimensions:  (area: 3, init: 12, lead: 20, member: 10, period: 5)
Coordinates:
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  * period   (period) object 'DJF' 'JJA' 'MAM' 'SON' 'ym'
  * area     (area) object 'global' 'North_Atlantic' 'North_Atlantic_SPG'
  * init     (init) int64 3014 3023 3045 3061 3124 ... 3175 3178 3228 3237 3257
  * member   (member) int64 0 1 2 3 4 5 6 7 8 9
```

(continues on next page)

(continued from previous page)

```
Data variables:
    tos      (period, lead, area, init, member) float32 ...
    sos      (period, lead, area, init, member) float32 ...
    AMO      (period, lead, area, init, member) float32 ...
```

[21]:
```
pm = climpred.PerfectModelEnsemble(ds)
pm = pm.add_control(control)
print(pm)
```

```
<climpred.PerfectModelEnsemble>
Initialized Ensemble:
    tos      (period, lead, area, init, member) float32 ...
    sos      (period, lead, area, init, member) float32 ...
    AMO      (period, lead, area, init, member) float32 ...
Control:
    tos      (period, time, area) float32 ...
    sos      (period, time, area) float32 ...
    AMO      (period, time, area) float32 ...
Uninitialized:
    None
```

```
/Users/ribr5703/miniconda3/envs/climpred-dev/lib/python3.6/site-packages/climpred/
↪utils.py:141: UserWarning: Assuming annual resolution due to numeric inits. Change␣
↪init to a datetime if it is another resolution.
  'Assuming annual resolution due to numeric inits. '
```

Our objects are carrying sea surface temperature (`tos`), sea surface salinity (`sos`), and the Atlantic Multidecadal Oscillation index (`AMO`). Say we just want to look at skill metrics for temperature and salinity over the North Atlantic in JJA. We can just call a few easy `xarray` commands to filter down our object.

[22]:
```
pm = pm.drop('AMO').sel(area='North_Atlantic', period='JJA')
```

Now we can easily compute for a host of metrics. Here I just show a number of deterministic skill metrics comparing all individual members to the initialized ensemble mean. See comparisons for more information on the `comparison` keyword.

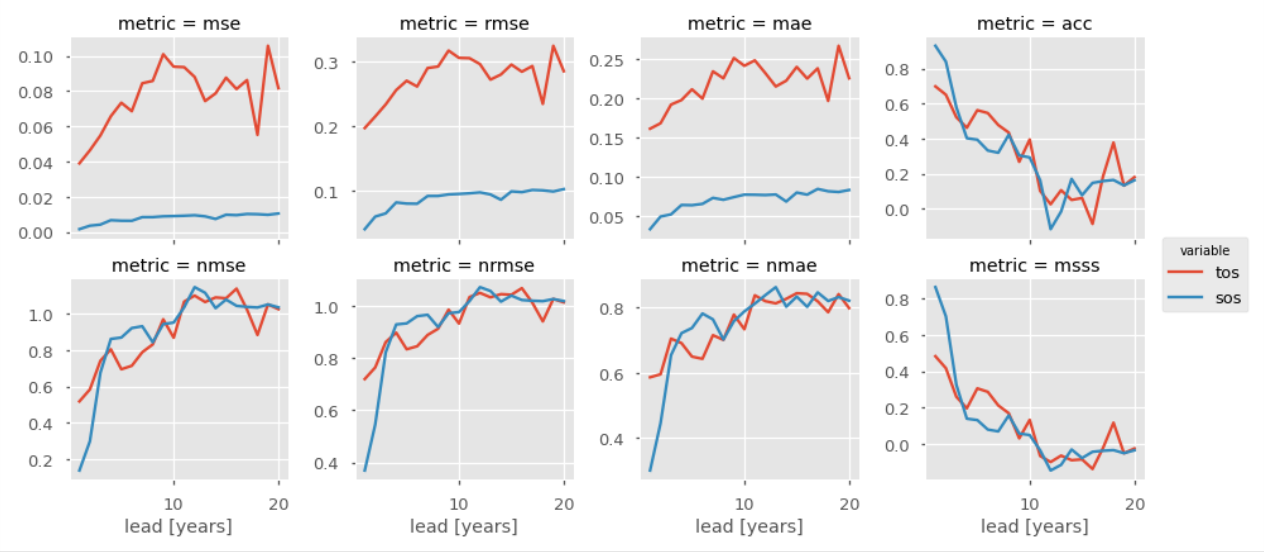
[23]:
```
METRICS = ['mse', 'rmse', 'mae', 'acc',
           'nmse', 'nrmse', 'nmae', 'msss']

result = []
for metric in METRICS:
    result.append(pm.compute_metric(metric, comparison='m2e'))

result = xr.concat(result, 'metric')
result['metric'] = METRICS

# Leverage the `xarray` plotting wrapper to plot all results at once.
result.to_array().plot(col='metric',
                       hue='variable',
                       col_wrap=4,
                       sharey=False,
                       sharex=True)
```

[23]:
```
<xarray.plot.facetgrid.FacetGrid at 0x124540940>
```



It is useful to compare the initialized ensemble to an uninitialized run. See terminology for a description on "uninitialized" simulations. This gives us information about how *initializations* lead to enhanced predictability over knowledge of external forcing, whereas a comparison to persistence just tells us how well a dynamical forecast simulation does in comparison to a naive method. We can use the `generate_uninitialized()` method to bootstrap the control run and create a pseudo-ensemble that approximates what an uninitialized ensemble would look like.

[24]:
```
pm = pm.generate_uninitialized()
print(pm)
```

```
<climpred.PerfectModelEnsemble>
Initialized Ensemble:
    tos       (lead, init, member) float32 13.464135 13.641711 ... 13.568891
    sos       (lead, init, member) float32 33.183903 33.146976 ... 33.25843
Control:
    tos       (time) float32 13.499312 13.742612 ... 13.076672 13.465583
    sos       (time) float32 33.232624 33.188156 33.201694 ... 33.16359 33.18352
Uninitialized:
    tos       (lead, init, member) float32 13.302329 13.270309 ... 13.211761
    sos       (lead, init, member) float32 33.223557 33.18921 ... 33.17692
```

[25]:
```
pm = pm.drop('tos') # Just assess for salinity.
```

Here I plot the ACC for the initialized, uninitialized, and persistence forecasts for North Atlantic sea surface salinity in JJA. I add circles to the lines if the correlations are statistically significant for $p <= 0.05$.

[26]:
```
# ACC for initialized ensemble
acc = pm.compute_metric('acc')
acc.sos.plot(color='red')
acc.where(pm.compute_metric('p_pval') <= 0.05).sos.plot(marker='o', linestyle='None',␣
↪color='red', label='initialized')

# ACC for 'uninitialized' ensemble
# acc = pm.compute_uninitialized('acc')
# acc.sos.plot(color='gray')
# acc.where(pm.compute_uninitialized('p_pval') <= 0.05).sos.plot(marker='o',␣
↪linestyle='None', color='gray', label='uninitialized')

# ACC for persistence forecast
```

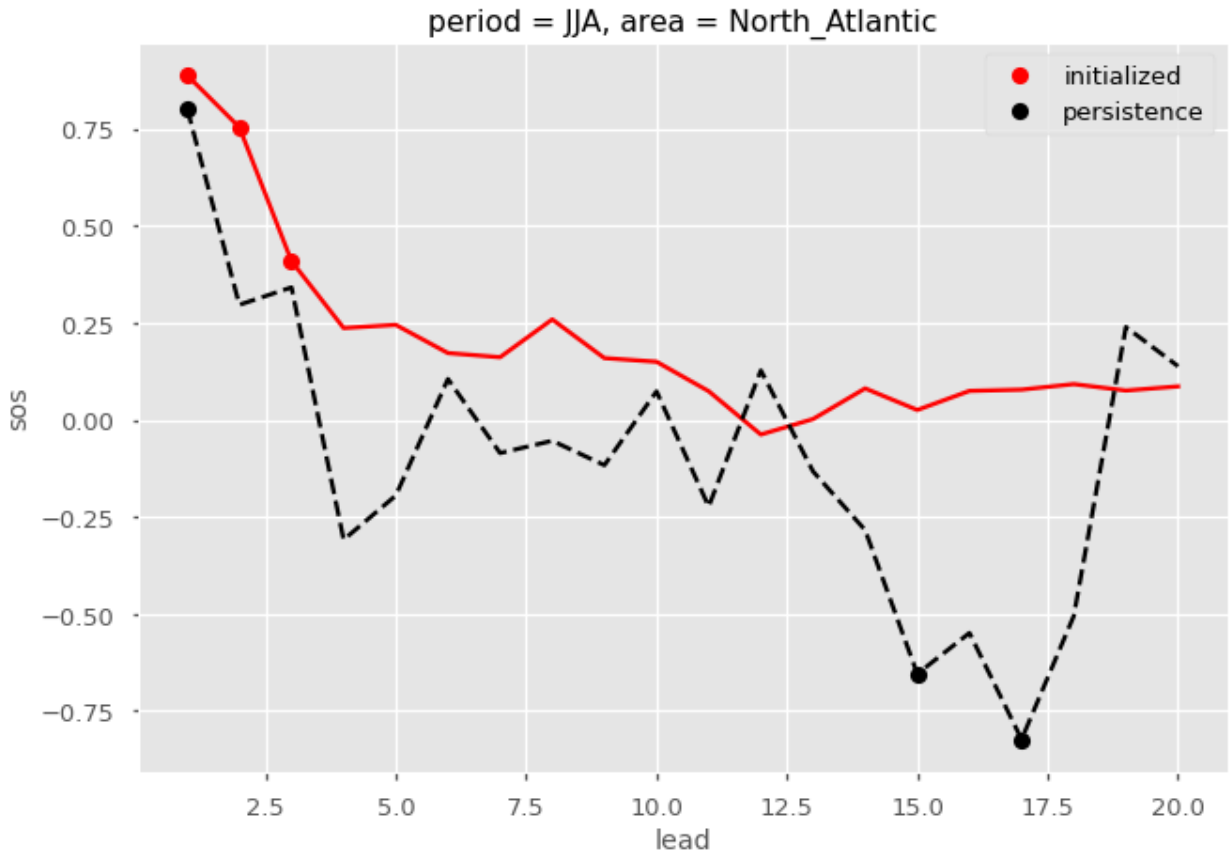
(continues on next page)

(continued from previous page)

```
acc = pm.compute_persistence('acc')
acc.sos.plot(color='k', linestyle='--')
acc.where(pm.compute_persistence('p_pval') <= 0.05).sos.plot(marker='o', linestyle=
↪'None', color='k', label='persistence')

plt.legend()
```

[26]: <matplotlib.legend.Legend at 0x12dccbe80>



2.7 Verification Alignment

A forecast is verified by comparing a set of initializations at a given lead to observations over some window of time. However, there are a few ways to decide *which* initializations or verification window to use in this alignment.

One can pass the keyword `alignment=...` to any hindcast compute functions (e.g., HindcastEnsemble.verify, compute_hindcast, compute_persistence) to change the behavior for aligning forecasts with the verification product. Note that the alignment decision only matters for hindcast experiments. Perfect-model experiments are perfectly time-aligned by design, equating to our `same_inits` keyword.

The available keywords for hindcast alignment are:

- `'same_inits'`: Use a common set of initializations that verify across all leads. This ensures that there is no bias in the result due to the state of the system for the given initializations.
- `'same_verifs'` (**default**): Use a common verification window across all leads. This ensures that there is no bias in the result due to the observational period being verified against.

- `'maximize'`: Use all available initializations at each lead that verify against the observations provided. This changes both the set of initializations and the verification window used at each lead.

[1]:
```
from climpred import HindcastEnsemble
from climpred.tutorial import load_dataset

import esmtools as et

import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
%matplotlib inline

import numpy as np

import warnings
# Supress datetime warnings for this page.
warnings.filterwarnings("ignore")
```

[2]:
```
def create_hindcast_object():
    """Loads in example data from CESM-DPLE and ERSST observations and
    bias-corrects and detrends."""
    hind = load_dataset('CESM-DP-SST')['SST']
    verif = load_dataset('ERSST')['SST']

    # Bias-correct over same period as CESM-DPLE.
    verif = verif - verif.sel(time=slice(1964, 2014)).mean('time')

    # Remove linear trend.
    hind_dt = et.stats.rm_trend(hind, 'init')
    verif_dt = et.stats.rm_trend(verif, 'time')

    # Create `HindcastEnsemble` object from `climpred`.
    hindcast = HindcastEnsemble(hind)
    hindcast = hindcast.add_observations(verif, 'ERSST')
    hindcast_dt = HindcastEnsemble(hind_dt)
    hindcast_dt = hindcast_dt.add_observations(verif_dt, 'ERSST')
    return hindcast, hindcast_dt
```

[3]:
```
hindcast, hindcast_dt = create_hindcast_object()
```

The user can simply change the alignment strategy by passing in the keyword `alignment=...`. Note that the choice of alignment strategy changes the lead-dependent metric results.

[4]:
```
f, axs = plt.subplots(ncols=2, figsize=(12,4), sharex=True)
for alignment in ['same_inits', 'same_verifs', 'maximize']:
    hindcast.verify(metric='acc', alignment=alignment)['SST'] \
            .plot(label=alignment, ax=axs[0])
    hindcast_dt.verify(metric='acc', alignment=alignment)['SST'] \
               .plot(label=alignment, ax=axs[1])

axs[0].legend()
axs[1].legend()
axs[0].set(ylabel='anomaly\ncorrelation coefficient',
       xlabel='lead year',
       xticks=np.arange(1, 11),
       title='SST with trend')
axs[1].set(ylabel='anomaly\ncorrelation coefficient',
```

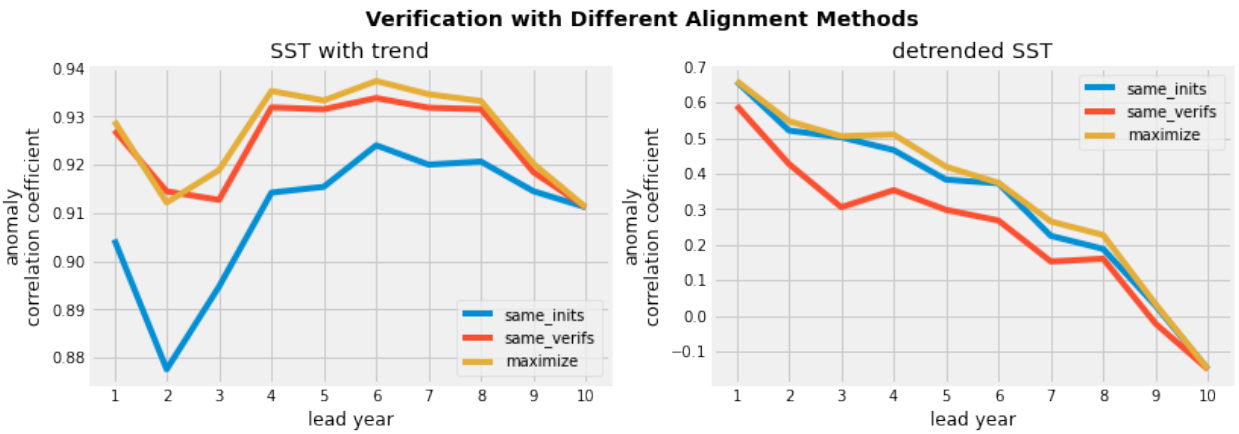
(continues on next page)

(continued from previous page)

```
            xlabel='lead year',
            title='detrended SST')

f.suptitle('Verification with Different Alignment Methods',
           fontsize=14, weight='bold')
plt.subplots_adjust(top=0.85)

plt.show()
```

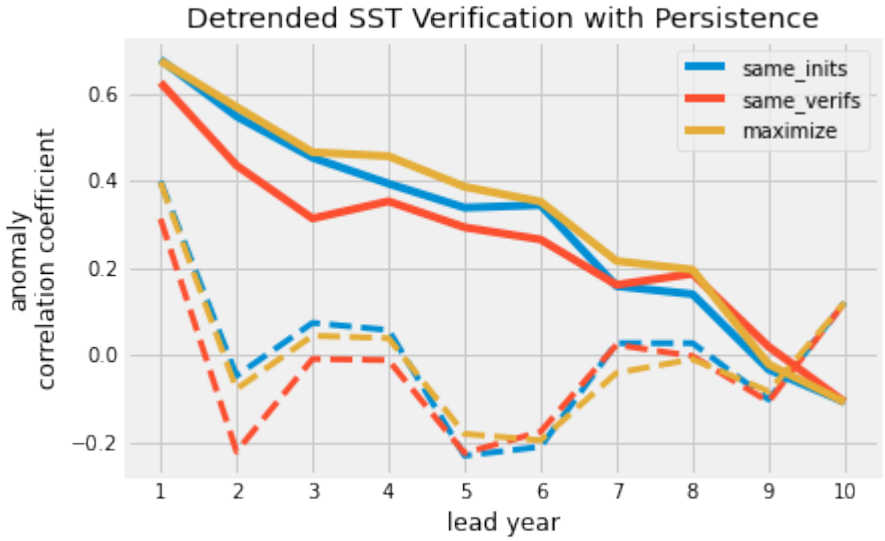


These alignment keywords also extend to `compute_persistence`, which uses the identical set of initializations (and alignment strategy) in its computation. Below, the dashed lines represent the persistence forecast for the given alignment strategy, while the solid lines denote the initialized anomaly correlation coefficient (as in the above plots).

[5]:
```
COLORS = ['#008FD5', '#FC4F30', '#E5AE38']
f, axs = plt.subplots()

for alignment, color in zip(['same_inits', 'same_verifs', 'maximize'], COLORS):
    result = hindcast_dt.verify(metric='acc', reference='persistence',␣
↪alignment=alignment)
    result.sel(skill='init').SST.plot(label=alignment, color=color)
    result.sel(skill='persistence').SST.plot(linestyle='--', color=color, lw=3)


axs.set(ylabel='anomaly\ncorrelation coefficient',
        xlabel='lead year',
        xticks=np.arange(1, 11),
        title='Detrended SST Verification with Persistence')
plt.legend()
plt.show()
```



We'll be using the same example data as above. `climpred` will be aligning the following initialization and verification dates:

[6]:
```
print(f'initialization dates: \n{hindcast.get_initialized().init.to_index()}')
```

```
initialization dates:
CFTimeIndex([1954-01-01 00:00:00, 1955-01-01 00:00:00, 1956-01-01 00:00:00,
             1957-01-01 00:00:00, 1958-01-01 00:00:00, 1959-01-01 00:00:00,
             1960-01-01 00:00:00, 1961-01-01 00:00:00, 1962-01-01 00:00:00,
             1963-01-01 00:00:00, 1964-01-01 00:00:00, 1965-01-01 00:00:00,
             1966-01-01 00:00:00, 1967-01-01 00:00:00, 1968-01-01 00:00:00,
             1969-01-01 00:00:00, 1970-01-01 00:00:00, 1971-01-01 00:00:00,
             1972-01-01 00:00:00, 1973-01-01 00:00:00, 1974-01-01 00:00:00,
             1975-01-01 00:00:00, 1976-01-01 00:00:00, 1977-01-01 00:00:00,
             1978-01-01 00:00:00, 1979-01-01 00:00:00, 1980-01-01 00:00:00,
             1981-01-01 00:00:00, 1982-01-01 00:00:00, 1983-01-01 00:00:00,
             1984-01-01 00:00:00, 1985-01-01 00:00:00, 1986-01-01 00:00:00,
             1987-01-01 00:00:00, 1988-01-01 00:00:00, 1989-01-01 00:00:00,
             1990-01-01 00:00:00, 1991-01-01 00:00:00, 1992-01-01 00:00:00,
             1993-01-01 00:00:00, 1994-01-01 00:00:00, 1995-01-01 00:00:00,
             1996-01-01 00:00:00, 1997-01-01 00:00:00, 1998-01-01 00:00:00,
             1999-01-01 00:00:00, 2000-01-01 00:00:00, 2001-01-01 00:00:00,
             2002-01-01 00:00:00, 2003-01-01 00:00:00, 2004-01-01 00:00:00,
             2005-01-01 00:00:00, 2006-01-01 00:00:00, 2007-01-01 00:00:00,
             2008-01-01 00:00:00, 2009-01-01 00:00:00, 2010-01-01 00:00:00,
             2011-01-01 00:00:00, 2012-01-01 00:00:00, 2013-01-01 00:00:00,
             2014-01-01 00:00:00, 2015-01-01 00:00:00, 2016-01-01 00:00:00,
             2017-01-01 00:00:00],
            dtype='object', name='init')
```

[7]:
```
print(f'verification dates: \n{hindcast.get_observations().time.to_index()}')
```

```
verification dates:
CFTimeIndex([1955-01-01 00:00:00, 1956-01-01 00:00:00, 1957-01-01 00:00:00,
             1958-01-01 00:00:00, 1959-01-01 00:00:00, 1960-01-01 00:00:00,
             1961-01-01 00:00:00, 1962-01-01 00:00:00, 1963-01-01 00:00:00,
             1964-01-01 00:00:00, 1965-01-01 00:00:00, 1966-01-01 00:00:00,
             1967-01-01 00:00:00, 1968-01-01 00:00:00, 1969-01-01 00:00:00,
```

(continues on next page)

(continued from previous page)

```
               1970-01-01 00:00:00, 1971-01-01 00:00:00, 1972-01-01 00:00:00,
               1973-01-01 00:00:00, 1974-01-01 00:00:00, 1975-01-01 00:00:00,
               1976-01-01 00:00:00, 1977-01-01 00:00:00, 1978-01-01 00:00:00,
               1979-01-01 00:00:00, 1980-01-01 00:00:00, 1981-01-01 00:00:00,
               1982-01-01 00:00:00, 1983-01-01 00:00:00, 1984-01-01 00:00:00,
               1985-01-01 00:00:00, 1986-01-01 00:00:00, 1987-01-01 00:00:00,
               1988-01-01 00:00:00, 1989-01-01 00:00:00, 1990-01-01 00:00:00,
               1991-01-01 00:00:00, 1992-01-01 00:00:00, 1993-01-01 00:00:00,
               1994-01-01 00:00:00, 1995-01-01 00:00:00, 1996-01-01 00:00:00,
               1997-01-01 00:00:00, 1998-01-01 00:00:00, 1999-01-01 00:00:00,
               2000-01-01 00:00:00, 2001-01-01 00:00:00, 2002-01-01 00:00:00,
               2003-01-01 00:00:00, 2004-01-01 00:00:00, 2005-01-01 00:00:00,
               2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
               2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
               2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
               2015-01-01 00:00:00],
              dtype='object', name='time')
```

We use the standard python library `logging` to log the initializations and verification dates used in alignment at each lead. The user can check these logs to ensure that the expected initializations and verification dates are being retained. See the logging section on this page for more details.

[8]:
```
import logging
# Print log to screen with initializations and verification dates.
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

2.7.1 Same Verification Dates

`alignment='same_verifs'` (**default**)

The `same_verifs` alignment finds a set of verification dates that can be verified against over all leads. It also requires that the verification data have an observation at each initialization being retained. This is so that the reference forecast, such as persistence, uses an identical set of initializations in deriving its forecast. Notice in the logger output that a common set of verification dates spanning 1965-2015 are used, while the initialization window slides one year at each lead.

References:

1. Boer, George J., et al. "The decadal climate prediction project (DCPP) contribution to CMIP6." Geoscientific Model Development (Online) 9.10 (2016). [https://doi.org/10.5194/gmd-9-3751-2016]
2. Hawkins, Ed, et al. "The interpretation and use of biases in decadal climate predictions." Journal of climate 27.8 (2014): 2931-2947. [https://doi.org/10.1175/JCLI-D-13-00473.1]
3. Smith, Doug M., Rosie Eade, and Holger Pohlmann. "A comparison of full-field and anomaly initialization for seasonal to decadal climate prediction." Climate dynamics 41.11-12 (2013): 3325-3338. [https://doi.org/10.1007/s00382-013-1683-2]

[9]:
```
skill = hindcast.verify(alignment='same_verifs')
```

```
INFO:root:lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

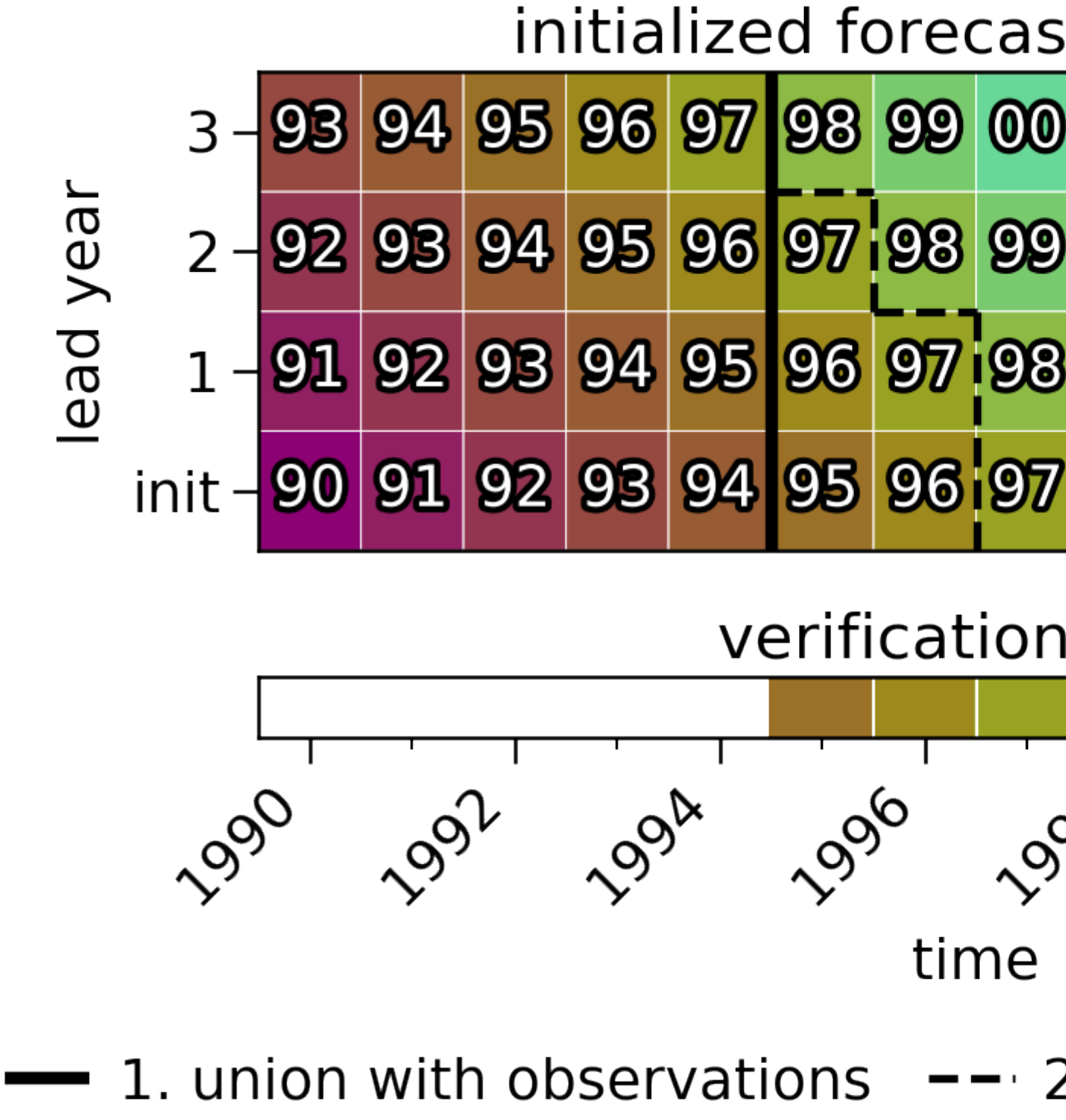
```
INFO:root:lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

Here, we include a figure of a simpler alignment case with annual initializations from 1990 through 2000 and three lead years. We verify this hypothetical initialized ensemble against a product that spans 1995 through 2002.

Two conditions must be met when selecting the verification window:

1. There must be a union between the initialization dates and verification dates. This is represented by the black vertical lines in the top panel below, which leave out 1990-1994 initializations since there aren't observations before 1995. This logic exists so that any reference forecasts (e.g. a persistence forecast) use an identical set of initializations as the initialized forecast.
2. A given verification time must exist across all leads. This is to ensure that at each lead, the entire set of chosen verification dates can be verified against. This is represented by diagonals in the top panel below (and the dashed black lines). Without the first stipulation, this would set the verification window at 1995-2001.

This leaves us with a verification window of [1998, 1999, 2000, 2001] which can be verified against across all leads (and have a complimentary persistence forecast with the same set of initializations used at each lead).



2.7.2 Same Initializations

```
alignment='same_inits'
```

The `same_inits` alignment finds a set of initializations that can verify over all leads. It also requires that the verification data have an observation at each initialization being retained. This is so that the reference forecast, such as persistence, uses an identical set of initializations in deriving its forecast. Notice in the logger output that a common set of initializations spanning 1955-2005 are used, while the verification window slides one year at each lead.

[10]:
```
skill = hindcast.verify(alignment='same_inits')
```

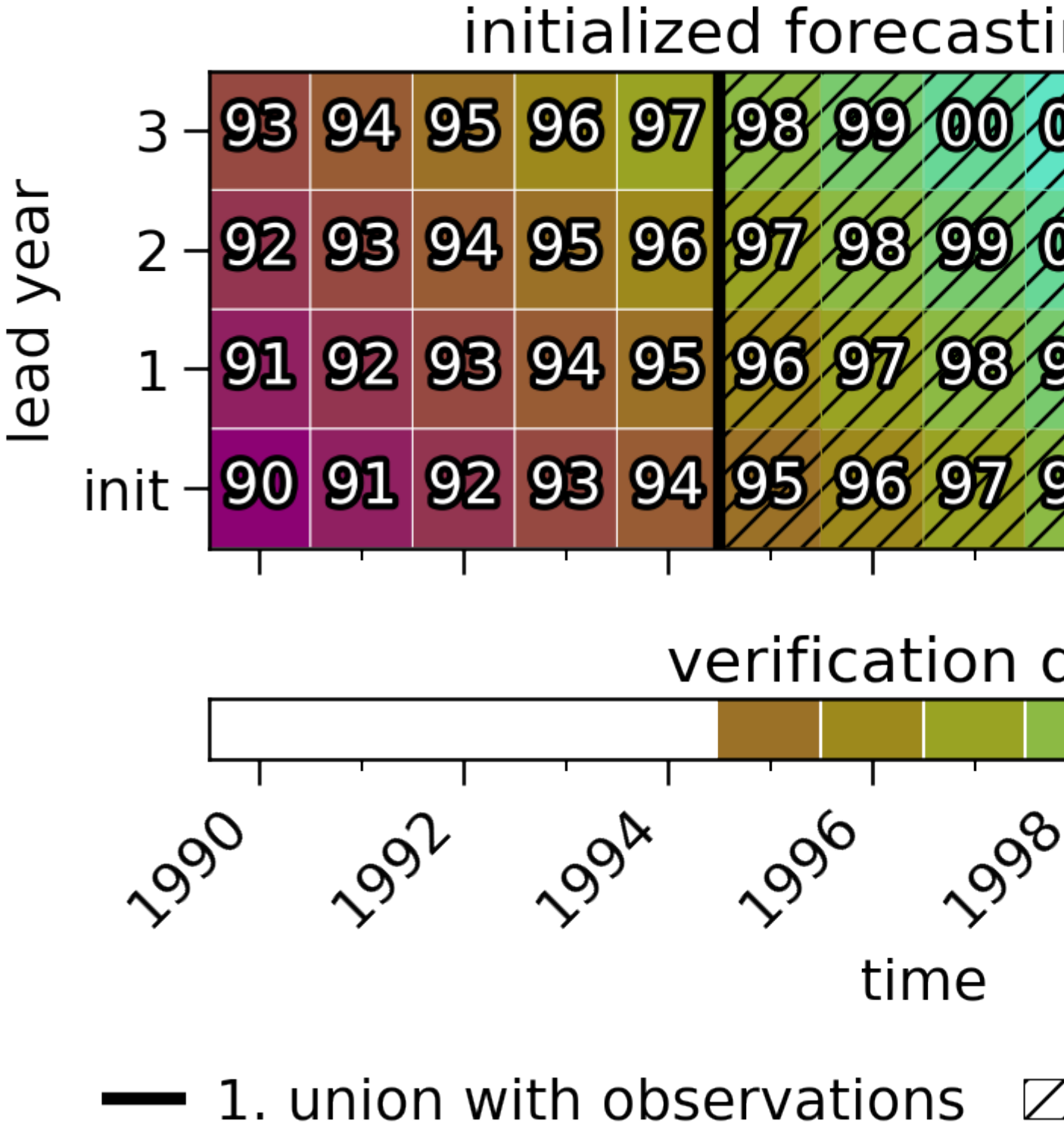
```
INFO:root:lead: 01 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1955-01-
↪01 00:00:00-2006-01-01 00:00:00
INFO:root:lead: 02 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1956-01-
↪01 00:00:00-2007-01-01 00:00:00
INFO:root:lead: 03 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1957-01-
↪01 00:00:00-2008-01-01 00:00:00
INFO:root:lead: 04 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1958-01-
↪01 00:00:00-2009-01-01 00:00:00
INFO:root:lead: 05 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1959-01-
↪01 00:00:00-2010-01-01 00:00:00
INFO:root:lead: 06 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1960-01-
↪01 00:00:00-2011-01-01 00:00:00
INFO:root:lead: 07 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1961-01-
↪01 00:00:00-2012-01-01 00:00:00
INFO:root:lead: 08 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1962-01-
↪01 00:00:00-2013-01-01 00:00:00
INFO:root:lead: 09 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1963-01-
↪01 00:00:00-2014-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

Here, we include a figure of a simpler alignment case with annual initializations from 1990 through 2000 and three lead years. We verify this hypothetical initialized ensemble against a product that spans 1995 through 2002.

Two conditions must be met to retain the initializations for verification:

1. There must be an observation in the verification data for the given initialization. In combination with (1), initializations 1990 through 1994 are left out. This logic exists so that any reference forecast (e.g. a persistence forecast) use an identical set of initializations as the initialized forecast.
2. All forecasted times (i.e., initialization + lead year) for a given initialization must be contained in the verification data. Schematically, this means that there must be a union between a column in the top panel and the time series in the bottom panel. The 2000 initialization below is left out since the verification data does not contain 2003.

This leaves us with initializations [1995, 1996, . . . , 1999] which can verify against the observations at all three lead years.


initialized forecasti
lead year
3
2
1
init
93 94 95 96 97 98 99 00
92 93 94 95 96 97 98 99
91 92 93 94 95 96 97 98
90 91 92 93 94 95 96 97
verification d
1990
1992
1994
1996
1998
time
1. union with observations

2.7.3 Maximize Degrees of Freedom

`alignment='maximize'`

The `maximize` alignment verifies against every available observation at each lead. This means that both the initializations and verification dates could be different at each lead. It also requires that the verification data have an observation at each initialization being retained. This is so that the reference forecast, such as persistence, uses an identical set of initializations in deriving its forecast.

Notice in the logger output that the initialization window shrinks from 1955-2014 (N=60) at lead year 1 to 1955-2005 (N=51) at lead year 10. Similarly, the verification window spans 1956-2015 at lead year 1 and 1965-2015 at lead year 10. However, using the other two alignment strategies (`same_verifs` and `same_inits`), there is a fixed N=51 to ensure constant initializations or verification dates, while the number of samples is extended to as high as 60 with this alignment strategy.

References:

1. Yeager, S. G., et al. "Predicting near-term changes in the Earth System: A large ensemble of initialized decadal prediction simulations using the Community Earth System Model." Bulletin of the American Meteorological Society 99.9 (2018): 1867-1886. [https://doi.org/10.1175/BAMS-D-17-0098.1]

[11]:
```
skill = hindcast.verify(alignment='maximize')
```

```
INFO:root:lead: 01 | inits: 1954-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1955-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 02 | inits: 1954-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1956-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 03 | inits: 1954-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1957-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 04 | inits: 1954-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1958-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 05 | inits: 1954-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1959-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 06 | inits: 1954-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1960-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 07 | inits: 1954-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1961-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 08 | inits: 1954-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1962-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 09 | inits: 1954-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1963-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

Here, we include a figure of a simpler alignment case with annual initializations from 1990 through 2000 and three lead years. We verify this hypothetical initialized ensemble against a product that spans 1995 through 2002.

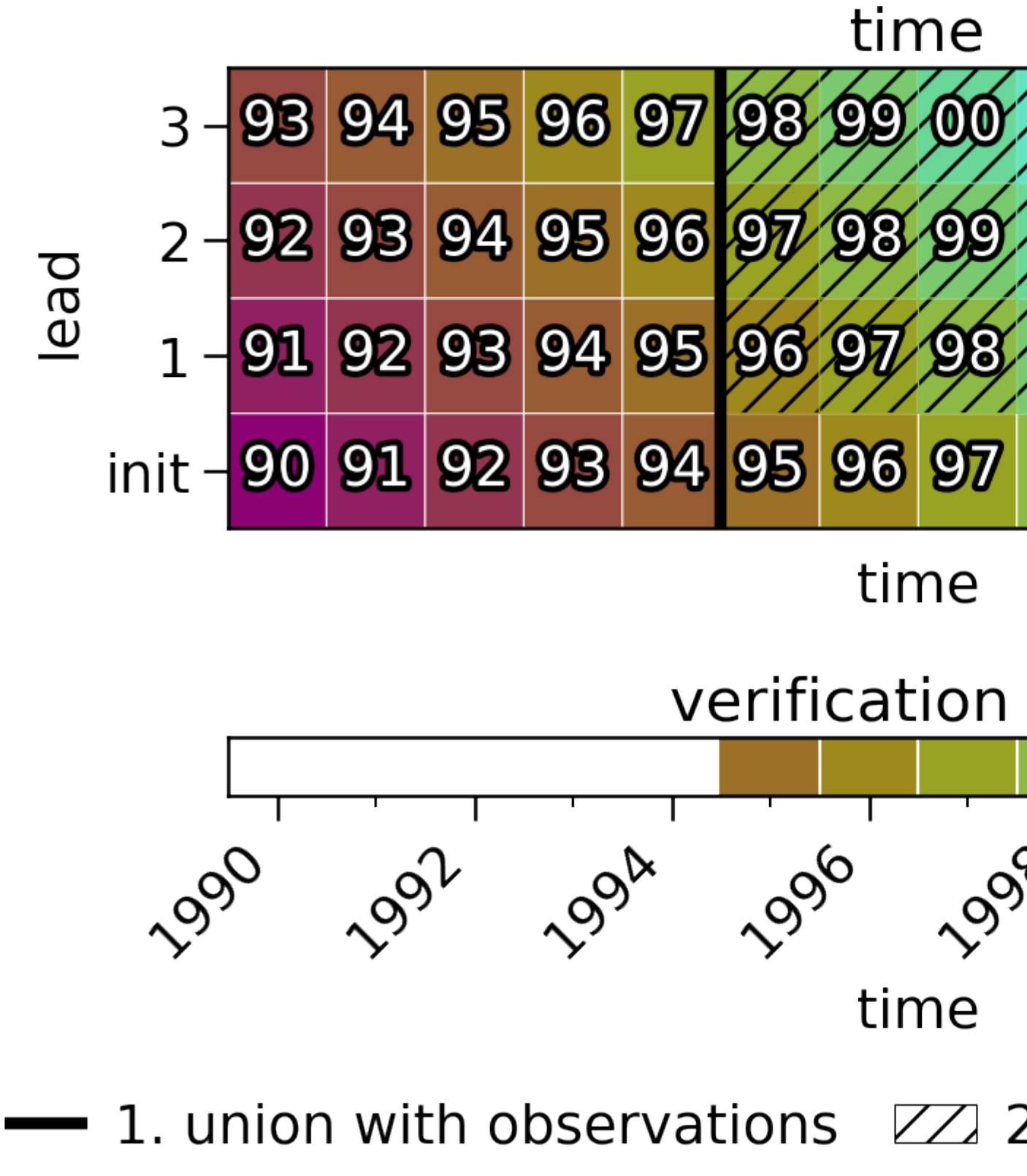
Two conditions must be met when selecting initializations/verifications at each lead:

1. There must be a union between the initialization dates and verification dates. This is represented by the black vertical lines in the top panel below, which leave out 1990-1994 initializations since there aren't observations before 1995. This logic exists so that any reference forecasts (e.g. a persistence forecast) use an identical set of initializations as the initialized forecast.
2. The selected initializations must verify with the provided observations for the given lead. This is shown by the hatching in the figure below. The 2000 initialization is left out at lead year 3 since there is no observation for 2003.

This leaves us with the following alignment:

- LY1 initializations: [1995, 1996, 1997, 1998, 1999, 2000]

- LY2 initializations: [1995, 1996, 1997, 1998, 1999, 2000]
- LY3 initializations: [1995, 1996, 1997, 1998, 1999]


time
lead
3
2
1
init
93 94 95 96 97 98 99 00
92 93 94 95 96 97 98 99
91 92 93 94 95 96 97 98
90 91 92 93 94 95 96 97
time
verification
1990
1992
1994
1996
time
1. union with observations

2.7.4 Logging

`climpred` uses the standard library `logging` to store the initializations and verification dates used at each lead for a given computation. This is used internally for testing, but more importantly, can be activated by the user so they can be sure of how computations are being done.

To see the log interactively, e.g. while working in Jupyter notebooks or on the command line use the following:

[12]:
```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

[13]:
```
skill = hindcast.verify(alignment='same_verifs')
```

```
INFO:root:lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

The `INFO` level reports the minimum and maximum bounds for initializations and verification dates. To see every single initialization and verification date used, set the level to `DEBUG`.

[14]:
```
logger.setLevel(logging.DEBUG)
```

[15]:
```
skill = hindcast.isel(lead=slice(0, 2)).verify(alignment='same_verifs')
```

```
INFO:root:lead: 01 | inits: 1955-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1956-01-
↪01 00:00:00-2015-01-01 00:00:00
DEBUG:root:
inits: ['1955-1-1', '1956-1-1', '1957-1-1', '1958-1-1', '1959-1-1', '1960-1-1', '1961-
↪1-1', '1962-1-1', '1963-1-1', '1964-1-1', '1965-1-1', '1966-1-1', '1967-1-1', '1968-
↪1-1', '1969-1-1', '1970-1-1', '1971-1-1', '1972-1-1', '1973-1-1', '1974-1-1', '1975-
↪1-1', '1976-1-1', '1977-1-1', '1978-1-1', '1979-1-1', '1980-1-1', '1981-1-1', '1982-
↪1-1', '1983-1-1', '1984-1-1', '1985-1-1', '1986-1-1', '1987-1-1', '1988-1-1', '1989-
↪1-1', '1990-1-1', '1991-1-1', '1992-1-1', '1993-1-1', '1994-1-1', '1995-1-1', '1996-
↪1-1', '1997-1-1', '1998-1-1', '1999-1-1', '2000-1-1', '2001-1-1', '2002-1-1', '2003-
↪1-1', '2004-1-1', '2005-1-1', '2006-1-1', '2007-1-1', '2008-1-1', '2009-1-1', '2010-
↪1-1', '2011-1-1', '2012-1-1', '2013-1-1', '2014-1-1']
verifs: ['1956-1-1', '1957-1-1', '1958-1-1', '1959-1-1', '1960-1-1', '1961-1-1',
↪'1962-1-1', '1963-1-1', '1964-1-1', '1965-1-1', '1966-1-1', '1967-1-1', '1968-1-1',
↪'1969-1-1', '1970-1-1', '1971-1-1', '1972-1-1', '1973-1-1', '1974-1-1', '1975-1-1',
↪'1976-1-1', '1977-1-1', '1978-1-1', '1979-1-1', '1980-1-1', '1981-1-1', '1982-1-1',
↪'1983-1-1', '1984-1-1', '1985-1-1', '1986-1-1', '1987-1-1', '1988-1-1', '1989-1-1',
↪'1990-1-1', '1991-1-1', '1992-1-1', '1993-1-1', '1994-1-1', '1995-1-1', '1996-1-1',
↪'1997-1-1', '1998-1-1', '1999-1-1', '2000-1-1', '2001-1-1', '2002-1-1', '2003-1-1',
↪'2004-1-1', '2005-1-1', '2006-1-1', '2007-1-1', '2008-1-1', '2009-1-1', '2010-1-1',
↪'2011-1-1', '2012-1-1', '2013-1-1', '2014-1-1', '2015-1-1']
```

(continues on next page)

(continued from previous page)

```
INFO:root:lead: 02 | inits: 1954-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1956-01-
↪01 00:00:00-2015-01-01 00:00:00
DEBUG:root:
inits: ['1954-1-1', '1955-1-1', '1956-1-1', '1957-1-1', '1958-1-1', '1959-1-1', '1960-
↪1-1', '1961-1-1', '1962-1-1', '1963-1-1', '1964-1-1', '1965-1-1', '1966-1-1', '1967-
↪1-1', '1968-1-1', '1969-1-1', '1970-1-1', '1971-1-1', '1972-1-1', '1973-1-1', '1974-
↪1-1', '1975-1-1', '1976-1-1', '1977-1-1', '1978-1-1', '1979-1-1', '1980-1-1', '1981-
↪1-1', '1982-1-1', '1983-1-1', '1984-1-1', '1985-1-1', '1986-1-1', '1987-1-1', '1988-
↪1-1', '1989-1-1', '1990-1-1', '1991-1-1', '1992-1-1', '1993-1-1', '1994-1-1', '1995-
↪1-1', '1996-1-1', '1997-1-1', '1998-1-1', '1999-1-1', '2000-1-1', '2001-1-1', '2002-
↪1-1', '2003-1-1', '2004-1-1', '2005-1-1', '2006-1-1', '2007-1-1', '2008-1-1', '2009-
↪1-1', '2010-1-1', '2011-1-1', '2012-1-1', '2013-1-1']
verifs: ['1956-1-1', '1957-1-1', '1958-1-1', '1959-1-1', '1960-1-1', '1961-1-1',
↪'1962-1-1', '1963-1-1', '1964-1-1', '1965-1-1', '1966-1-1', '1967-1-1', '1968-1-1',
↪'1969-1-1', '1970-1-1', '1971-1-1', '1972-1-1', '1973-1-1', '1974-1-1', '1975-1-1',
↪'1976-1-1', '1977-1-1', '1978-1-1', '1979-1-1', '1980-1-1', '1981-1-1', '1982-1-1',
↪'1983-1-1', '1984-1-1', '1985-1-1', '1986-1-1', '1987-1-1', '1988-1-1', '1989-1-1',
↪'1990-1-1', '1991-1-1', '1992-1-1', '1993-1-1', '1994-1-1', '1995-1-1', '1996-1-1',
↪'1997-1-1', '1998-1-1', '1999-1-1', '2000-1-1', '2001-1-1', '2002-1-1', '2003-1-1',
↪'2004-1-1', '2005-1-1', '2006-1-1', '2007-1-1', '2008-1-1', '2009-1-1', '2010-1-1',
↪'2011-1-1', '2012-1-1', '2013-1-1', '2014-1-1', '2015-1-1']
```

One can also save out the log to a file.

[16]:
```
logger = logging.getLogger()
logger.setLevel(logging.INFO)
fh = logging.FileHandler('hindcast.out')
logger.addHandler(fh)
```

[17]:
```
skill = hindcast.verify(alignment='same_verifs')
```

```
INFO:root:lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

[18]:
```
skill = hindcast.verify(alignment='same_verifs')
```

```
INFO:root:lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
INFO:root:lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
INFO:root:lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-
↪01 00:00:00-2015-01-01 00:00:00
```

[19]:
```
!cat hindcast.out
```

```
lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 01 | inits: 1963-01-01 00:00:00-2014-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 02 | inits: 1962-01-01 00:00:00-2013-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 03 | inits: 1961-01-01 00:00:00-2012-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 04 | inits: 1960-01-01 00:00:00-2011-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 05 | inits: 1959-01-01 00:00:00-2010-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 06 | inits: 1958-01-01 00:00:00-2009-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 07 | inits: 1957-01-01 00:00:00-2008-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 08 | inits: 1956-01-01 00:00:00-2007-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
lead: 09 | inits: 1955-01-01 00:00:00-2006-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
lead: 10 | inits: 1954-01-01 00:00:00-2005-01-01 00:00:00 | verifs: 1964-01-01 00:00:
↪00-2015-01-01 00:00:00
```

[20]:
```
!rm hindcast.out
```

2.8 Metrics

All high-level functions have an optional `metric` argument that can be called to determine which metric is used in computing predictability.

Note: We use the phrase 'observations' `o` here to refer to the 'truth' data to which we compare the forecast `f`. These metrics can also be applied relative to a control simulation, reconstruction, observations, etc. This would just change the resulting score from quantifying skill to quantifying potential predictability.

Internally, all metric functions require `forecast` and `observations` as inputs. The dimension `dim` is set internally by *`compute_hindcast()`* or *`compute_perfect_model()`* to specify over which dimensions the `metric` is applied. See *Comparisons* for more on the `dim` argument.

2.8.1 Deterministic

Deterministic metrics assess the forecast as a definite prediction of the future, rather than in terms of probabilities. Another way to look at deterministic metrics is that they are a special case of probabilistic metrics where a value of one is assigned to one category and zero to all others [Jolliffe2011].

Correlation Metrics

The below metrics rely fundamentally on correlations in their computation. In the literature, correlation metrics are typically referred to as the Anomaly Correlation Coefficient (ACC). This implies that anomalies in the forecast and observations are being correlated. Typically, this is computed using the linear *Pearson Product-Moment Correlation*. However, `climpred` also offers the *Spearman's Rank Correlation*.

Note that the p value associated with these correlations is computed via a separate metric. Use `pearson_r_p_value` or `spearman_r_p_value` to compute p values assuming that all samples in the correlated time series are independent. Use `pearson_r_eff_p_value` or `spearman_r_eff_p_value` to account for autocorrelation in the time series by calculating the `effective_sample_size`.

Pearson Product-Moment Correlation Coefficient

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [1]: print(f"\n\nKeywords: {metric_aliases['pearson_r']}")


Keywords: ['pearson_r', 'pr', 'acc', 'pacc']
```

`climpred.metrics.`**`_pearson_r`**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Pearson product-moment correlation coefficient.

A measure of the linear association between the forecast and verification data that is independent of the mean and variance of the individual distributions. This is also known as the Anomaly Correlation Coefficient (ACC) when correlating anomalies.

$$corr = \frac{cov(f, o)}{\sigma_f \cdot \sigma_o},$$

where σ_f and σ_o represent the standard deviation of the forecast and verification data over the experimental period, respectively.

Note: Use metric `pearson_r_p_value` or `pearson_r_eff_p_value` to get the corresponding p value.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	-1.0
maximum	1.0
perfect	1.0
orientation	positive

See also:

- xskillscore.pearson_r
- xskillscore.pearson_r_p_value
- climpred.pearson_r_p_value
- climpred.pearson_r_eff_p_value

Pearson Correlation p value

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [2]: print(f"\n\nKeywords: {metric_aliases['pearson_r_p_value']}")


Keywords: ['pearson_r_p_value', 'p_pval', 'pvalue', 'pval']
```

`climpred.metrics.`**`_pearson_r_p_value`**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Probability that forecast and verification data are linearly uncorrelated.

Two-tailed p value associated with the Pearson product-moment correlation coefficient (`pearson_r`), assuming that all samples are independent. Use `pearson_r_eff_p_value` to account for autocorrelation in the forecast and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to None.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to False.

Details:

minimum	0.0
maximum	1.0
perfect	1.0
orientation	negative

See also:

- xskillscore.pearson_r
- xskillscore.pearson_r_p_value
- climpred.pearson_r
- climpred.pearson_r_eff_p_value

Effective Sample Size

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [3]: print(f"\n\nKeywords: {metric_aliases['effective_sample_size']}")


Keywords: ['effective_sample_size', 'n_eff', 'eff_n']
```

climpred.metrics.**_effective_sample_size**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Effective sample size for temporally correlated data.

Note: Weights are not included here due to the dependence on temporal autocorrelation.

Note: This metric can only be used for hindcast-type simulations.

The effective sample size extracts the number of independent samples between two time series being correlated. This is derived by assessing the magnitude of the lag-1 autocorrelation coefficient in each of the time series being correlated. A higher autocorrelation induces a lower effective sample size which raises the correlation coefficient for a given p value.

The effective sample size is used in computing the effective p value. See `pearson_r_eff_p_value` and `spearman_r_eff_p_value`.

$$N_{eff} = N \left(\frac{1 - \rho_f \rho_o}{1 + \rho_f \rho_o} \right),$$

where ρ_f and ρ_o are the lag-1 autocorrelation coefficients for the forecast and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	N/A
orientation	positive

Reference:

- Bretherton, Christopher S., et al. "The effective number of spatial degrees of freedom of a time-varying field." Journal of climate 12.7 (1999): 1990-2009.

Pearson Correlation Effective p value

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [4]: print(f"\n\nKeywords: {metric_aliases['pearson_r_eff_p_value']}")


Keywords: ['pearson_r_eff_p_value', 'p_pval_eff', 'pvalue_eff', 'pval_eff']
```

climpred.metrics.**_pearson_r_eff_p_value**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Probability that forecast and verification data are linearly uncorrelated, accounting for autocorrelation.

Note: Weights are not included here due to the dependence on temporal autocorrelation.

Note: This metric can only be used for hindcast-type simulations.

The effective p value is computed by replacing the sample size N in the t-statistic with the effective sample size, N_{eff}. The same Pearson product-moment correlation coefficient r is used as when computing the standard p value.

$$t = r \sqrt{\frac{N_{eff} - 2}{1 - r^2}},$$

where N_{eff} is computed via the autocorrelation in the forecast and verification data.

$$N_{eff} = N\left(\frac{1-\rho_f\rho_o}{1+\rho_f\rho_o}\right),$$

where ρ_f and ρ_o are the lag-1 autocorrelation coefficients for the forecast and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to False.

Details:

minimum	0.0
maximum	1.0
perfect	1.0
orientation	negative

See also:

- climpred.effective_sample_size
- climpred.spearman_r_eff_p_value

Reference:

- Bretherton, Christopher S., et al. "The effective number of spatial degrees of freedom of a time-varying field." Journal of climate 12.7 (1999): 1990-2009.

Spearman's Rank Correlation Coefficient

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [5]: print(f"\n\nKeywords: {metric_aliases['spearman_r']}")


Keywords: ['spearman_r', 'sacc', 'sr']
```

climpred.metrics.**_spearman_r**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Spearman's rank correlation coefficient.

$$corr = \text{pearsonr}(ranked(f), ranked(o))$$

This correlation coefficient is nonparametric and assesses how well the relationship between the forecast and verification data can be described using a monotonic function. It is computed by first ranking the forecasts and verification data, and then correlating those ranks using the pearson_r correlation.

This is also known as the anomaly correlation coefficient (ACC) when comparing anomalies, although the Pearson product-moment correlation coefficient (pearson_r) is typically used when computing the ACC.

Note: Use metric `spearman_r_p_value` or `spearman_r_eff_p_value` to get the corresponding p value.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	-1.0
maximum	1.0
perfect	1.0
orientation	positive

See also:

- xskillscore.spearman_r
- xskillscore.spearman_r_p_value
- climpred.spearman_r_p_value
- climpred.spearman_r_eff_p_value

Spearman's Rank Correlation Coefficient p value

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [6]: print(f"\n\nKeywords: {metric_aliases['spearman_r_p_value']}")


Keywords: ['spearman_r_p_value', 's_pval', 'spvalue', 'spval']
```

`climpred.metrics.`**`_spearman_r_p_value`**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Probability that forecast and verification data are monotonically uncorrelated.

Two-tailed p value associated with the Spearman's rank correlation coefficient (`spearman_r`), assuming that all samples are independent. Use `spearman_r_eff_p_value` to account for autocorrelation in the forecast and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.

- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to None.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to False.

Details:

minimum	0.0
maximum	1.0
perfect	1.0
orientation	negative

See also:

- xskillscore.spearman_r
- xskillscore.spearman_r_p_value
- climpred.spearman_r
- climpred.spearman_r_eff_p_value

Spearman's Rank Correlation Effective p value

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [7]: print(f"\n\nKeywords: {metric_aliases['spearman_r_eff_p_value']}")


Keywords: ['spearman_r_eff_p_value', 's_pval_eff', 'spvalue_eff', 'spval_eff']
```

climpred.metrics.**_spearman_r_eff_p_value**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Probability that forecast and verification data are monotonically uncorrelated, accounting for autocorrelation.

Note: Weights are not included here due to the dependence on temporal autocorrelation.

Note: This metric can only be used for hindcast-type simulations.

The effective p value is computed by replacing the sample size N in the t-statistic with the effective sample size, N_{eff}. The same Spearman's rank correlation coefficient r is used as when computing the standard p value.

$$t = r\sqrt{\frac{N_{eff} - 2}{1 - r^2}},$$

where N_{eff} is computed via the autocorrelation in the forecast and verification data.

$$N_{eff} = N\left(\frac{1 - \rho_f \rho_o}{1 + \rho_f \rho_o}\right),$$

where ρ_f and ρ_o are the lag-1 autocorrelation coefficients for the forecast and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.

- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	1.0
perfect	1.0
orientation	negative

See also:

- climpred.effective_sample_size
- climpred.pearson_r_eff_p_value

Reference:

- Bretherton, Christopher S., et al. "The effective number of spatial degrees of freedom of a time-varying field." Journal of climate 12.7 (1999): 1990-2009.

Distance Metrics

This class of metrics simply measures the distance (or difference) between forecasted values and observed values.

Mean Squared Error (MSE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [8]: print(f"\n\nKeywords: {metric_aliases['mse']}")


Keywords: ['mse']
```

`climpred.metrics.`**_mse** (*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Mean Sqaure Error (MSE).

$$MSE = \overline{(f - o)^2}$$

The average of the squared difference between forecasts and verification data. This incorporates both the variance and bias of the estimator. Because the error is squared, it is more sensitive to large forecast errors than `mae`, and thus a more conservative metric. For example, a single error of 2°C counts the same as two 1°C errors when using `mae`. On the other hand, the 2°C error counts double for `mse`. See Jolliffe and Stephenson, 2011.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.

- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

See also:

- xskillscore.mse

Reference:

- Ian T. Jolliffe and David B. Stephenson. Forecast Verification: A Practitioner's Guide in Atmospheric Science. John Wiley & Sons, Ltd, Chichester, UK, December 2011. ISBN 978-1-119-96000-3 978-0-470-66071-3. URL: http://doi.wiley.com/10.1002/9781119960003.

Root Mean Square Error (RMSE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [9]: print(f"\n\nKeywords: {metric_aliases['rmse']}")


Keywords: ['rmse']
```

climpred.metrics.**_rmse**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Root Mean Sqaure Error (RMSE).

$$RMSE = \sqrt{(f-o)^2}$$

The square root of the average of the squared differences between forecasts and verification data.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

See also:

- xskillscore.rmse

Mean Absolute Error (MAE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [10]: print(f"\n\nKeywords: {metric_aliases['mae']}")


Keywords: ['mae']
```

climpred.metrics.**_mae**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Mean Absolute Error (MAE).

$$MAE = \overline{|f - o|}$$

The average of the absolute differences between forecasts and verification data. A more robust measure of forecast accuracy than `mse` which is sensitive to large outlier forecast errors.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

See also:

- xskillscore.mae

Reference:

- Ian T. Jolliffe and David B. Stephenson. Forecast Verification: A Practitioner's Guide in Atmospheric Science. John Wiley & Sons, Ltd, Chichester, UK, December 2011. ISBN 978-1-119-96000-3 978-0-470-66071-3. URL: http://doi.wiley.com/10.1002/9781119960003.

Median Absolute Error

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [11]: print(f"\n\nKeywords: {metric_aliases['median_absolute_error']}")


Keywords: ['median_absolute_error']
```

climpred.metrics.**_median_absolute_error**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Median Absolute Error.

$$median(|f - o|)$$

The median of the absolute differences between forecasts and verification data. Applying the median function to absolute error makes it more robust to outliers.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

See also:

- xskillscore.median_absolute_error

Normalized Distance Metrics

Distance metrics like `mse` can be normalized to 1. The normalization factor depends on the comparison type choosen. For example, the distance between an ensemble member and the ensemble mean is half the distance of an ensemble member with other ensemble members. See *_get_norm_factor()*.

Normalized Mean Square Error (NMSE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [12]: print(f"\n\nKeywords: {metric_aliases['nmse']}")


Keywords: ['nmse', 'nev']
```

climpred.metrics.**_nmse**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Normalized MSE (NMSE), also known as Normalized Ensemble Variance (NEV).

Mean Square Error (`mse`) normalized by the variance of the verification data.

$$NMSE = NEV = \frac{MSE}{\sigma_o^2 \cdot fac} = \frac{\overline{(f-o)^2}}{\sigma_o^2 \cdot fac},$$

where fac is 1 when using comparisons involving the ensemble mean (`m2e`, `e2c`, `e2o`) and 2 when using comparisons involving individual ensemble members (`m2c`, `m2m`, `m2o`). See *`_get_norm_factor()`*.

Note: `climpred` uses a single-valued internal reference forecast for the NMSE, in the terminology of Murphy 1988. I.e., we use a single climatological variance of the verification data *within* the experimental window for normalizing MSE.

Parameters

- **`forecast`** (*`xarray object`*) – Forecast.
- **`verif`** (*`xarray object`*) – Verification data.
- **`dim`** (*`str`*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **`weights`** (*`xarray object, optional`*) – Weights to apply over dimension. Defaults to `None`.
- **`skipna`** (*`bool, optional`*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.
- **`comparison`** (*`str`*) – Name comparison needed for normalization factor *fac*, see *`_get_norm_factor()`* (Handled internally by the compute functions)

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative
better than climatology	0.0 - 1.0
worse than climatology	> 1.0

Reference:

- Griffies, S. M., and K. Bryan. "A Predictability Study of Simulated North Atlantic Multidecadal Variability." Climate Dynamics 13, no. 7–8 (August 1, 1997): 459–87. https://doi.org/10/ch4kc4.
- Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.

Normalized Mean Absolute Error (NMAE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [13]: print(f"\n\nKeywords: {metric_aliases['nmae']}")


Keywords: ['nmae']
```

climpred.metrics.**_nmae**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Normalized Mean Absolute Error (NMAE).

Mean Absolute Error (mae) normalized by the standard deviation of the verification data.

$$NMAE = \frac{MAE}{\sigma_o \cdot fac} = \frac{\overline{|f - o|}}{\sigma_o \cdot fac},$$

where fac is 1 when using comparisons involving the ensemble mean (m2e, e2c, e2o) and 2 when using comparisons involving individual ensemble members (m2c, m2m, m2o). See *_get_norm_factor()*.

Note: climpred uses a single-valued internal reference forecast for the NMAE, in the terminology of Murphy 1988. I.e., we use a single climatological standard deviation of the verification data *within* the experimental window for normalizing MAE.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to None.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to False.
- **comparison** (*str*) – Name comparison needed for normalization factor *fac*, see *_get_norm_factor()* (Handled internally by the compute functions)

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative
better than climatology	0.0 - 1.0
worse than climatology	> 1.0

Reference:

- Griffies, S. M., and K. Bryan. "A Predictability Study of Simulated North Atlantic Multidecadal Variability." Climate Dynamics 13, no. 7–8 (August 1, 1997): 459–87. https://doi.org/10/ch4kc4.
- Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.

Normalized Root Mean Square Error (NRMSE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [14]: print(f"\n\nKeywords: {metric_aliases['nrmse']}")
```

(continues on next page)

(continued from previous page)

```
Keywords: ['nrmse']
```

climpred.metrics.**_nrmse**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Normalized Root Mean Square Error (NRMSE).

Root Mean Square Error (`rmse`) normalized by the standard deviation of the verification data.

$$NRMSE = \frac{RMSE}{\sigma_o \cdot \sqrt{fac}} = \sqrt{\frac{MSE}{\sigma_o^2 \cdot fac}} = \sqrt{\frac{\overline{(f-o)^2}}{\sigma_o^2 \cdot fac}},$$

where fac is 1 when using comparisons involving the ensemble mean (`m2e`, `e2c`, `e2o`) and 2 when using comparisons involving individual ensemble members (`m2c`, `m2m`, `m2o`). See *_get_norm_factor()*.

Note: `climpred` uses a single-valued internal reference forecast for the NRMSE, in the terminology of Murphy 1988. I.e., we use a single climatological variance of the verification data *within* the experimental window for normalizing RMSE.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.
- **comparison** (*str*) – Name comparison needed for normalization factor *fac*, see *_get_norm_factor()* (Handled internally by the compute functions)

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative
better than climatology	0.0 - 1.0
worse than climatology	> 1.0

Reference:

- Bushuk, Mitchell, Rym Msadek, Michael Winton, Gabriel Vecchi, Xiaosong Yang, Anthony Rosati, and Rich Gudgel. "Regional Arctic Sea–Ice Prediction: Potential versus Operational Seasonal Forecast Skill." Climate Dynamics, June 9, 2018. https://doi.org/10/gd7hfq.
- Hawkins, Ed, Steffen Tietsche, Jonathan J. Day, Nathanael Melia, Keith Haines, and Sarah Keeley. "Aspects of Designing and Evaluating Seasonal-to-Interannual Arctic Sea-Ice Prediction Systems." Quarterly Journal of the Royal Meteorological Society 142, no. 695 (January 1, 2016): 672–83. https://doi.org/10/gfb3pn.

- Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.

Mean Square Error Skill Score (MSESS)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [15]: print(f"\n\nKeywords: {metric_aliases['msess']}")


Keywords: ['msess', 'ppp', 'msss']
```

climpred.metrics.**_msess**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Mean Squared Error Skill Score (MSESS).

$$MSESS = 1 - \frac{MSE}{\sigma^2_{ref} \cdot fac} = 1 - \frac{\overline{(f-o)^2}}{\sigma^2_{ref} \cdot fac},$$

where fac is 1 when using comparisons involving the ensemble mean (`m2e`, `e2c`, `e2o`) and 2 when using comparisons involving individual ensemble members (`m2c`, `m2m`, `m2o`). See *_get_norm_factor()*.

This skill score can be intepreted as a percentage improvement in accuracy. I.e., it can be multiplied by 100%.

Note: `climpred` uses a single-valued internal reference forecast for the MSSS, in the terminology of Murphy 1988. I.e., we use a single climatological variance of the verification data *within* the experimental window for normalizing MSE.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.
- **comparison** (*str*) – Name comparison needed for normalization factor *fac*, see *_get_norm_factor()* (Handled internally by the compute functions)

Details:

minimum	-∞
maximum	1.0
perfect	1.0
orientation	positive
better than climatology	> 0.0
equal to climatology	0.0
worse than climatology	< 0.0

Reference:

- Griffies, S. M., and K. Bryan. "A Predictability Study of Simulated North Atlantic Multidecadal Variability." Climate Dynamics 13, no. 7–8 (August 1, 1997): 459–87. https://doi.org/10/ch4kc4.
- Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.
- Pohlmann, Holger, Michael Botzet, Mojib Latif, Andreas Roesch, Martin Wild, and Peter Tschuck. "Estimating the Decadal Predictability of a Coupled AOGCM." Journal of Climate 17, no. 22 (November 1, 2004): 4463–72. https://doi.org/10/d2qf62.
- Bushuk, Mitchell, Rym Msadek, Michael Winton, Gabriel Vecchi, Xiaosong Yang, Anthony Rosati, and Rich Gudgel. "Regional Arctic Sea–Ice Prediction: Potential versus Operational Seasonal Forecast Skill. Climate Dynamics, June 9, 2018. https://doi.org/10/gd7hfq.

Mean Absolute Percentage Error (MAPE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [16]: print(f"\n\nKeywords: {metric_aliases['mape']}")


Keywords: ['mape']
```

climpred.metrics.**_mape**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Mean Absolute Percentage Error (MAPE).

Mean absolute error (`mae`) expressed as the fractional error relative to the verification data.

$$MAPE = \frac{1}{n}\sum\frac{|f-o|}{|o|}$$

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

See also:

- xskillscore.mape

Symmetric Mean Absolute Percentage Error (sMAPE)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [17]: print(f"\n\nKeywords: {metric_aliases['smape']}")


Keywords: ['smape']
```

climpred.metrics.**_smape**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Symmetric Mean Absolute Percentage Error (sMAPE).

Similar to the Mean Absolute Percentage Error (`mape`), but sums the forecast and observation mean in the denominator.

$$sMAPE = \frac{1}{n}\sum\frac{|f-o|}{|f|+|o|}$$

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	0.0
maximum	1.0
perfect	0.0
orientation	negative

See also:

- xskillscore.smape

Unbiased Anomaly Correlation Coefficient (uACC)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [18]: print(f"\n\nKeywords: {metric_aliases['uacc']}")


Keywords: ['uacc']
```

climpred.metrics.**_uacc**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Bushuk's unbiased Anomaly Correlation Coefficient (uACC).

This is typically used in perfect model studies. Because the perfect model Anomaly Correlation Coefficient (ACC) is strongly state dependent, a standard ACC (e.g. one computed using `pearson_r`) will be highly sensitive to the set of start dates chosen for the perfect model study. The Mean Square Skill Score (`MSSS`)

can be related directly to the ACC as `MSSS = ACC^(2)` (see Murphy 1988 and Bushuk et al. 2019), so the unbiased ACC can be derived as `uACC = sqrt(MSSS)`.

$$uACC = \sqrt{MSSS} = \sqrt{1 - \frac{\overline{(f-o)^2}}{\sigma^2_{ref} \cdot fac}},$$

where fac is 1 when using comparisons involving the ensemble mean (`m2e`, `e2c`, `e2o`) and 2 when using comparisons involving individual ensemble members (`m2c`, `m2m`, `m2o`). See *`_get_norm_factor()`*.

Note: Because of the square root involved, any negative `MSSS` values are automatically converted to NaNs.

Parameters

- **`forecast`** (*`xarray object`*) – Forecast.
- **`verif`** (*`xarray object`*) – Verification data.
- **`dim`** (*`str`*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **`weights`** (*`xarray object, optional`*) – Weights to apply over dimension. Defaults to `None`.
- **`skipna`** (*`bool, optional`*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.
- **`comparison`** (*`str`*) – Name comparison needed for normalization factor `fac`, see *`_get_norm_factor()`* (Handled internally by the compute functions)

Details:

minimum	0.0
maximum	1.0
perfect	1.0
orientation	positive
better than climatology	> 0.0
equal to climatology	0.0

Reference:

- Bushuk, Mitchell, Rym Msadek, Michael Winton, Gabriel Vecchi, Xiaosong Yang, Anthony Rosati, and Rich Gudgel. "Regional Arctic Sea–Ice Prediction: Potential versus Operational Seasonal Forecast Skill." Climate Dynamics, June 9, 2018. https://doi.org/10/gd7hfq.
- Allan H. Murphy. Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient. Monthly Weather Review, 116(12):2417–2424, December 1988. https://doi.org/10/fc7mxd.

Murphy Decomposition Metrics

Metrics derived in [Murphy1988] which decompose the `MSESS` into a correlation term, a conditional bias term, and an unconditional bias term. See https://www-miklip.dkrz.de/about/murcss/ for a walk through of the decomposition.

Standard Ratio

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [19]: print(f"\n\nKeywords: {metric_aliases['std_ratio']}")


Keywords: ['std_ratio']
```

climpred.metrics.**_std_ratio**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Ratio of standard deviations of the forecast over the verification data.

$$\text{std ratio} = \frac{\sigma_f}{\sigma_o},$$

where σ_f and σ_o are the standard deviations of the forecast and the verification data over the experimental period, respectively.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute functions.

Details:

minimum	0.0
maximum	∞
perfect	1.0
orientation	N/A

Reference:

- https://www-miklip.dkrz.de/about/murcss/

Conditional Bias

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [20]: print(f"\n\nKeywords: {metric_aliases['conditional_bias']}")


Keywords: ['conditional_bias', 'c_b', 'cond_bias']
```

climpred.metrics.**_conditional_bias**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Conditional bias between forecast and verification data.

$$\text{conditional bias} = r_{fo} - \frac{\sigma_f}{\sigma_o},$$

where σ_f and σ_o are the standard deviations of the forecast and verification data over the experimental period, respectively.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.

- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute functions.

Details:

minimum	-∞
maximum	1.0
perfect	0.0
orientation	negative

Reference:

- https://www-miklip.dkrz.de/about/murcss/

Unconditional Bias

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [21]: print(f"\n\nKeywords: {metric_aliases['unconditional_bias']}")


Keywords: ['unconditional_bias', 'u_b', 'bias']
```

Simple bias of the forecast minus the observations.

climpred.metrics.**_unconditional_bias**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Unconditional bias.

$$bias = f - o$$

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute functions.

Details:

minimum	-∞
maximum	∞
perfect	0.0
orientation	negative

Reference:

- https://www.cawcr.gov.au/projects/verification/
- https://www-miklip.dkrz.de/about/murcss/

Bias Slope

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [22]: print(f"\n\nKeywords: {metric_aliases['bias_slope']}")


Keywords: ['bias_slope']
```

climpred.metrics.**_bias_slope**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Bias slope between verification data and forecast standard deviations.

$$\text{bias slope} = \frac{s_o}{s_f} \cdot r_{fo},$$

where r_{fo} is the Pearson product-moment correlation between the forecast and the verification data and s_o and s_f are the standard deviations of the verification data and forecast over the experimental period, respectively.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.
- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute functions.

Details:

minimum	0.0
maximum	∞
perfect	1.0
orientation	negative

Reference:

- https://www-miklip.dkrz.de/about/murcss/

Murphy's Mean Square Error Skill Score

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [23]: print(f"\n\nKeywords: {metric_aliases['msess_murphy']}")


Keywords: ['msess_murphy', 'msss_murphy']
```

climpred.metrics.**_msess_murphy**(*forecast*, *verif*, *dim=None*, ***metric_kwargs*)

Murphy's Mean Square Error Skill Score (MSESS).

$$MSESS_{Murphy} = r_{fo}^2 - [\text{conditional bias}]^2 - [\frac{\text{(unconditional) bias}}{\sigma_o}]^2,$$

where r_{fo}^2 represents the Pearson product-moment correlation coefficient between the forecast and verification data and σ_o represents the standard deviation of the verification data over the experimental period. See `conditional_bias` and `unconditional_bias` for their respective formulations.

Parameters

- **forecast** (*xarray object*) – Forecast.
- **verif** (*xarray object*) – Verification data.

- **dim** (*str*) – Dimension(s) to perform metric over. Automatically set by compute function.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to None.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to False.

Details:

minimum	-∞
maximum	1.0
perfect	1.0
orientation	positive

See also:

- climpred.pearson_r
- climpred.conditional_bias
- climpred.unconditional_bias

Reference:

- https://www-miklip.dkrz.de/about/murcss/
- Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.

2.8.2 Probabilistic

Probabilistic metrics include the spread of the ensemble simulations in their calculations and assign a probability value between 0 and 1 to their forecasts [Jolliffe2011].

Continuous Ranked Probability Score (CRPS)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [24]: print(f"\n\nKeywords: {metric_aliases['crps']}")


Keywords: ['crps']
```

climpred.metrics.**_crps**(*forecast*, *verif*, ***metric_kwargs*)

Continuous Ranked Probability Score (CRPS).

The CRPS can also be considered as the probabilistic Mean Absolute Error (mae). It compares the empirical distribution of an ensemble forecast to a scalar observation. Smaller scores indicate better skill.

$$CRPS = \int_{-\infty}^{\infty} (F(f) - H(f - o))^2 df,$$

where $F(f)$ is the cumulative distribution function (CDF) of the forecast (since the verification data are not assigned a probability), and H() is the Heaviside step function where the value is 1 if the argument is positive

(i.e., the forecast overestimates verification data) or zero (i.e., the forecast equals verification data) and is 0 otherwise (i.e., the forecast is less than verification data).

Note: The CRPS is expressed in the same unit as the observed variable. It generalizes the Mean Absolute Error (MAE), and reduces to the MAE if the forecast is determinstic.

Parameters

- **forecast** (*xr.object*) – Forecast with *member* dim.
- **verif** (*xr.object*) – Verification data without *member* dim.
- **metric_kwargs** (*xr.object*) – If provided, the CRPS is calculated exactly with the assigned probability weights to each forecast. Weights should be positive, but do not need to be normalized. By default, each forecast is weighted equally.

Details:

minimum	0.0
maximum	∞
perfect	0.0
orientation	negative

Reference:

- Matheson, James E., and Robert L. Winkler. "Scoring Rules for Continuous Probability Distributions." Management Science 22, no. 10 (June 1, 1976): 1087–96. https://doi.org/10/cwwt4g.
- https://www.lokad.com/continuous-ranked-probability-score

See also:

- properscoring.crps_ensemble
- xskillscore.crps_ensemble

Continuous Ranked Probability Skill Score (CRPSS)

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [25]: print(f"\n\nKeywords: {metric_aliases['crpss']}")


Keywords: ['crpss']
```

climpred.metrics.**_crpss**(*forecast*, *verif*, ***metric_kwargs*)

Continuous Ranked Probability Skill Score.

This can be used to assess whether the ensemble spread is a useful measure for the forecast uncertainty by comparing the CRPS of the ensemble forecast to that of a reference forecast with the desired spread.

$$CRPSS = 1 - \frac{CRPS_{initialized}}{CRPS_{clim}}$$

Note: When assuming a Gaussian distribution of forecasts, use default `gaussian=True`. If not gaussian, you may specify the distribution type, xmin/xmax/tolerance for integration (see xskillscore.crps_quadrature).

Parameters

- **forecast** (*xr.object*) – Forecast with `member` dim.
- **verif** (*xr.object*) – Verification data without `member` dim.
- **gaussian** (*bool, optional*) – If `True`, assum Gaussian distribution for baseline skill. Defaults to `True`.
- **cdf_or_dist** (*scipy.stats*) – Function which returns the cumulative density of the forecast at value x. This can also be an object with a callable `cdf()` method such as a `scipy.stats.distribution` object. Defaults to `scipy.stats.norm`.
- **xmin** (*float*) – Lower bounds for integration. Only use if not assuming Gaussian.
- **xmax** (*float*) –
- **tol** (*float, optional*) – The desired accuracy of the CRPS. Larger values will speed up integration. If `tol` is set to `None`, bounds errors or integration tolerance errors will be ignored. Only use if not assuming Gaussian.

Details:

minimum	-∞
maximum	1.0
perfect	1.0
orientation	positive
better than climatology	> 0.0
worse than climatology	< 0.0

Reference:

- Matheson, James E., and Robert L. Winkler. "Scoring Rules for Continuous Probability Distributions." Management Science 22, no. 10 (June 1, 1976): 1087–96. https://doi.org/10/cwwt4g.
- Gneiting, Tilmann, and Adrian E Raftery. "Strictly Proper Scoring Rules, Prediction, and Estimation." Journal of the American Statistical Association 102, no. 477 (March 1, 2007): 359–78. https://doi.org/10/c6758w.

Example

```
>>> compute_perfect_model(ds, control, metric='crpss')
>>> compute_perfect_model(ds, control, metric='crpss', gaussian=False,
                          cdf_or_dist=scipy.stats.norm, xminimum=-10,
                          xmaximum=10, tol=1e-6)
```

See also:

- properscoring.crps_ensemble
- xskillscore.crps_ensemble

Continuous Ranked Probability Skill Score Ensemble Spread

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [26]: print(f"\n\nKeywords: {metric_aliases['crpss_es']}")


Keywords: ['crpss_es']
```

climpred.metrics.**_crpss_es**(*forecast*, *verif*, ***metric_kwargs*)

Continuous Ranked Probability Skill Score Ensemble Spread.

If the ensemble variance is smaller than the observed `mse`, the ensemble is said to be under-dispersive (or overconfident). An ensemble with variance larger than the verification data indicates one that is over-dispersive (underconfident).

$$CRPSS = 1 - \frac{CRPS(\sigma_f^2)}{CRPS(\sigma_o^2)}$$

Parameters

- **forecast** (*xr.object*) – Forecast with `member` dim.
- **verif** (*xr.object*) – Verification data without `member` dim.
- **weights** (*xarray object, optional*) – Weights to apply over dimension. Defaults to `None`.
- **skipna** (*bool, optional*) – If True, skip NaNs over dimension being applied to. Defaults to `False`.

Details:

minimum	-∞
maximum	0.0
perfect	0.0
orientation	positive
under-dispersive	> 0.0
over-dispersive	< 0.0

Reference:

- Kadow, Christopher, Sebastian Illing, Oliver Kunst, Henning W. Rust, Holger Pohlmann, Wolfgang A. Müller, and Ulrich Cubasch. "Evaluation of Forecasts by Accuracy and Spread in the MiKlip Decadal Climate Prediction System." Meteorologische Zeitschrift, December 21, 2016, 631–43. https://doi.org/10/f9jrhw.

Range:

- perfect: 0
- else: negative

Brier Score

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [27]: print(f"\n\nKeywords: {metric_aliases['brier_score']}")


Keywords: ['brier_score', 'brier', 'bs']
```

climpred.metrics.**_brier_score**(*forecast*, *verif*, ***metric_kwargs*)

Brier Score.

The Mean Square Error (`mse`) of probabilistic two-category forecasts where the verification data are either 0 (no occurrence) or 1 (occurrence) and forecast probability may be arbitrarily distributed between occurrence and non-occurrence. The Brier Score equals zero for perfect (single-valued) forecasts and one for forecasts that are always incorrect.

$$BS(f, o) = (f_1 - o)^2,$$

where f_1 is the forecast probability of $o = 1$.

Note: The Brier Score requires that the observation is binary, i.e., can be described as one (a "hit") or zero (a "miss").

Parameters

- **forecast** (*xr.object*) – Forecast with `member` dim.
- **verif** (*xr.object*) – Verification data without `member` dim.
- **func** (*function*) – Function to be applied to verification data and forecasts and then `mean('member')` to get forecasts and verification data in interval [0,1].

Details:

minimum	0.0
maximum	1.0
perfect	0.0
orientation	negative

Reference:

- Brier, Glenn W. Verification of forecasts expressed in terms of probability." Monthly Weather Review 78, no. 1 (1950). https://doi.org/10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2.
- https://www.nws.noaa.gov/oh/rfcdev/docs/ Glossary_Forecast_Verification_Metrics.pdf

See also:

- properscoring.brier_score
- xskillscore.brier_score

Example

```
>>> def pos(x): return x > 0
>>> compute_perfect_model(ds, control, metric='brier_score', func=pos)
```

Threshold Brier Score

```
# Enter any of the below keywords in ``metric=...`` for the compute functions.
In [28]: print(f"\n\nKeywords: {metric_aliases['threshold_brier_score']}")


Keywords: ['threshold_brier_score', 'tbs']
```

climpred.metrics.**_threshold_brier_score**(*forecast*, *verif*, ***metric_kwargs*)

Brier score of an ensemble for exceeding given thresholds.

$$CRPS = \int_f BS(F(f), H(f-o))df,$$

where $F(o) = \int_{f \leq o} p(f)df$ is the cumulative distribution function (CDF) of the forecast distribution F, o is a point estimate of the true observation (observational error is neglected), BS denotes the Brier score and $H(x)$ denotes the Heaviside step function, which we define here as equal to 1 for $x \geq 0$ and 0 otherwise.

Parameters

- **forecast** (*xr.object*) – Forecast with member dim.
- **verif** (*xr.object*) – Verification data without member dim.
- **threshold** (*int, float, xr.object*) – Threshold to check exceedance, see properscoring.threshold_brier_score.

Details:

minimum	0.0
maximum	1.0
perfect	0.0
orientation	negative

Reference:

- Brier, Glenn W. Verification of forecasts expressed in terms of probability." Monthly Weather Review 78, no. 1 (1950). https://doi.org/10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2.

See also:

- properscoring.threshold_brier_score
- xskillscore.threshold_brier_score

Example

```
>>> compute_perfect_model(ds, control,
                          metric='threshold_brier_score', threshold=.5)
```

2.8.3 User-defined metrics

You can also construct your own metrics via the *climpred.metrics.Metric* class.

Metric(name, function, positive, ... [, ...])	Master class for all metrics.

First, write your own metric function, similar to the existing ones with required arguments `forecast`, `observations`, `dim=None`, and `**metric_kwargs`:

```
from climpred.metrics import Metric

def _my_msle(forecast, observations, dim=None, **metric_kwargs):
    """Mean squared logarithmic error (MSLE).
    https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-
↪model/loss-functions/mean-squared-logarithmic-error."""
    return ( (np.log(forecast + 1) + np.log(observations + 1) ) ** 2).mean(dim)
```

Then initialize this metric function with *climpred.metrics.Metric*:

```
_my_msle = Metric(
    name='my_msle',
    function=_my_msle,
    probabilistic=False,
    positive=False,
    unit_power=0,
    )
```

Finally, compute skill based on your own metric:

```
skill = compute_perfect_model(ds, control, metric=_my_msle)
```

Once you come up with an useful metric for your problem, consider contributing this metric to *climpred*, so all users can benefit from your metric, see contributing.

2.8.4 References

2.9 Comparisons

Forecasts have to be verified against some product to evaluate their performance. However, when verifying against a product, there are many different ways one can compare the ensemble of forecasts. Here we cover the comparison options for both hindcast and perfect model ensembles. See terminology for clarification on the differences between these two experimental setups.

Note that all compute functions (*compute_hindcast()*, *compute_perfect_model()*, *compute_hindcast()*, *bootstrap_hindcast()*, *bootstrap_perfect_model()*) take an optional `comparison=''` keyword to select the comparison style. See below for a detailed description on the differences between these comparisons.

2.9.1 Hindcast Ensembles

In hindcast ensembles, the ensemble mean forecast (`comparison='e2o'`) is expected to perform better than individual ensemble members (`comparison='m2o'`) as the chaotic component of forecasts is expected to be suppressed by this averaging, while the memory of the system sustains. [Boer2016]

`keyword: 'e2o', 'e2r'`

This is the default option.

_e2o(hind, verif[, metric])	Compare the ensemble mean forecast to the verification data for a HindcastEnsemble setup.

keyword: 'm2o', 'm2r'

_m2o(hind, verif[, metric])	Compares each ensemble member individually to the verification data for a HindcastEnsemble setup.

2.9.2 Perfect Model Ensembles

In perfect-model frameworks, there are many more ways of verifying forecasts. [Seferian2018] uses a comparison of all ensemble members against the control run (comparison='m2c') and all ensemble members against all other ensemble members (comparison='m2m'). Furthermore, the ensemble mean forecast can be verified against one control member (comparison='e2c') or all members (comparison='m2e') as done in [Griffies1997].

keyword: 'm2e'

This is the default option.

_m2e(ds[, metric])	Compare all members to ensemble mean while leaving out the reference in

keyword: 'm2c'

_m2c(ds[, control_member, metric])	Compare all other members forecasts to control member verification.

keyword: 'm2m'

_m2m(ds[, metric])	Compare all members to all others in turn while leaving out the verification member.

keyword: 'e2c'

_e2c(ds[, control_member, metric])	Compare ensemble mean forecast to control member verification.

2.9.3 Normalization

The goal of a normalized distance metric is to get a constant or comparable value of typically 1 (or 0 for metrics defined as 1 - metric) when the metric saturates and the predictability horizon is reached (see metrics).

A factor is added in the normalized metric formula (see [Seferian2018]) to accomodate different comparison styles. For example, nrmse gets smalled in comparison m2e than m2m by design, since the ensembe mean is always closer to individual members than the ensemble members to each other. In turn, the normalization factor is 2 for comparisons m2c, m2m, and m2o. It is 1 for m2e, e2c, and e2o.

2.9.4 Interpretation of Results

While `HindcastEnsemble` skill is computed over all initializations `init` of the hindcast, the resulting skill is a mean forecast skill over all initializations.

`PerfectModelEnsemble` skill is computed over a supervector comprised of all initializations and members, which allows the computation of the ACC-based skill [Bushuk2018], but also returns a mean forecast skill over all initializations.

The supervector approach shown in [Bushuk2018] and just calculating a distance-based metric like `rmse` over the member dimension as in [Griffies1997] yield very similar results.

2.9.5 Compute over dimension

The optional argument `dim` defines over which dimension a metric is computed. We can apply a metric over `dim` from `['init', 'member', ['member', 'init']]` in *compute_perfect_model()* and `['init', 'member']` in *compute_hindcast()*. The resulting skill is then reduced by this `dim`. Therefore, applying a metric over `dim='member'` creates a skill for all initializations individually. This can show the initial conditions dependence of skill. Likewise when computing skill over `'init'`, we get skill for each member. This `dim` argument is different from the `comparison` argument which just specifies how `forecast` and `observations` are defined.

However, this above logic applies to deterministic metrics. Probabilistic metrics need to be applied to the `member` dimension and `comparison` from `['m2c', 'm2m']` in *compute_perfect_model()* and `'m2o'` comparison in *compute_hindcast()*. Using a probabilistic metric automatically switches internally to using `dim='member'`.

2.9.6 User-defined comparisons

You can also construct your own comparisons via the *Comparison* class.

Comparison(name, function, hindcast, ... [, ...])	Master class for all comparisons.

First, write your own comparison function, similar to the existing ones. If a comparison should also be used for probabilistic metrics, make sure that `metric.probabilistic` returns `forecast` with `member` dimension and `observations` without. For deterministic metrics, return `forecast` and `observations` with identical dimensions but without an identical comparison:

```
from climpred.comparisons import Comparison, _drop_members

def _my_m2median_comparison(ds, metric=None):
    """Identical to m2e but median."""
    observations_list = []
    forecast_list = []
    supervector_dim = 'member'
    for m in ds.member.values:
        forecast = _drop_members(ds, rmd_member=[m]).median('member')
        observations = ds.sel(member=m).squeeze()
        forecast_list.append(forecast)
        observations_list.append(observations)
    observations = xr.concat(observations_list, supervector_dim)
    forecast = xr.concat(forecast_list, supervector_dim)
    forecast[supervector_dim] = np.arange(forecast[supervector_dim].size)
    observations[supervector_dim] = np.arange(observations[supervector_dim].size)
    return forecast, observations
```

Then initialize this comparison function with *Comparison*:

```
__my_m2median_comparison = Comparison(
    name='m2me',
    function=_my_m2median_comparison,
    probabilistic=False,
    hindcast=False)
```

Finally, compute skill based on your own comparison:

```
skill = compute_perfect_model(ds, control,
                              metric='rmse',
                              comparison=__my_m2median_comparison)
```

Once you come up with an useful comparison for your problem, consider contributing this comparison to `climpred`, so all users can benefit from your comparison, see contributing.

2.9.7 References

2.10 Significance Testing

Significance testing is important for assessing whether a given initialized prediction system is skillful. Some questions that significance testing can answer are:

- Is the correlation coefficient of a lead time series significantly different from zero?
- What is the probability that the retrospective forecast is more valuable than a historical simulation?
- Are correlation coefficients statistically significant despite temporal and spatial autocorrelation?

All of these questions deal with statistical significance. See below on how to use `climpred` to address these questions. Please also have a look at the significance testing example.

2.10.1 p value for temporal correlations

For the correlation metrics, like *`_pearson_r()`* and *`_spearman_r()`*, `climpred` also hosts the associated p-value, like *`_pearson_r_p_value()`*, that this correlation is significantly different from zero. *`_pearson_r_eff_p_value()`* also incorporates the reduced degrees of freedom due to temporal autocorrelation. See example.

2.10.2 Bootstrapping with replacement

Testing statistical significance through bootstrapping is commonly used in the field of climate prediction [could add some example citations here]. Bootstrapping relies on resampling the underlying data with replacement for a large number of iterations, as proposed by the decadal prediction framework of Goddard et al. 2013 [Goddard2013]. This means that the initialized ensemble is resampled with replacement along a dimension (`init` or `member`) and then that resampled ensemble is verified against the observations. This leads to a distribution of initialized skill. Further, a reference forecast uses the resampled initialized ensemble, e.g. `compute_persistence()`, which creates a reference skill distribution. Lastly, an uninitialized skill distribution is created from the underlying historical members or the control simulation.

The probability or p value is the fraction of these resampled initialized metrics beaten by the uninitialized or resampled reference metrics calculated from their respective distributions. Confidence intervals using these distributions are also calculated.

This behavior is incorporated into `climpred` by the base function *`bootstrap_compute()`*, which is wrapped by *`bootstrap_hindcast()`* and *`bootstrap_perfect_model()`* for the respective prediction simulation type. See example

2.10.3 Field significance

Please use `esmtools.testing.multipletests()` to control the false discovery rate (FDR) in geospatial data from the above obtained p-values [Wilks2016]. See the FDR example.

2.10.4 References

2.11 Prediction Terminology

Terminology is often confusing and highly variable amongst those that make predictions in the geoscience community. Here we define some common terms in climate prediction and how we use them in `climpred`.

2.11.1 Simulation Design

Hindcast Ensemble: `m` ensemble members are initialized from a simulation (generally a reconstruction from reanalysis) or an analysis (representing the current state of the atmosphere, land, and ocean by assimilation of obsevations) at `n` initialization dates and integrated for `l` lead years [Boer2016] (*`HindcastEnsemble`*).

Perfect Model Experiment: `m` ensemble members are initialized from a control simulation at `n` randomly chosen initialization dates and integrated for `l` lead years [Griffies1997] (*`PerfectModelEnsemble`*).

Reconstruction/Assimilation: A "reconstruction" is a model solution that uses observations in some capacity to approximate historical or current conditions of the atmosphere, ocean, sea ice, and/or land. This could be done via a forced simulation, such as an OMIP run that uses a dynamical ocean/sea ice core with reanalysis forcing from atmospheric winds. This could also be a fully data assimilative model, which assimilates observations into the model solution. For weather, subseasonal, and seasonal predictions, the terms re-analysis and analysis are the terms typically used, while reconstruction is more commonly used for decadal predictions.

Uninitialized Ensemble: In this framework, an *uninitialized ensemble* is one that is generated by perturbing initial conditions only at one point in the historical run. These are generated via micro (round-off error perturbations) or macro (starting from completely different restart files) methods. Uninitialized ensembles are used to approximate the magnitude of internal climate variability and to confidently extract the forced response (ensemble mean) in the climate system. In `climpred`, we use uninitialized ensembles as a baseline for how important (reoccurring) initializations are for lending predictability to the system. Some modeling centers (such as NCAR) provide a dynamical uninitialized ensemble (the CESM Large Ensemble) along with their initialized prediction system (the CESM Decadal Prediction Large Ensemble). If this isn't available, one can approximate the unintiailized response by bootstrapping a control simulation.

2.11.2 Forecast Assessment

Accuracy: The average degree of correspondence between individual pairs of forecasts and observations [Murphy1988]; [Jolliffe2011]. Examples include Mean Absolute Error (MAE) and Mean Square Error (MSE). See metrics.

Association: The overall strength of the relationship between individual pairs of forecasts and observations [Jolliffe2011]. The primary measure of association is the Anomaly Correlation Coefficient (ACC), which can be measured using the Pearson product-moment correlation or Spearman's Rank correlation. See metrics.

(Potential) Predictability: This characterizes the "ability to be predicted" rather than the current "capability to predict." One estimates this by computing a metric (like the anomaly correlation coefficient (ACC)) between the prediction ensemble and a member (or collection of members) selected as the verification member(s) (in a perfect-model setup) or the reconstruction that initialized it (in a hindcast setup) [Meehl2013] [Pegion2017].

(Prediction) Skill: This characterizes the current ability of the ensemble forecasting system to predict the real world. This is derived by computing a metric between the prediction ensemble and observations, reanalysis, or analysis of the real world [Meehl2013] [Pegion2017].

Skill Score: The most generic skill score can be defined as the following [Murphy1988]:

$$S = \\ frac A_f - A_r A_p - A_r,$$

where A_f, A_p, and A_r represent the accuracy of the forecast being assessed, the accuracy of a perfect forecast, and the accuracy of the reference forecast (e.g. persistence), respectively [Murphy1985]. Here, S represents the improvement in accuracy of the forecasts over the reference forecasts relative to the total possible improvement in accuracy. They are typically designed to take a value of 1 for a perfect forecast and 0 for equivelant to the reference forecast [Jolliffe2011].

2.11.3 Forecasting

Hindcast: Retrospective forecasts of the past initialized from a reconstruction integrated forward in time, also called re-forcasts. Depending on the length of time of the integeration, external forcings may or may not be included. The longer the integeration (e.g. decadal vs. daily), the more important it is to include external forcnig. [Boer2016]. Because they represent so-called forecasts over periods that already occurred, their prediction skill can be evaluated.

Prediction: Forecasts initialized from a reconstruction integrated into the future. Depending on the length of time of the integeration, external forcings may or may not be included. The longer the integeration (e.g. decadal vs. daily), the more important it is to include external forcing. [Boer2016] Because predictions are made into the future, it is necessary to wait until the forecast occurs before one can quantify the skill of the forecast.

Projection An estimate of the future climate that is dependent on the externally forced climate response, such as anthropogenic greenhouse gases, aerosols, and volcanic eruptions [Meehl2013].

2.11.4 References

2.12 Reference Forecasts

To quantify the quality of an initialized forecast, it is useful to judge it against some simple reference forecast. `climpred` currently supports a persistence forecast, but future releases will allow computation of other reference forecasts. Consider opening a Pull Request to get it implemented more quickly.

Persistence Forecast: Whatever is observed at the time of initialization is forecasted to persist into the forecast period [Jolliffe2012]. You can compute this directly via `compute_persistence()` or as a method of `HindcastEnsemble` and `PerfectModelEnsemble`.

Damped Persistence Forecast: (*Not Implemented*) The amplitudes of the anomalies reduce in time exponentially at a time scale of the local autocorrelation [Yuan2016].

$$v_{dp}(t) = v(0)e^{-\alpha t}$$

Climatology: (*Not Implemented*) The average values at the temporal forecast resolution (e.g., annual, monthly) over some long period, which is usually 30 years [Jolliffe2012].

Random Mechanism: (*Not Implemented*) A probability distribution is assigned to the possible range of the variable being forecasted, and a sequence of forecasts is produced by taking a sequence of independent values from

that distribution [Jolliffe2012]. This would be similar to computing an uninitialized forecast, using `climpred`'s *`compute_uninitialized()`* function.

2.12.1 References

Help & Reference

- *API Reference*
- *What's New*
- *Helpful Links*
- *Publications Using climpred*
- *Contribution Guide*
- *Code of Conduct*
- *Release Procedure*
- *Contributors*

2.13 API Reference

This page provides an auto-generated summary of climpred's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

2.13.1 High-Level Classes

A primary feature of `climpred` is our prediction ensemble objects, *`HindcastEnsemble`* and *`PerfectModelEnsemble`*. Users can append their initialized ensemble to these classes, as well as an arbitrary number of verification products (assimilations, reconstructions, observations), control runs, and uninitialized ensembles.

HindcastEnsemble

A `HindcastEnsemble` is a prediction ensemble that is initialized off of some form of observations (an assimilation, renanalysis, etc.). Thus, it is anticipated that forecasts are verified against observation-like products. Read more about the terminology here.

`HindcastEnsemble`(xobj)	An object for climate prediction ensembles initialized by a data-like product.

climpred.classes.HindcastEnsemble

class climpred.classes.**HindcastEnsemble**(*xobj*)

An object for climate prediction ensembles initialized by a data-like product.

HindcastEnsemble is a sub-class of *PredictionEnsemble*. It tracks all verification data associated with the prediction ensemble for easy computation across multiple variables and products.

This object is built on *xarray* and thus requires the input object to be an *xarray* Dataset or DataArray.

__init__ (*xobj*)
Create a *HindcastEnsemble* object by inputting output from a prediction ensemble in *xarray* format.

Parameters **xobj** (*xarray object*) – decadal prediction ensemble output.

observations
Dictionary of verification data to associate with the decadal prediction ensemble.

uninitialized
Dictionary of companion (or bootstrapped) uninitialized ensemble run.

Methods

__init__(xobj)	Create a *HindcastEnsemble* object by inputting output from a prediction ensemble in *xarray* format.
add_observations(xobj, name)	Add a verification data with which to verify the initialized ensemble.
HindcastEnsemble.add_reference	
add_uninitialized(xobj)	Add a companion uninitialized ensemble for comparison to verification data.
HindcastEnsemble.compute_metric	
HindcastEnsemble.compute_persistence	
HindcastEnsemble.compute_uninitialized	
get_initialized()	Returns the xarray dataset for the initialized ensemble.
get_observations([name])	Returns xarray Datasets of the observations/verification data.
HindcastEnsemble.get_reference	
get_uninitialized()	Returns the xarray dataset for the uninitialized ensemble.
smooth([smooth_kws])	Smooth all entries of PredictionEnsemble in the same manner to be able to still calculate prediction skill afterwards.
verify([name, reference, metric, ...])	Verifies the initialized ensemble against observations/verification data.

Add and Retrieve Datasets

HindcastEnsemble.__init__(xobj)	Create a *HindcastEnsemble* object by inputting output from a prediction ensemble in *xarray* format.
HindcastEnsemble.add_observations(xobj, name)	Add a verification data with which to verify the initialized ensemble.
HindcastEnsemble.add_uninitialized(xobj)	Add a companion uninitialized ensemble for comparison to verification data.
HindcastEnsemble.get_initialized()	Returns the xarray dataset for the initialized ensemble.
HindcastEnsemble.get_observations([name])	Returns xarray Datasets of the observations/verification data.
HindcastEnsemble.get_uninitialized()	Returns the xarray dataset for the uninitialized ensemble.

climpred.classes.HindcastEnsemble.__init__

HindcastEnsemble.**__init__**(*xobj*)

Create a *HindcastEnsemble* object by inputting output from a prediction ensemble in *xarray* format.

Parameters **xobj** (*xarray object*) – decadal prediction ensemble output.

observations
Dictionary of verification data to associate with the decadal prediction ensemble.

climpred.classes.**uninitialized**
Dictionary of companion (or bootstrapped) uninitialized ensemble run.

climpred.classes.HindcastEnsemble.add_observations

HindcastEnsemble.**add_observations**(*xobj*, *name*)

Add a verification data with which to verify the initialized ensemble.

Parameters

- **xobj** (*xarray object*) – Dataset/DataArray to append to the HindcastEnsemble object.
- **name** (*str*) – Short name for referencing the verification data.

climpred.classes.HindcastEnsemble.add_uninitialized

HindcastEnsemble.**add_uninitialized**(*xobj*)

Add a companion uninitialized ensemble for comparison to verification data.

Parameters **xobj** (*xarray object*) – Dataset/DataArray of the uninitialzed ensemble.

climpred.classes.HindcastEnsemble.get_initialized

HindcastEnsemble.**get_initialized**()

Returns the xarray dataset for the initialized ensemble.

climpred.classes.HindcastEnsemble.get_observations

HindcastEnsemble.**get_observations**(*name=None*)

Returns xarray Datasets of the observations/verification data.

Parameters **name** (*str, optional*) – Name of the observations/verification data to return. If None, return dictionary of all observations/verification data.

Returns Dictionary of xarray Datasets (if name is None) or single xarray Dataset.

climpred.classes.HindcastEnsemble.get_uninitialized

HindcastEnsemble.**get_uninitialized**()

Returns the xarray dataset for the uninitialized ensemble.

Analysis Functions

HindcastEnsemble.verify([name, reference, ...])	Verifies the initialized ensemble against observations/verification data.
HindcastEnsemble.compute_persistence	
HindcastEnsemble.compute_uninitialized	

climpred.classes.HindcastEnsemble.verify

HindcastEnsemble.**verify**(*name=None*, *reference=None*, *metric='pearson_r'*, *comparison='e2o'*, *alignment='same_verifs'*, *dim='init'*)

Verifies the initialized ensemble against observations/verification data.

This will automatically verify against all shared variables between the initialized ensemble and observations/verification data.

Parameters

- **name** (*str*) – Short name of observations/verification data to compare to. If None, compare to all observations/verification data.
- **metric** (*str, default 'pearson_r'*) – Metric to apply for verification.
- **comparison** (*str, default 'e2o'*) – How to compare to the observations/verification data. ('e2o' for ensemble mean to observations/verification data. 'm2o' for each individual member to observations/verification data).
- **alignment** (*str*) – which inits or verification times should be aligned? - maximize/None: maximize the degrees of freedom by slicing hind and verif to a common time frame at each lead. - same_inits: slice to a common init frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of initializations. - same_verif: slice to a common/consistent verification time frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of verification dates.

Returns Dataset of comparison results (if comparing to one observational product), or dictionary of Datasets with keys corresponding to observations/verification data short name.

Pre-Processing

HindcastEnsemble.smooth([smooth_kws])	Smooth all entries of PredictionEnsemble in the same manner to be able to still calculate prediction skill afterwards.

climpred.classes.HindcastEnsemble.smooth

HindcastEnsemble.**smooth**(*smooth_kws='goddard2013'*)

Smooth all entries of PredictionEnsemble in the same manner to be able to still calculate prediction skill afterwards.

Parameters **xobj** (*xarray object*) – decadal prediction ensemble output.

smooth_kws

Dictionary to specify the dims to smooth compatible with *spatial_smoothing_xesmf*, *temporal_smoothing* or *spatial_smoothing_xrcoarsen*. Shortcut for Goddard et al. 2013 recommendations: 'goddard2013'

Type dict or str

Example: >>> PredictionEnsemble.smooth(smooth_kws={'time': 2,

'lat': 5, 'lon': 4'})

```
>>> PredictionEnsemble.smooth(smooth_kws='goddard2013')
```

PerfectModelEnsemble

A `PerfectModelEnsemble` is a prediction ensemble that is initialized off of a control simulation for a number of randomly chosen initialization dates. Thus, forecasts cannot be verified against real-world observations. Instead, they are compared to one another and to the original control run. Read more about the terminology here.

PerfectModelEnsemble(xobj)	An object for "perfect model" climate prediction ensembles.

climpred.classes.PerfectModelEnsemble

class climpred.classes.**PerfectModelEnsemble**(*xobj*)

An object for "perfect model" climate prediction ensembles.

PerfectModelEnsemble is a sub-class of *PredictionEnsemble*. It tracks the control run used to initialize the ensemble for easy computations, bootstrapping, etc.

This object is built on *xarray* and thus requires the input object to be an *xarray* Dataset or DataArray.

__init__(*xobj*)

Create a *PerfectModelEnsemble* object by inputting output from the control run in *xarray* format.

Parameters **xobj** (*xarray object*) – decadal prediction ensemble output.

control

Dictionary of control run associated with the initialized ensemble.

uninitialized

Dictionary of uninitialized run that is bootstrapped from the initialized run.

Methods

__init__(xobj)	Create a *PerfectModelEnsemble* object by inputting output from the control run in *xarray* format.
add_control(xobj)	Add the control run that initialized the climate prediction ensemble.
bootstrap([metric, comparison, sig, ...])	Bootstrap ensemble simulations with replacement.
compute_metric([metric, comparison])	Compares the initialized ensemble to the control run.
compute_persistence([metric])	Compute a simple persistence forecast for the control run.
compute_uninitialized([metric, comparison])	Compares the bootstrapped uninitialized run to the control run.

Continued on next page

Table 15 – continued from previous page

generate_uninitialized()	Generate an uninitialized ensemble by bootstrapping the initialized prediction ensemble.
get_control()	Returns the control as an xarray dataset.
get_initialized()	Returns the xarray dataset for the initialized ensemble.
get_uninitialized()	Returns the xarray dataset for the uninitialized ensemble.
smooth([smooth_kws])	Smooth all entries of PredictionEnsemble in the same manner to be able to still calculate prediction skill afterwards.

Add and Retrieve Datasets

PerfectModelEnsemble.__init__(xobj)	Create a *PerfectModelEnsemble* object by inputting output from the control run in *xarray* format.
PerfectModelEnsemble.add_control(xobj)	Add the control run that initialized the climate prediction ensemble.
PerfectModelEnsemble.get_initialized()	Returns the xarray dataset for the initialized ensemble.
PerfectModelEnsemble.get_control()	Returns the control as an xarray dataset.
PerfectModelEnsemble.get_uninitialized()	Returns the xarray dataset for the uninitialized ensemble.

climpred.classes.PerfectModelEnsemble.__init__

PerfectModelEnsemble.**__init__**(*xobj*)

Create a *PerfectModelEnsemble* object by inputting output from the control run in *xarray* format.

Parameters **xobj** (*xarray object*) – decadal prediction ensemble output.

control

Dictionary of control run associated with the initialized ensemble.

climpred.classes.**uninitialized**

Dictionary of uninitialized run that is bootstrapped from the initialized run.

climpred.classes.PerfectModelEnsemble.add_control

PerfectModelEnsemble.**add_control**(*xobj*)

Add the control run that initialized the climate prediction ensemble.

Parameters **xobj** (*xarray object*) – Dataset/DataArray of the control run.

climpred.classes.PerfectModelEnsemble.get_initialized

PerfectModelEnsemble.**get_initialized**()

Returns the xarray dataset for the initialized ensemble.

climpred.classes.PerfectModelEnsemble.get_control

PerfectModelEnsemble.**get_control**()
Returns the control as an xarray dataset.

climpred.classes.PerfectModelEnsemble.get_uninitialized

PerfectModelEnsemble.**get_uninitialized**()
Returns the xarray dataset for the uninitialized ensemble.

Analysis Functions

PerfectModelEnsemble.bootstrap([metric, ...])	Bootstrap ensemble simulations with replacement.
PerfectModelEnsemble.compute_metric([...])	Compares the initialized ensemble to the control run.
PerfectModelEnsemble.compute_persistence([...])	Compute a simple persistence forecast for the control run.
PerfectModelEnsemble.compute_uninitialized([...])	Compares the bootstrapped uninitialized run to the control run.

climpred.classes.PerfectModelEnsemble.bootstrap

PerfectModelEnsemble.**bootstrap**(*metric='pearson_r'*, *comparison='m2e'*, *sig=95*, *iterations=500*, *pers_sig=None*)
Bootstrap ensemble simulations with replacement.

Parameters

- **metric** (*str, default 'pearson_r'*) – Metric to apply for bootstrapping.
- **comparison** (*str, default 'm2e'*) – Comparison style for bootstrapping.
- **sig** (*int, default 95*) – Significance level for uninitialized and initialized comparison.
- **iterations** (*int, default 500*) – Number of resampling iterations for bootstrapping with replacement.
- **pers_sig** (*int, default None*) – If not None, the separate significance level for persistence.

Returns

Dictionary of Datasets for each variable applied to with the following variables:

- init_ci: confidence levels of init_skill.
- uninit_ci: confidence levels of uninit_skill.
- pers_ci: confidence levels of pers_skill.
- **p_uninit_over_init: p value of the hypothesis that the** difference of skill between the initialized and uninitialized simulations is smaller or equal to zero based on bootstrapping with replacement.

- **p_pers_over_init: p value of the hypothesis that the** difference of skill between the initialized and persistence simulations is smaller or equal to zero based on bootstrapping with replacement.

Reference:

- Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

climpred.classes.PerfectModelEnsemble.compute_metric

PerfectModelEnsemble.**compute_metric**(*metric='pearson_r'*, *comparison='m2m'*)

Compares the initialized ensemble to the control run.

Parameters

- **metric** (*str, default 'pearson_r'*) – Metric to apply in the comparison.
- **comparison** (*str, default 'm2m'*) – How to compare the climate prediction ensemble to the control.

Returns Result of the comparison as a Dataset.

climpred.classes.PerfectModelEnsemble.compute_persistence

PerfectModelEnsemble.**compute_persistence**(*metric='pearson_r'*)

Compute a simple persistence forecast for the control run.

Parameters **metric** (*str, default 'pearson_r'*) – Metric to apply to the persistence forecast.

Returns Dataset of persistence forecast results (if `refname` is declared), or dictionary of Datasets with keys corresponding to verification data name.

Reference:

- Chapter 8 (Short-Term Climate Prediction) in Van den Dool, Huug. Empirical methods in short-term climate prediction. Oxford University Press, 2007.

climpred.classes.PerfectModelEnsemble.compute_uninitialized

PerfectModelEnsemble.**compute_uninitialized**(*metric='pearson_r'*, *comparison='m2e'*)

Compares the bootstrapped uninitialized run to the control run.

Parameters

- **metric** (*str, default 'pearson_r'*) – Metric to apply in the comparison.
- **comparison** (*str, default 'm2m'*) – How to compare to the control run.
- **running** (*int, default None*) – Size of the running window for variance smoothing.

Returns Result of the comparison as a Dataset.

Generate Data

PerfectModelEnsemble.generate_uninitialized()	Generate an uninitialized ensemble by bootstrapping the initialized prediction ensemble.

climpred.classes.PerfectModelEnsemble.generate_uninitialized

PerfectModelEnsemble.**generate_uninitialized**()

Generate an uninitialized ensemble by bootstrapping the initialized prediction ensemble.

Returns Bootstrapped (uninitialized) ensemble as a Dataset.

2.13.2 Direct Function Calls

A user can directly call functions in `climpred`. This requires entering more arguments, e.g. the initialized ensemble `Dataset`/`xarray.core.dataarray.DataArray` directly as well as a verification product. Our object *HindcastEnsemble* and *PerfectModelEnsemble* wrap most of these functions, making the analysis process much simpler. Once we have wrapped all of the functions in their entirety, we will likely depricate the ability to call them directly.

Bootstrap

bootstrap_compute(hind, verif[, hist, ...])	Bootstrap compute with replacement.
bootstrap_hindcast(hind, hist, verif[, ...])	Bootstrap compute with replacement. Wrapper of
bootstrap_perfect_model(init_pm, control[, ...])	Bootstrap compute with replacement. Wrapper of
bootstrap_uninit_pm_ensemble_from_control(...)	Create a pseudo-ensemble from control run.
bootstrap_uninitialized_ensemble(hind, hist)	Resample uninitialized hindcast from historical members.
dpp_threshold(control[, sig, iterations, dim])	Calc DPP significance levels from re-sampled dataset.
varweighted_mean_period_threshold(control[, ...])	Calc the variance-weighted mean period significance levels from re-sampled dataset.

climpred.bootstrap.bootstrap_compute

climpred.bootstrap.**bootstrap_compute**(*hind*, *verif*, *hist=None*, *alignment='same_verifs'*, *metric='pearson_r'*, *comparison='m2e'*, *dim='init'*, *resample_dim='member'*, *sig=95*, *iterations=500*, *pers_sig=None*, *compute=<function compute_hindcast>*, *resample_uninit=<function bootstrap_uninitialized_ensemble>*, *reference_compute=<function compute_persistence>*, ***metric_kwargs*)

Bootstrap compute with replacement.

Parameters

- **hind** (*xr.Dataset*) – prediction ensemble.
- **verif** (*xr.Dataset*) – Verification data.
- **hist** (*xr.Dataset*) – historical/uninitialized simulation.

- **metric** (*str*) – *metric*. Defaults to 'pearson_r'.
- **comparison** (*str*) – *comparison*. Defaults to 'm2e'.
- **dim** (*str or list*) – dimension(s) to apply metric over. default: 'init'.
- **resample_dim** (*str*) – dimension to resample from. default: 'member':

```
- 'member': select a different set of members from hind
- 'init': select a different set of initializations from hind
```

- **sig** (*int*) – Significance level for uninitialized and initialized skill. Defaults to 95.
- **pers_sig** (*int*) – Significance level for persistence skill confidence levels. Defaults to sig.
- **iterations** (*int*) – number of resampling iterations (bootstrap with replacement). Defaults to 500.
- **compute** (*func*) – function to compute skill. Choose from [*climpred.prediction.compute_perfect_model()*, *climpred.prediction.compute_hindcast()*].
- **resample_uninit** (*func*) – function to create an uninitialized ensemble from a control simulation or uninitialized large ensemble. Choose from: [*bootstrap_uninitialized_ensemble()*, bootstrap_uninit_pm_ensemble_from_control()].
- **reference_compute** (*func*) – function to compute a reference forecast skill with. Default: climpred.prediction.compute_persistence().
- **metric_kwargs** (****) – additional keywords to be passed to metric (see the arguments required for a given metric in *Metrics*).

Returns

(xr.Dataset): bootstrapped results for the three different kinds of

predictions:

- *init* for the initialized hindcast *hind* and describes skill due to

  initialization and external forcing

- *uninit* for the uninitialized historical *hist* and approximates skill

  from external forcing

- *pers* for the reference forecast computed by *reference_compute*, which

  defaults to *compute_persistence*

the different results:

- *skill*: skill values
- *p*: p value
- *low_ci* and *high_ci*: high and low ends of confidence intervals based

  on significance threshold *sig*

Return type results

Reference:

- Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

See also:

- climpred.bootstrap.bootstrap_hindcast
- climpred.bootstrap.bootstrap_perfect_model

climpred.bootstrap.bootstrap_hindcast

climpred.bootstrap.**bootstrap_hindcast**(*hind*, *hist*, *verif*, *alignment='same_verifs'*, *metric='pearson_r'*, *comparison='e2o'*, *dim='init'*, *resample_dim='member'*, *sig=95*, *iterations=500*, *pers_sig=None*, *reference_compute=<function compute_persistence>*, ***metric_kwargs*)

Bootstrap compute with replacement. Wrapper of py:func:*bootstrap_compute* for hindcasts.

Parameters

- **hind** (*xr.Dataset*) – prediction ensemble.
- **verif** (*xr.Dataset*) – Verification data.
- **hist** (*xr.Dataset*) – historical/uninitialized simulation.
- **metric** (*str*) – *metric*. Defaults to 'pearson_r'.
- **comparison** (*str*) – *comparison*. Defaults to 'e2o'.
- **dim** (*str*) – dimension to apply metric over. default: 'init'.
- **resample_dim** (*str or list*) – dimension to resample from. default: 'member'.
  - 'member': select a different set of members from hind
  - 'init': select a different set of initializations from hind
- **sig** (*int*) – Significance level for uninitialized and initialized skill. Defaults to 95.
- **pers_sig** (*int*) – Significance level for persistence skill confidence levels. Defaults to sig.
- **iterations** (*int*) – number of resampling iterations (bootstrap with replacement). Defaults to 500.
- **reference_compute** (*func*) – function to compute a reference forecast skill with. Default: `climpred.prediction.compute_persistence()`.
- **metric_kwargs** (****) – additional keywords to be passed to metric (see the arguments required for a given metric in *Metrics*).

Returns

(xr.Dataset): bootstrapped results for the three different kinds of

predictions:

- *init* for the initialized hindcast *hind* and describes skill due to initialization and external forcing
- *uninit* for the uninitialized historical *hist* and approximates skill from external forcing
- *pers* for the reference forecast computed by *reference_compute*, which defaults to *compute_persistence*

the different results:

- *skill*: skill values
- *p*: p value
- *low_ci* and *high_ci*: high and low ends of confidence intervals based on significance threshold *sig*

Return type results

Reference:

- Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

See also:

- climpred.bootstrap.bootstrap_compute
- climpred.prediction.compute_hindcast

Example

```
>>> hind = climpred.tutorial.load_dataset('CESM-DP-SST')['SST']
>>> hist = climpred.tutorial.load_dataset('CESM-LE')['SST']
>>> obs = load_dataset('ERSST')['SST']
>>> bootstrapped_skill = climpred.bootstrap.bootstrap_hindcast(hind, hist, obs)
>>> bootstrapped_skill.coords
Coordinates:
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
  * kind     (kind) object 'init' 'pers' 'uninit'
  * results  (results) <U7 'skill' 'p' 'low_ci' 'high_ci'
```

climpred.bootstrap.bootstrap_perfect_model

climpred.bootstrap.**bootstrap_perfect_model**(*init_pm*, *control*, *metric='pearson_r'*, *comparison='m2e'*, *dim=None*, *resample_dim='member'*, *sig=95*, *iterations=500*, *pers_sig=None*, *reference_compute=<function compute_persistence>*, ***metric_kwargs*)

Bootstrap compute with replacement. Wrapper of py:func:*bootstrap_compute* for perfect-model framework.

Parameters

- **hind** (*xr.Dataset*) – prediction ensemble.
- **verif** (*xr.Dataset*) – Verification data.
- **hist** (*xr.Dataset*) – historical/uninitialized simulation.
- **metric** (*str*) – *metric*. Defaults to 'pearson_r'.
- **comparison** (*str*) – *comparison*. Defaults to 'm2e'.
- **dim** (*str*) – dimension to apply metric over. default: ['init', 'member'].
- **resample_dim** (*str or list*) – dimension to resample from. default: 'member'.
  - 'member': select a different set of members from hind
  - 'init': select a different set of initializations from hind
- **sig** (*int*) – Significance level for uninitialized and initialized skill. Defaults to 95.
- **pers_sig** (*int*) – Significance level for persistence skill confidence levels. Defaults to sig.
- **iterations** (*int*) – number of resampling iterations (bootstrap with replacement). Defaults to 500.
- **reference_compute** (*func*) – function to compute a reference forecast skill with. Default: `climpred.prediction.compute_persistence()`.
- **metric_kwargs** (****) – additional keywords to be passed to metric (see the arguments required for a given metric in *Metrics*).

Returns

(xr.Dataset): bootstrapped results for the three different kinds of

predictions:

- *init* for the initialized hindcast *hind* and describes skill due to

  initialization and external forcing

- *uninit* for the uninitialized historical *hist* and approximates skill

  from external forcing

- *pers* for the reference forecast computed by *reference_compute*, which

  defaults to *compute_persistence*

the different results:

- *skill*: skill values
- *p*: p value
- *low_ci* and *high_ci*: high and low ends of confidence intervals based

  on significance threshold *sig*

Return type results

Reference:

- Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

See also:

- climpred.bootstrap.bootstrap_compute
- climpred.prediction.compute_perfect_model

Example

```
>>> init = climpred.tutorial.load_dataset('MPI-PM-DP-1D')
>>> control = climpred.tutorial.load_dataset('MPI-control-1D')
>>> bootstrapped_s = climpred.bootstrap.bootstrap_perfect_model(init, control)
>>> bootstrapped_s.coords
Coordinates:
  * lead     (lead) int64 1 2 3 4 5 6 7 8 9 10
  * kind     (kind) object 'init' 'pers' 'uninit'
  * results  (results) <U7 'skill' 'p' 'low_ci' 'high_ci'
```

climpred.bootstrap.bootstrap_uninit_pm_ensemble_from_control_cftime

climpred.bootstrap.**bootstrap_uninit_pm_ensemble_from_control_cftime**(*init_pm*, *control*)

Create a pseudo-ensemble from control run.

Bootstrap random numbers for years to construct an uninitialized ensemble from. This assumes a continous control simulation without gaps.

Note: Needed for block bootstrapping a metric in perfect-model framework. Takes random segments of length `block_length` from control based on `dayofyear` (and therefore assumes a constant climate control simulation) and rearranges them into ensemble and member dimensions.

Parameters

- **init_pm** (*xarray object*) – initialized ensemble simulation.
- **control** (*xarray object*) – control simulation.

Returns uninitialized ensemble generated from control run.

Return type uninit_pm (xarray object)

climpred.bootstrap.bootstrap_uninitialized_ensemble

climpred.bootstrap.**bootstrap_uninitialized_ensemble**(*hind*, *hist*)
Resample uninitialized hindcast from historical members.

Note: Needed for bootstrapping confidence intervals and p_values of a metric in the hindcast framework. Takes hind.lead.size timesteps from historical at same forcing and rearranges them into ensemble and member dimensions.

Parameters

- **hind** (*xarray object*) – hindcast.
- **hist** (*xarray object*) – historical uninitialized.

Returns uninitialize hindcast with hind.coords.

Return type uninit_hind (xarray object)

climpred.bootstrap.dpp_threshold

climpred.bootstrap.**dpp_threshold**(*control*, *sig=95*, *iterations=500*, *dim='time'*, ***dpp_kwargs*)
Calc DPP significance levels from re-sampled dataset.

Reference:

- Feng, X., T. DelSole, and P. Houser. "Bootstrap Estimated Seasonal Potential Predictability of Global Temperature and Precipitation." Geophysical Research Letters 38, no. 7 (2011). https://doi.org/10/ft272w.

See also:

- climpred.bootstrap._bootstrap_func
- climpred.stats.dpp

climpred.bootstrap.varweighted_mean_period_threshold

climpred.bootstrap.**varweighted_mean_period_threshold**(*control*, *sig=95*, *iterations=500*, *time_dim='time'*)
Calc the variance-weighted mean period significance levels from re-sampled dataset.

See also:

- climpred.bootstrap._bootstrap_func
- climpred.stats.varweighted_mean_period

Prediction

compute_hindcast(hind, verif[, metric, ...])	Verify hindcast predictions against verification data.
compute_perfect_model(init_pm, control[, ...])	Compute a predictability skill score for a perfect-model framework simulation dataset.

climpred.prediction.compute_hindcast

climpred.prediction.**compute_hindcast**(*hind*, *verif*, *metric='pearson_r'*, *comparison='e2o'*, *dim='init'*, *alignment='same_verifs'*, *add_attrs=True*, ***metric_kwargs*)

Verify hindcast predictions against verification data.

Parameters

- **hind** (*xarray object*) – Hindcast ensemble. Expected to follow package conventions: * `init` : dim of initialization dates * `lead` : dim of lead time from those initializations Additional dims can be member, lat, lon, depth, ...
- **verif** (*xarray object*) – Verification data with some temporal overlap with the hindcast.
- **metric** (*str*) – Metric used in comparing the decadal prediction ensemble with the verification data. (see `get_metric_class()` and *Metrics*).
- **comparison** (*str*) – How to compare the decadal prediction ensemble to the verification data:
  - e2o : ensemble mean to verification data (Default)
  - m2o : each member to the verification data

  (see *Comparisons*)
- **dim** (*str or list*) – dimension to apply metric over. default: 'init'
- **alignment** (*str*) – which inits or verification times should be aligned? - maximize/None: maximize the degrees of freedom by slicing `hind` and `verif` to a common time frame at each lead. - same_inits: slice to a common init frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of initializations. - same_verif: slice to a common/consistent verification time frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of verification dates.
- **add_attrs** (*bool*) – write climpred compute args to attrs. default: True
- ****metric_kwargs** (*dict*) – additional keywords to be passed to metric (see the arguments required for a given metric in *Metrics*).

Returns Verification metric over `lead` reduced by dimension(s) `dim`.

Return type result (xarray object)

climpred.prediction.compute_perfect_model

climpred.prediction.**compute_perfect_model**(*init_pm*, *control*, *metric='pearson_r'*, *comparison='m2e'*, *dim=None*, *add_attrs=True*, ***metric_kwargs*)

Compute a predictability skill score for a perfect-model framework simulation dataset.

Parameters

- **init_pm** (*xarray object*) – ensemble with dims `lead`, `init`, `member`.
- **control** (*xarray object*) – control with dimension `time`.
- **metric** (*str*) – *metric* name, see `climpred.utils.get_metric_class()` and (see *Metrics*).
- **comparison** (*str*) – *comparison* name defines what to take as forecast and verification (see `climpred.utils.get_comparison_class()` and *Comparisons*).
- **dim** (*str or list*) – dimension to apply metric over. default: ['member', 'init']
- **add_attrs** (*bool*) – write climpred compute args to attrs. default: True
- **metric_kwargs** (****) – additional keywords to be passed to metric. (see the arguments required for a given metric in metrics.py)

Returns

skill score with dimensions as input *ds* without *dim*.

Return type skill (xarray object)

Reference

compute_persistence(hind, verif[, metric, ...])	Computes the skill of a persistence forecast from a simulation.
compute_uninitialized(hind, uninit, verif[, ...])	Verify an uninitialized ensemble against verification data.

climpred.reference.compute_persistence

climpred.reference.**compute_persistence**(*hind*, *verif*, *metric='pearson_r'*, *alignment='same_verifs'*, *add_attrs=True*, ***metric_kwargs*)

Computes the skill of a persistence forecast from a simulation.

Parameters

- **hind** (*xarray object*) – The initialized ensemble.
- **verif** (*xarray object*) – Verification data.
- **metric** (*str*) – Metric name to apply at each lag for the persistence computation. Default: 'pearson_r'
- **alignment** (*str*) – which inits or verification times should be aligned? - maximize/None: maximize the degrees of freedom by slicing `hind` and `verif` to a common time frame at each lead. - same_inits: slice to a common init frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of initializations. - same_verif: slice to a common/consistent verification time frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of verification dates.
- **add_attrs** (*bool*) – write climpred compute_persistence args to attrs. default: True
- **metric_kwargs** (****) – additional keywords to be passed to metric (see the arguments required for a given metric in *Metrics*).

Returns

Results of persistence forecast with the input metric applied.

Return type pers (xarray object)

Reference:

- Chapter 8 (Short-Term Climate Prediction) in Van den Dool, Huug. Empirical methods in short-term climate prediction. Oxford University Press, 2007.

climpred.reference.compute_uninitialized

climpred.reference.**compute_uninitialized**(*hind*, *uninit*, *verif*, *metric='pearson_r'*, *comparison='e2o'*, *dim='time'*, *alignment='same_verifs'*, *add_attrs=True*, ***metric_kwargs*)

Verify an uninitialized ensemble against verification data.

Note: Based on Decadal Prediction protocol, this should only be computed for the first lag and then projected out to any further lags being analyzed.

Parameters

- **hind** (*xarray object*) – Initialized ensemble.
- **uninit** (*xarray object*) – Uninitialized ensemble.
- **verif** (*xarray object*) – Verification data with some temporal overlap with the uninitialized ensemble.
- **metric** (*str*) – Metric used in comparing the uninitialized ensemble with the verification data.
- **comparison** (*str*) –

  How to compare the uninitialized ensemble to the verification data:

  - e2o : ensemble mean to verification data (Default)
  - m2o : each member to the verification data
- **alignment** (*str*) – which inits or verification times should be aligned? - maximize/None: maximize the degrees of freedom by slicing `hind` and `verif` to a common time frame at each lead. - same_inits: slice to a common init frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of initializations. - same_verif: slice to a common/consistent verification time frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of verification dates.
- **add_attrs** (*bool*) – write climpred compute args to attrs. default: True
- **metric_kwargs** (****) – additional keywords to be passed to metric

Returns Results from comparison at the first lag.

Return type u (xarray object)

Metrics

Metric(name, function, positive, ... [, ...])	Master class for all metrics.
_get_norm_factor(comparison)	Get normalization factor for normalizing distance metrics.

climpred.metrics.Metric

class climpred.metrics.**Metric**(*name*, *function*, *positive*, *probabilistic*, *unit_power*, *long_name=None*, *aliases=None*, *minimum=None*, *maximum=None*, *perfect=None*)

Master class for all metrics.

__init__(*name*, *function*, *positive*, *probabilistic*, *unit_power*, *long_name=None*, *aliases=None*, *minimum=None*, *maximum=None*, *perfect=None*)

Metric initialization.

Parameters

- **name** (*str*) – name of metric.
- **function** (*function*) – metric function.
- **positive** (*bool*) – Is metric positively oriented? Higher metric values means higher skill.
- **probabilistic** (*bool*) – Is metric probabilistic? *False* means deterministic.
- **unit_power** (*float, int*) – Power of the unit of skill based on unit of input, e.g. input unit [m]: skill unit [(m)**unit_power]
- **long_name** (*str, optional*) – long_name of metric. Defaults to None.
- **aliases** (*list of str, optional*) – Allowed aliases for this metric. Defaults to None.
- **min** (*float, optional*) – Minimum skill for metric. Defaults to None.
- **max** (*float, optional*) – Maxmimum skill for metric. Defaults to None.
- **perfect** (*float, optional*) – Perfect skill for metric. Defaults to None.

Returns metric class Metric.

Return type *Metric*

Methods

__init__(name, function, positive, ... [, ...])	Metric initialization.

climpred.metrics._get_norm_factor

climpred.metrics.**_get_norm_factor**(*comparison*)

Get normalization factor for normalizing distance metrics.

A distance metric is normalized by the standard deviation or variance of the verification product. The goal of a normalized distance metric is to get a constant and comparable value of typically 1 (or 0 for metrics defined as 1 - metric), when the metric saturates and the predictability horizon is reached.

To directly compare skill between different comparisons used, a factor is added in the normalized metric formula, see Seferian et al. 2018. For example, NRMSE gets smaller in comparison `m2e` than `m2m` by design, because the ensemble mean is always closer to individual ensemble members than ensemble members to each other.

Note: This is used for NMSE, NRMSE, MSSS, NMAE.

Parameters **comparison** (*class*) – comparison class.

Returns normalization factor.

Return type fac (int)

Raises **KeyError** – if comparison is not matching.

Example

```
>>> # check skill saturation value of roughly 1 for different comparisons
>>> metric = 'nrmse'
>>> for c in ['m2m', 'm2e', 'm2c', 'e2c']:
        s = compute_perfect_model(ds, control, metric=metric, comparison=c)
        s.plot(label=' '.join([metric,c]))
>>> plt.legend()
```

Reference:

- Séférian, Roland, Sarah Berthet, and Matthieu Chevallier. "Assessing the Decadal Predictability of Land and Ocean Carbon Uptake." Geophysical Research Letters, March 15, 2018. https://doi.org/10/gdb424.

Comparisons

Comparison(name, function, hindcast, . . . [, . . .])	Master class for all comparisons.

climpred.comparisons.Comparison

class climpred.comparisons.**Comparison**(*name*, *function*, *hindcast*, *probabilistic*, *long_name=None*, *aliases=None*)

Master class for all comparisons.

__init__(*name*, *function*, *hindcast*, *probabilistic*, *long_name=None*, *aliases=None*)

Comparison initialization.

Parameters

- **name** (*str*) – name of comparison.
- **function** (*function*) – comparison function.
- **hindcast** (*bool*) – Can comparison be used in *compute_hindcast*? *False* means *compute_perfect_model*
- **probabilistic** (*bool*) – Can this comparison be used for probabilistic metrics also? Probabilistic metrics require multiple forecasts. *False* means that comparison is only de-

terministic. *True* means that comparison can be used both deterministic and probabilistic.

- **long_name** (*str, optional*) – longname of comparison. Defaults to None.
- **aliases** (*list of str, optional*) – Allowed aliases for this comparison. Defaults to None.

Returns comparison class Comparison.

Return type comparison

Methods

__init__(name, function, hindcast, probabilistic)	Comparison initialization.

Statistics

autocorr(ds[, lag, dim, return_p])	Calculate the lagged correlation of time series.
corr(x, y[, dim, lag, return_p])	Computes the Pearson product-moment coefficient of linear correlation.
decorrelation_time(da[, r, dim])	Calculate the decorrelaton time of a time series.
dpp(ds[, dim, m, chunk])	Calculates the Diagnostic Potential Predictability (dpp)
rm_poly(ds, order[, dim])	Returns xarray object with nth-order fit removed.
rm_trend(da[, dim])	Remove linear trend from time series.
varweighted_mean_period(da[, dim])	Calculate the variance weighted mean period of time series based on xrft.power_spectrum.

climpred.stats.autocorr

climpred.stats.**autocorr**(*ds*, *lag=1*, *dim='time'*, *return_p=False*)

Calculate the lagged correlation of time series.

Parameters

- **ds** (*xarray object*) – Time series or grid of time series.
- **lag** (*optional int*) – Number of time steps to lag correlate to.
- **dim** (*optional str*) – Name of dimension to autocorrelate over.
- **return_p** (*optional bool*) – If True, return correlation coefficients and p values.

Returns

Pearson correlation coefficients.

If return_p, also returns their associated p values.

climpred.stats.corr

climpred.stats.**corr**(*x*, *y*, *dim='time'*, *lag=0*, *return_p=False*)

Computes the Pearson product-moment coefficient of linear correlation.

Note: This version calculates the effective degrees of freedom, accounting for autocorrelation within each time

series that could fluff the significance of the correlation.

Parameters

- **x** (*xarray object*) – Independent variable time series or grid of time series.
- **y** (*xarray object*) – Dependent variable time series or grid of time series
- **dim** (*optional str*) – Correlation dimension
- **lag** (*optional int*) – Lag to apply to correlaton, with x predicting y.
- **return_p** (*optional bool*) – If True, return correlation coefficients as well as p values.

Returns Pearson correlation coefficients If return_p True, associated p values.

References

- Wilks, Daniel S. Statistical methods in the atmospheric sciences. Vol. 100. Academic press, 2011.
- Lovenduski, Nicole S., and Nicolas Gruber. "Impact of the Southern Annular Mode on Southern Ocean circulation and biology." Geophysical Research Letters 32.11 (2005).

climpred.stats.decorrelation_time

climpred.stats.**decorrelation_time**(*da*, *r=20*, *dim='time'*)

Calculate the decorrelaton time of a time series.

$$\tau_d = 1 + 2 * \sum_{k=1}^{r} (\alpha_k)^k$$

Parameters

- **da** (*xarray object*) – Time series.
- **r** (*optional int*) – Number of iterations to run the above formula.
- **dim** (*optional str*) – Time dimension for xarray object.

Returns Decorrelation time of time series.

Reference:

- Storch, H. v, and Francis W. Zwiers. Statistical Analysis in Climate Research. Cambridge; New York: Cambridge University Press, 1999., p.373

climpred.stats.dpp

climpred.stats.**dpp**(*ds*, *dim='time'*, *m=10*, *chunk=True*)

Calculates the Diagnostic Potential Predictability (dpp)

$$DPP_{\text{unbiased}}(m) = \frac{\sigma_m^2 - \frac{1}{m} \cdot \sigma^2}{\sigma^2}$$

Note: Resplandy et al. 2015 and Seferian et al. 2018 calculate unbiased DPP in a slightly different way: chunk=False.

Parameters

- **ds** (*xr.DataArray*) – control simulation with time dimension as years.
- **dim** (*str*) – dimension to apply DPP on. Default: time.
- **m** (*optional int*) – separation time scale in years between predictable low-freq component and high-freq noise.
- **chunk** (*optional boolean*) – Whether chunking is applied. Default: True. If False, then uses Resplandy 2015 / Seferian 2018 method.

Returns ds without time dimension.

Return type dpp (xr.DataArray)

References

- Boer, G. J. "Long Time-Scale Potential Predictability in an Ensemble of Coupled Climate Models." Climate Dynamics 23, no. 1 (August 1, 2004): 29–44. https://doi.org/10/csjjbh.
- Resplandy, L., R. Séférian, and L. Bopp. "Natural Variability of CO2 and O2 Fluxes: What Can We Learn from Centuries-Long Climate Models Simulations?" Journal of Geophysical Research: Oceans 120, no. 1 (January 2015): 384–404. https://doi.org/10/f63c3h.
- Séférian, Roland, Sarah Berthet, and Matthieu Chevallier. "Assessing the Decadal Predictability of Land and Ocean Carbon Uptake." Geophysical Research Letters, March 15, 2018. https://doi.org/10/gdb424.

climpred.stats.rm_poly

climpred.stats.**rm_poly**(*ds*, *order*, *dim='time'*)

Returns xarray object with nth-order fit removed.

Note: This automatically performs a linear interpolation across any NaNs in the time series.

Parameters

- **ds** (*xarray object*) – Time series to be detrended.
- **order** (*int*) – Order of polynomial fit to be removed.
- **dim** (*optional str*) – Dimension over which to remove the polynomial fit.

Returns xarray object with polynomial fit removed.

climpred.stats.rm_trend

climpred.stats.**rm_trend**(*da*, *dim='time'*)

Remove linear trend from time series.

Parameters

- **ds** (*xarray object*) – Time series to be detrended.
- **dim** (*optional str*) – Dimension over which to remove the linear trend.

Returns xarray object with linear trend removed.

climpred.stats.varweighted_mean_period

climpred.stats.**varweighted_mean_period**(*da*, *dim='time'*, ***kwargs*)

Calculate the variance weighted mean period of time series based on xrft.power_spectrum.

$$P_x = \frac{\sum_k V(f_k, x)}{\sum_k f_k \cdot V(f_k, x)}$$

Parameters

- **da** (*xarray object*) – input data including dim.
- **dim** (*optional str*) – Name of time dimension.
- ****kwargs see xrft.power_spectrum** (*for*) –

Reference:

- Branstator, Grant, and Haiyan Teng. "Two Limits of Initial-Value Decadal Predictability in a CGCM." Journal of Climate 23, no. 23 (August 27, 2010): 6292-6311. https://doi.org/10/bwq92h.

See also: https://xrft.readthedocs.io/en/latest/api.html#xrft.xrft.power_spectrum

Tutorial

load_dataset([name, cache, cache_dir, ...])	Load example data or a mask from an online repository.

climpred.tutorial.load_dataset

climpred.tutorial.**load_dataset**(*name=None*, *cache=True*, *cache_dir='~/.climpred_data'*, *github_url='https://github.com/bradyrx/climpred-data'*, *branch='master'*, *extension=None*, *proxy_dict=None*, ***kws*)

Load example data or a mask from an online repository.

Parameters

- **name** – (str, default None) Name of the netcdf file containing the dataset, without the .nc extension. If None, this function prints out the available datasets to import.
- **cache_dir** – (str, optional) The directory in which to search for and cache the data.
- **cache** – (bool, optional) If True, cache data locally for use on later calls.
- **github_url** – (str, optional) Github repository where the data is stored.
- **branch** – (str, optional) The git branch to download from.
- **extension** – (str, optional) Subfolder within the repository where the data is stored.
- **proxy_dict** – (dict, optional) Dictionary with keys as either 'http' or 'https' and values as the proxy server. This is useful if you are on a work computer behind a firewall and need to use a proxy out to download data.

- **kws** – (dict, optional) Keywords passed to xarray.open_dataset

Returns The desired xarray dataset.

Examples

```
>>> from climpred.tutorial import load_dataset()
>>> proxy_dict = {'http': '127.0.0.1'}
>>> ds = load_dataset('FOSI-SST', cache=False, proxy_dict=proxy_dict)
```

Preprocessing

load_hindcast([inits, members, preprocess, ...])	Load multi-member, multi-initialization hindcast experiment into one *xr.Dataset* compatible with *climpred*.
rename_to_climpred_dims(xro)	Rename existing dimension in xr.object *xro* to *CLIMPRED_ENSEMBLE_DIMS* from existing dimension names.
rename_SLM_to_climpred_dims(xro)	Rename ensemble dimensions common to SubX or CESM output:

climpred.preprocessing.shared.load_hindcast

climpred.preprocessing.shared.**load_hindcast**(*inits=range(1961, 1965)*, *members=range(1, 3)*, *preprocess=None*, *lead_offset=1*, *parallel=True*, *engine=None*, *get_path=<function get_path>*, ***get_path_kwargs*)

Load multi-member, multi-initialization hindcast experiment into one *xr.Dataset* compatible with *climpred*.

Parameters

- **inits** (*list, array*) – List of initializations to be loaded. Defaults to range(1961, 1965).
- **members** (*list, array*) – List of initializations to be loaded. Defaults to range(1, 3).
- **preprocess** (*function*) – *preprocess* function accepting and returning *xr.Dataset* only. To be passed to `xarray.open_dataset()`. Defaults to None.
- **parallel** (*bool*) – passed to *xr.open_mfdataset*. Defaults to True.
- **engine** (*str*) – passed to *xr.open_mfdataset*. Defaults to None.

:param .. note::: To load MPI-ESM grb files, pass *engine='pynio'*. :param get_path: *get_path* function specific to modelling center output

format. Defaults to *get_path()*.

Parameters ****get_path_kwargs** (*dict*) – parameters passed to ***get_path*.

Returns *climpred* compatible dataset with dims: *member*, *init*, *lead*.

Return type xr.Dataset

climpred.preprocessing.shared.rename_to_climpred_dims

climpred.preprocessing.shared.**rename_to_climpred_dims**(*xro*)

Rename existing dimension in xr.object *xro* to *CLIMPRED_ENSEMBLE_DIMS* from existing dimension names. This function attempts to autocorrect dimension names to climpred standards. e.g., *ensemble_member* becomes *member* and *lead_time* becomes *lead*, and *time* gets renamed to *lead*.

Parameters

- **xro** (*xr.object*) – input from DCPP via intake-esm
- **dimension names like dcpp_init_year, time, member_id.** (*containing*) –

Returns *climpred* compatible with dimensions: *member*, *init*, *lead*.

Return type xr.object

climpred.preprocessing.shared.rename_SLM_to_climpred_dims

climpred.preprocessing.shared.**rename_SLM_to_climpred_dims**(*xro*)

Rename ensemble dimensions common to SubX or CESM output:

- `S` : Refers to start date and is changed to `init`
- `L` : Refers to lead time and is changed to `lead`
- `M`: Refers to ensemble member and is changed to `member`

Parameters **xro** (*xr.object*) – input from CESM/SubX containing dimensions: *S*, *L*, *M*.

Returns *climpred* compatible with dimensions: *member*, *init*, *lead*.

Return type xr.object

get_path([dir_base_experiment, member, ...])	Get the path of a file for MPI-ESM standard output file names and directory.

climpred.preprocessing.mpi.get_path

climpred.preprocessing.mpi.**get_path**(*dir_base_experiment='/work/bm1124/m300086/CMIP6/experiments'*, *member=1*, *init=1960*, *model='hamocc'*, *output_stream='monitoring_ym'*, *timestr='*1231'*, *ending='nc'*)

Get the path of a file for MPI-ESM standard output file names and directory.

Parameters

- **dir_base_experiment** (*str*) – Path of experiments folder. Defaults to "/work/bm1124/m300086/CMIP6/experiments".
- **member** (*int*) – member label. Defaults to 1.
- **init** (*int*) – initialization label. Typically year. Defaults to 1960.
- **model** (*str*) – submodel name. Defaults to "hamocc". Allowed: ['echam6', 'jsbach', 'mpiom', 'hamocc'].

- **output_stream** (*str*) – output_stream name. Defaults to "monitoring_ym". Allowed: ['data_2d_mm', 'data_3d_ym', 'BOT_mm', ...]
- **timestr** (*str*) – timestr likely including *. Defaults to "*1231".
- **ending** (*str*) – ending indicating file format. Defaults to "nc". Allowed: ['nc', 'grb'].

Returns path of requested file(s)

Return type str

2.14 What's New

2.14.1 climpred v2.1.0 (2020-04-##)

Breaking change

- replace keyword *bootstrap* with *iterations* (GH#354) Aaron Spring.
- HTML `__repr__` for *HindcastEnsemble* and *PerfectModelEnsemble* following *xarray.core.options.OPTIONS* (GH#371) Aaron Spring.

New Features

- Union with observations only enforced if `reference='persistence'`. (GH#341) Riley X. Brady.
- `HindcastEnsemble.verify()` now takes `reference=...` keyword. (GH#341) Riley X. Brady.
- compute_perfect_model now accepts inits as cftime and integer. Implemented cftime into bootstrap_uninit function, which requires a Leap or NoLeap calendar. (GH#332) Aaron Spring.
- speed-up in bootstrapping uninitialized skill in perfect-model for annual leads by reshaping. (GH#332) Aaron Spring.
- **speed-up in bootstrap functions: (GH#285) Aaron Spring.**
  - `xr.quantile` exchanged for `dask.map_blocks(np.percentile)`
  - properly implemented handling for lazy results when chunked inputs
  - user gets warned when chunking potentially (un)-necessary
- new explicit keywords in bootstrap functions for `resampling_dim` and `reference_compute` (GH#320) Aaron Spring.
- Logging now included for `compute_hindcast` which displays the inits and verification dates used at each lead (GH#324) Aaron Spring, (GH#338) Riley X. Brady..
- new explicit keywords added for `alignment` of verification dates and initializations. (GH#324) Aaron Spring.
  - `'maximize'`: Maximize the degrees of freedom by slicing `hind` and `verif` to a common time frame at each lead. (GH#338) Riley X. Brady.
  - `'same_inits'`: slice to a common init frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of initializations. (GH#328) Riley X. Brady.
  - `'same_verifs'`: slice to a common/consistent verification time frame prior to computing metric. This philosophy follows the thought that each lead should be based on the same set of verification dates. (GH#331) Riley X. Brady.

Bug Fixes

- Alignment options now account for differences in the historical time series if `reference='historical'`. (GH#341) Riley X. Brady.

Internals/Minor Fixes

- Added our Code of Conduct (GH#285) Aaron Spring.
- Gather all `pytest.fixture``s in ``conftest.py`. (GH#313) Aaron Spring.
- Move `x_METRICS` and `COMPARISONS` to `metrics.py` and `comparisons.py` in order to avoid circular import dependencies. (GH#315) Aaron Spring.
- `asv` benchmarks for `HindcastEnsemble` (GH#285) Aaron Spring.
- ignore irrelevant warnings in `pytest` and mark slow tests (GH#333) Aaron Spring.
- default `CONCAT_KWARGS` now in all `xr.concat` to speed up bootstrapping. (GH#330) Aaron Spring.
- Remove `member` coords for `m2c` comparison for probabilistic metrics. (GH#330) Aaron Spring.
- Refactoring *`compute_hindcast()`* and *`compute_perfect_model()`*. (GH#330) Aaron Spring.
- Changed lead0 coordinate modifications to be compliant with xarray=0.15.1 in *`compute_persistence()`*. (GH#348) Aaron Spring.
- Exchanged my_quantile with xr.quantile(skipna=False). (GH#348) Aaron Spring.
- Remove `sig` from `plot_bootstrapped_skill_over_leadyear()`. (GH#351) Aaron Spring.
- Require `xskillscore v0.0.15` and use their functions for effective sample size-based metrics. (:pr: *353*) Riley X. Brady.
- Faster bootstrapping without replacement used in threshold functions of climpred.stats (GH#354) Aaron Spring.
- Require `cftime v1.1.2`, which modifies their object handling to create 200-400x speedups in some basic operations. (GH#356) Riley X. Brady.
- Resample first and then calculate skill in *`bootstrap_perfect_model()`* and *`bootstrap_hindcast()`* (GH#355) Aaron Spring.

Documentation

- Demo and wrapper to setup your own raw model output compliant to `climpred` (GH#296) Aaron Spring.
- Demo using `intake-esm` with `climpred` (GH#296) Aaron Spring.
- Add Verification Alignment page explaining how initializations are selected and aligned with verification data. (GH#328) Riley X. Brady.

2.14.2 climpred v2.0.0 (2020-01-22)

New Features

- Add support for `days`, `pentads`, `weeks`, `months`, `seasons` for lead time resolution. `climpred` now requires a `lead` attribute "units" to decipher what resolution the predictions are at. (GH#294) Kathy Pegion and Riley X. Brady.

```
>>> hind = climpred.tutorial.load_dataset('CESM-DP-SST')
>>> hind.lead.attrs['units'] = 'years'
```

- `HindcastEnsemble` now has `.add_observations()` and `.get_observations()` methods. These are the same as `.add_reference()` and `.get_reference()`, which will be deprecated eventually. The name change clears up confusion, since "reference" is the appropriate name for a reference forecast, e.g. persistence. (GH#310) Riley X. Brady.
- `HindcastEnsemble` now has `.verify()` function, which duplicates the `.compute_metric()` function. We feel that `.verify()` is more clear and easy to write, and follows the terminology of the field. (GH#310) Riley X. Brady.
- `e2o` and `m2o` are now the preferred keywords for comparing hindcast ensemble means and ensemble members to verification data, respectively. (GH#310) Riley X. Brady.

Documentation

- New example pages for subseasonal-to-seasonal prediction using `climpred`. (GH#294) Kathy Pegion
  - Calculate the skill of the MJO index as a function of lead time (link).
  - Calculate the skill of the MJO index as a function of lead time for weekly data (link).
  - Calculate ENSO skill as a function of initial month vs. lead time (link).
  - Calculate Seasonal ENSO skill (link).
- Comparisons page rewritten for more clarity. (GH#310) Riley X. Brady.

Bug Fixes

- Fixed *m2m* broken comparison issue and removed correction (GH#290) Aaron Spring.

Internals/Minor Fixes

- Updates to `xskillscore` v0.0.12 to get a 30-50% speedup in compute functions that rely on metrics from there. (GH#309) Riley X. Brady.
- Stacking dims is handled by `comparisons`, no need for internal keyword `stack_dims`. Therefore `comparison` now takes `metric` as argument instead. (GH#290) Aaron Spring.
- `assign_attrs` now carries *dim* (GH#290) Aaron Spring.
- "reference" changed to "verif" throughout hindcast compute functions. This is more clear, since "reference" usually refers to a type of forecast, such as persistence. (GH#310) Riley X. Brady.
- `Comparison` objects can now have aliases. (GH#310) Riley X. Brady.

2.14.3 climpred v1.2.1 (2020-01-07)

Depreciated

- `mad` no longer a keyword for the median absolute error metric. Users should now use `median_absolute_error`, which is identical to changes in `xskillscore` version 0.0.10. (GH#283) Riley X. Brady

- `pacc` no longer a keyword for the p value associated with the Pearson product-moment correlation, since it is used by the correlation coefficient. (GH#283) Riley X. Brady
- `msss` no longer a keyword for the Murphy's MSSS, since it is reserved for the standard MSSS. (GH#283) Riley X. Brady

New Features

- Metrics `pearson_r_eff_p_value` and `spearman_r_eff_p_value` account for autocorrelation in computing p values. (GH#283) Riley X. Brady
- Metric `effective_sample_size` computes number of independent samples between two time series being correlated. (GH#283) Riley X. Brady
- Added keywords for metrics: (GH#283) Riley X. Brady
  - `'pval'` for `pearson_r_p_value`
  - `['n_eff', 'eff_n']` for `effective_sample_size`
  - `['p_pval_eff', 'pvalue_eff', 'pval_eff']` for `pearson_r_eff_p_value`
  - `['spvalue', 'spval']` for `spearman_r_p_value`
  - `['s_pval_eff', 'spvalue_eff', 'spval_eff']` for `spearman_r_eff_p_value`
  - `'nev'` for `nmse`

Internals/Minor Fixes

- `climpred` now requires `xarray` version 0.14.1 so that the `drop_vars()` keyword used in our package does not throw an error. (GH#276) Riley X. Brady
- Update to `xskillscore` version 0.0.10 to fix errors in weighted metrics with pairwise NaNs. (GH#283) Riley X. Brady
- `doc8` added to `pre-commit` to have consistent formatting on `.rst` files. (GH#283) Riley X. Brady
- Remove `proper` attribute on `Metric` class since it isn't used anywhere. (GH#283) Riley X. Brady
- Add testing for effective p values. (GH#283) Riley X. Brady
- Add testing for whether metric aliases are repeated/overwrite each other. (GH#283) Riley X. Brady
- `ppp` changed to `msess`, but keywords allow for `ppp` and `msss` still. (GH#283) Riley X. Brady

Documentation

- Expansion of metrics documentation with much more detail on how metrics are computed, their keywords, references, min/max/perfect scores, etc. (GH#283) Riley X. Brady
- Update terminology page with more information on metrics terminology. (GH#283) Riley X. Brady

2.14.4 climpred v1.2.0 (2019-12-17)

Depreciated

- Abbreviation `pval` depreciated. Use `p_pval` for `pearson_r_p_value` instead. (GH#264) Aaron Spring.

New Features

- Users can now pass a custom `metric` or `comparison` to compute functions. (GH#268) Aaron Spring.
  - See user-defined-metrics and user-defined-comparisons.
- New deterministic metrics (see metrics). (GH#264) Aaron Spring.
  - Spearman ranked correlation (spearman_r)
  - Spearman ranked correlation p-value (spearman_r_p_value)
  - Mean Absolute Deviation (mad)
  - Mean Absolute Percent Error (mape)
  - Symmetric Mean Absolute Percent Error (smape)
- Users can now apply arbitrary `xarray` methods to *`HindcastEnsemble`* and *`PerfectModelEnsemble`*. (GH#243) Riley X. Brady.
  - See the Prediction Ensemble objects demo page.
- Add "getter" methods to *`HindcastEnsemble`* and *`PerfectModelEnsemble`* to retrieve `xarray` datasets from the objects. (GH#243) Riley X. Brady.

```
>>> hind = climpred.tutorial.load_dataset('CESM-DP-SST')
>>> ref = climpred.tutorial.load_dataset('ERSST')
>>> hindcast = climpred.HindcastEnsemble(hind)
>>> hindcast = hindcast.add_reference(ref, 'ERSST')
>>> print(hindcast)
<climpred.HindcastEnsemble>
Initialized Ensemble:
    SST      (init, lead, member) float64 ...
ERSST:
    SST      (time) float32 ...
Uninitialized:
    None
>>> print(hindcast.get_initialized())
<xarray.Dataset>
Dimensions:  (init: 64, lead: 10, member: 10)
Coordinates:
* lead     (lead) int32 1 2 3 4 5 6 7 8 9 10
* member   (member) int32 1 2 3 4 5 6 7 8 9 10
* init     (init) float32 1954.0 1955.0 1956.0 1957.0 ... 2015.0 2016.0␣
↪2017.0
Data variables:
    SST      (init, lead, member) float64 ...
>>> print(hindcast.get_reference('ERSST'))
<xarray.Dataset>
Dimensions:  (time: 61)
Coordinates:
* time     (time) int64 1955 1956 1957 1958 1959 ... 2011 2012 2013 2014␣
↪2015
Data variables:
    SST      (time) float32 ...
```

- `metric_kwargs` can be passed to *`Metric`*. (GH#264) Aaron Spring.
  - See `metric_kwargs` under metrics.

Bug Fixes

- `compute_metric()` doesn't drop coordinates from the initialized hindcast ensemble anymore. (GH#258) Aaron Spring.
- Metric `uacc` does not crash when `ppp` negative anymore. (GH#264) Aaron Spring.
- Update `xskillscore` to version 0.0.9 to fix all-NaN issue with `pearson_r` and `pearson_r_p_value` when there's missing data. (GH#269) Riley X. Brady.

Internals/Minor Fixes

- Rewrote *`varweighted_mean_period()`* based on `xrft`. Changed `time_dim` to `dim`. Function no longer drops coordinates. (GH#258) Aaron Spring
- Add `dim='time'` in *`dpp()`*. (GH#258) Aaron Spring
- Comparisons `m2m`, `m2e` rewritten to not stack dims into supervector because this is now done in `xskillscore`. (GH#264) Aaron Spring
- Add `tqdm` progress bar to *`bootstrap_compute()`*. (GH#244) Aaron Spring
- Remove inplace behavior for *`HindcastEnsemble`* and *`PerfectModelEnsemble`*. (GH#243) Riley X. Brady
  - See demo page on prediction ensemble objects
- Added tests for chunking with `dask`. (GH#258) Aaron Spring
- Fix test issues with esmpy 8.0 by forcing esmpy 7.1 (GH#269). Riley X. Brady
- Rewrote `metrics` and `comparisons` as classes to accomodate custom metrics and comparisons. (GH#268) Aaron Spring
  - See user-defined-metrics and user-defined-comparisons.

Documentation

- Add examples notebook for temporal and spatial smoothing. (GH#244) Aaron Spring
- Add documentation for computing a metric over a specified dimension. (GH#244) Aaron Spring
- Update API to be more organized with individual function/class pages. (GH#243) Riley X. Brady.
- Add page describing the *`HindcastEnsemble`* and *`PerfectModelEnsemble`* objects more clearly. (GH#243) Riley X. Brady
- Add page for publications and helpful links. (GH#270) Riley X. Brady.

2.14.5 climpred v1.1.0 (2019-09-23)

Features

- Write information about skill computation to netcdf attributes(GH#213) Aaron Spring
- Temporal and spatial smoothing module (GH#224) Aaron Spring
- Add metrics *brier_score*, *threshold_brier_score* and *crpss_es* (GH#232) Aaron Spring

- Allow *compute_hindcast* and *compute_perfect_model* to specify which dimension *dim* to calculate metric over (GH#232) Aaron Spring

Bug Fixes

- Correct implementation of probabilistic metrics from *xskillscore* in *compute_perfect_model*, *bootstrap_perfect_model*, *compute_hindcast* and *bootstrap_hindcast*, now requires xskillscore>=0.05 (GH#232) Aaron Spring

Internals/Minor Fixes

- Rename .stats.DPP to dpp (GH#232) Aaron Spring
- Add *matplotlib* as a main dependency so that a direct pip installation works (GH#211) Riley X. Brady.
- `climpred` is now installable from conda-forge (GH#212) Riley X. Brady.
- Fix erroneous descriptions of sample datasets (GH#226) Riley X. Brady.
- Benchmarking time and peak memory of compute functions with *asv* (GH#231) Aaron Spring

Documentation

- Add scope of package to docs for clarity for users and developers. (GH#235) Riley X. Brady.

2.14.6 climpred v1.0.1 (2019-07-04)

Bug Fixes

- Accomodate for lead-zero within the `lead` dimension (GH#196) Riley X. Brady.
- Fix issue with adding uninitialized ensemble to `HindcastEnsemble` object (GH#199) Riley X. Brady.
- Allow `max_dof` keyword to be passed to `compute_metric` and `compute_persistence` for `HindcastEnsemble` (GH#199) Riley X. Brady.

Internals/Minor Fixes

- Force `xskillscore` version 0.0.4 or higher to avoid `ImportError` (GH#204) Riley X. Brady.
- Change `max_dfs` keyword to `max_dof` (GH#199) Riley X. Brady.
- Add testing for `HindcastEnsemble` and `PerfectModelEnsemble` (GH#199) Riley X. Brady

2.14.7 climpred v1.0.0 (2019-07-03)

`climpred` v1.0.0 represents the first stable release of the package. It includes `HindcastEnsemble` and `PerfectModelEnsemble` objects to perform analysis with. It offers a suite of deterministic and probabilistic metrics that are optimized to be run on single time series or grids of data (e.g., lat, lon, and depth). Currently, `climpred` only supports annual forecasts.

Features

- Bootstrap prediction skill based on resampling with replacement consistently in `ReferenceEnsemble` and `PerfectModelEnsemble`. (GH#128) Aaron Spring
- Consistent bootstrap function for `climpred.stats` functions via `bootstrap_func` wrapper. (GH#167) Aaron Spring
- many more metrics: `_msss_murphy`, `_less` and probabilistic `_crps`, `_crpss` (GH#128) Aaron Spring

Bug Fixes

- `compute_uninitialized` now trims input data to the same time window. (GH#193) Riley X. Brady
- `rm_poly` now properly interpolates/fills NaNs. (GH#192) Riley X. Brady

Internals/Minor Fixes

- The `climpred` version can be printed. (GH#195) Riley X. Brady
- Constants are made elegant and pushed to a separate module. (GH#184) Andrew Huang
- Checks are consolidated to their own module. (GH#173) Andrew Huang

Documentation

- Documentation built extensively in multiple PRs.

2.14.8 climpred v0.3 (2019-04-27)

`climpred` v0.3 really represents the entire development phase leading up to the version 1 release. This was done in collaboration between Riley X. Brady, Aaron Spring, and Andrew Huang. Future releases will have less additions.

Features

- Introduces object-oriented system to `climpred`, with classes `ReferenceEnsemble` and `PerfectModelEnsemble`. (GH#86) Riley X. Brady
- Expands bootstrapping module for perfect-module configurations. (GH#78, GH#87) Aaron Spring
- Adds functions for computing Relative Entropy (GH#73) Aaron Spring
- Sets more intelligible dimension expectations for `climpred` (GH#98, GH#105) Riley X. Brady and Aaron Spring:
  - `init`: initialization dates for the prediction ensemble
  - `lead`: retrospective forecasts from prediction ensemble; returned dimension for prediction calculations
  - `time`: time dimension for control runs, references, etc.
  - `member`: ensemble member dimension.
- Updates `open_dataset` to display available dataset names when no argument is passed. (GH#123) Riley X. Brady
- Change `ReferenceEnsemble` to `HindcastEnsemble`. (GH#124) Riley X. Brady

- Add probabilistic metrics to `climpred`. (GH#128) Aaron Spring
- Consolidate separate perfect-model and hindcast functions into singular functions (GH#128) Aaron Spring
- Add option to pass proxy through to `open_dataset` for firewalled networks. (GH#138) Riley X. Brady

Bug Fixes

- `xr_rm_poly` can now operate on Datasets and with multiple variables. It also interpolates across NaNs in time series. (GH#94) Andrew Huang
- Travis CI, `treon`, and `pytest` all run for automated testing of new features. (GH#98, GH#105, GH#106) Riley X. Brady and Aaron Spring
- Clean up `check_xarray` decorators and make sure that they work. (GH#142) Andrew Huang
- Ensures that `help()` returns proper docstring even with decorators. (GH#149) Andrew Huang
- Fixes bootstrap so p values are correct. (GH#170) Aaron Spring

Internals/Minor Fixes

- Adds unit testing for all perfect-model comparisons. (GH#107) Aaron Spring
- Updates CESM-LE uninitialized ensemble sample data to have 34 members. (GH#113) Riley X. Brady
- Adds MPI-ESM hindcast, historical, and assimilation sample data. (GH#119) Aaron Spring
- Replaces `check_xarray` with a decorator for checking that input arguments are xarray objects. (GH#120) Andrew Huang
- Add custom exceptions for clearer error reporting. (GH#139) Riley X. Brady
- Remove "xr" prefix from stats module. (GH#144) Riley X. Brady
- Add codecoverage for testing. (GH#152) Riley X. Brady
- Update exception messages for more pretty error reporting. (GH#156) Andrew Huang
- Add `pre-commit` and `flake8`/`black` check in CI. (GH#163) Riley X. Brady
- Change `loadutils` module to `tutorial` and `open_dataset` to `load_dataset`. (GH#164) Riley X. Brady
- Remove predictability horizon function to revisit for v2. (GH#165) Riley X. Brady
- Increase code coverage through more testing. (GH#167) Aaron Spring
- Consolidates checks and constants into modules. (GH#173) Andrew Huang

2.14.9 climpred v0.2 (2019-01-11)

Name changed to `climpred`, developed enough for basic decadal prediction tasks on a perfect-model ensemble and reference-based ensemble.

2.14.10 climpred v0.1 (2018-12-20)

Collaboration between Riley Brady and Aaron Spring begins.

2.15 Helpful Links

We hope to curate in the `climpred` documentation a comprehensive report of terminology, best practices, analysis methods, etc. in the prediction community. Here we suggest other resources for initialized prediction of the Earth system to round out the information provided in our documentation.

2.15.1 Forecast Verification

- CAWCR Forecast Verification Overview: A nice overview of forecast verification, including a suite of metrics and their derivation.

2.16 Publications Using `climpred`

Below is a list of publications that have made use of `climpred` in their analysis. You can nod to `climpred`, e.g., in your acknowledgements section to help build the community. The main developers of the package intend to release a manuscript documenting `climpred` in 2020 with a citable DOI, so this can be referenced in the future.

Feel free to open a Pull Request to add your publication to the list!

2.16.1 2020

- Brady, R.X., Lovenduski, N.S., Yeager, S.G., Long, M.C., Lindsay, K (2020). Skillful multiyear predictions of ocean acidification in the California Current System. *Nature Communications*, 11, 2166. https://doi.org/10.1038/s41467-020-15722-x
- Spring, A., Ilyina, T. (2020). Predictability horizons in theglobal carbon cycle inferred from a perfect-model framework. *Geophysical Research Letters*, 47, e2019GL085311. https://doi.org/10.1029/2019GL085311

2.17 Contribution Guide

Contributions are highly welcomed and appreciated. Every little help counts, so do not hesitate! You can make a high impact on `climpred` just by using it and reporting issues.

The following sections cover some general guidelines regarding development in `climpred` for maintainers and contributors.

Please also review our Code of Conduct.

Nothing here is set in stone and can't be changed. Feel free to suggest improvements or changes in the workflow.

Contribution links

- *Contribution Guide*
  - *Feature requests and feedback*
  - *Report bugs*
  - *Fix bugs*
  - *Write documentation*

– *Preparing Pull Requests*

2.17.1 Feature requests and feedback

We are eager to hear about your requests for new features and any suggestions about the API, infrastructure, and so on. Feel free to submit these as issues with the label "feature request."

Please make sure to explain in detail how the feature should work and keep the scope as narrow as possible. This will make it easier to implement in small PRs.

2.17.2 Report bugs

Report bugs for `climpred` in the issue tracker with the label "bug".

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting, specifically the Python interpreter version, installed libraries, and `climpred` version.
- Detailed steps to reproduce the bug.

If you can write a demonstration test that currently fails but should pass, that is a very useful commit to make as well, even if you cannot fix the bug itself.

2.17.3 Fix bugs

Look through the GitHub issues for bugs.

Talk to developers to find out how you can fix specific bugs.

2.17.4 Write documentation

`climpred` could always use more documentation. What exactly is needed?

- More complementary documentation. Have you perhaps found something unclear?
- Docstrings. There can never be too many of them.
- Example notebooks with different Earth System Models, lead times, etc. – they're all very appreciated.

You can also edit documentation files directly in the GitHub web interface, without using a local copy. This can be convenient for small fixes.

Our documentation is written in reStructuredText. You can follow our conventions in already written documents. Some helpful guides are located here and here.

Note: Build the documentation locally with the following command:

```
$ conda env update -f ci/environment-dev-3.6.yml
$ cd docs
$ make html
```

The built documentation should be available in the `docs/build/`.

If you need to add new functions to the API, run `sphinx-autogen -o api api.rst` from the `docs/source` directory and add the functions to `api.rst`.

2.17.5 Preparing Pull Requests

1. Fork the climpred GitHub repository. It's fine to use `climpred` as your fork repository name because it will live under your user.
2. Clone your fork locally using git, connect your repository to the upstream (main project), and create a branch:

```
$ git clone git@github.com:YOUR_GITHUB_USERNAME/climpred.git
$ cd climpred
$ git remote add upstream git@github.com:bradyrx/climpred.git

# now, to fix a bug or add feature create your own branch off "master":

$ git checkout -b your-bugfix-feature-branch-name master
```

   If you need some help with Git, follow this quick start guide: https://git.wiki.kernel.org/index.php/QuickStart

3. Install dependencies into a new conda environment:

```
$ conda env update -f ci/environment-dev-3.7.yml
$ conda activate climpred-dev
```

4. Make an editable install of climpred by running:

```
$ pip install -e .
```

5. Install pre-commit and its hook on the `climpred` repo:

```
$ pip install --user pre-commit
$ pre-commit install
```

   Afterwards `pre-commit` will run whenever you commit.

   https://pre-commit.com/ is a framework for managing and maintaining multi-language pre-commit hooks to ensure code-style and code formatting is consistent.

   > Now you have an environment called `climpred-dev` that you can work in. You'll need to make sure to activate that environment next time you want to use it after closing the terminal or your system.
   >
   > You can now edit your local working copy and run/add tests as necessary. Please follow PEP-8 for naming. When committing, `pre-commit` will modify the files as needed, or will generally be quite clear about what you need to do to pass the commit test.

6. Break your edits up into reasonably sized commits:

```
$ git commit -a -m "<commit message>"
$ git push -u
```

7. Run all the tests

   Now running tests is as simple as issuing this command:

```
$ pytest climpred
```

Check that your contribution is covered by tests and therefore increases the overall test coverage:

```
$ coverage run --source climpred -m py.test
$ coverage report
$ coveralls
```

Please stick to xarray's testing recommendations.

1. Running the performance test suite

Performance matters and it is worth considering whether your code has introduced performance regressions. *climpred* is starting to write a suite of benchmarking tests using asv to enable easy monitoring of the performance of critical *climpred* operations. These benchmarks are all found in the `asv_bench` directory.

If you need to run a benchmark, change your directory to `asv_bench/` and run:

```
$ asv continuous -f 1.1 upstream/master HEAD
```

You can replace `HEAD` with the name of the branch you are working on, and report benchmarks that changed by more than 10%. The command uses `conda` by default for creating the benchmark environments.

Running the full benchmark suite can take up to half an hour and use up a few GBs of RAM. Usually it is sufficient to paste only a subset of the results into the pull request to show that the committed changes do not cause unexpected performance regressions. You can run specific benchmarks using the `-b` flag, which takes a regular expression. For example, this will only run tests from a `asv_bench/benchmarks/benchmarks_perfect_model.py` file:

```
$ asv continuous -f 1.1 upstream/master HEAD -b ^benchmarks_perfect_model
```

If you want to only run a specific group of tests from a file, you can do it using `.` as a separator. For example:

```
$ asv continuous -f 1.1 upstream/master HEAD -b benchmarks_perfect_model.Compute.time_
↪bootstrap_perfect_model
```

will only run the `time_bootstrap_perfect_model` benchmark of class `Compute` defined in `benchmarks_perfect_model.py`.

1. Create a new changelog entry in `CHANGELOG.rst`:

   - The entry should be entered as:

     <description> (:pr:`#<pull request number>`) `<author's names>`_

     where `<description>` is the description of the PR related to the change and `<pull request number>` is the pull request number and `<author's names>` are your first and last names.

   - Add yourself to list of authors at the end of `CHANGELOG.rst` file if not there yet, in alphabetical order.

1. Add yourself to the contributors list via `docs/source/contributors.rst`.

1. Finally, submit a pull request through the GitHub website using this data:

```
head-fork: YOUR_GITHUB_USERNAME/climpred
compare: your-branch-name

base-fork: bradyrx/climpred
base: master
```

Note that you can create the Pull Request while you're working on this. The PR will update as you add more commits. `climpred` developers and contributors can then review your code and offer suggestions.

2.18 Code of Conduct

2.18.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

2.18.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

2.18.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

2.18.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

2.18.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

2.18.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, and xgcm. For answers to common questions about this code of conduct, see Contributor Covenant.

2.19 Release Procedure

We follow semantic versioning, e.g., v1.0.0. A major version causes incompatible API changes, a minor version adds functionality, and a patch covers bug fixes.

1. Create a new branch `release-vX.x.x` with the version for the release.

- Update *CHANGELOG.rst*
- Make sure all new changes, features are reflected in the documentation.

1. Open a new pull request for this branch targeting *master*
2. After all tests pass and the PR has been approved, merge the PR into `master`
3. Tag a release and push to github:

```
$ git tag -a v1.0.0 -m "Version 1.0.0"
$ git push origin master --tags
```

4. Build and publish release on PyPI:

```
$ git clean -xfd  # remove any files not checked into git
$ python setup.py sdist bdist_wheel --universal  # build package
$ twine upload dist/*  # register and push to pypi
```

5. Update the stable branch (used by ReadTheDocs):

```
$ git checkout stable
$ git rebase master
$ git push -f origin stable
$ git checkout master
```

6. Go to https://readthedocs.org and add the new version to "Active Versions" under the version tab. Force-build "stable" if it isn't already building.
7. Update climpred conda-forge feedstock
    - Fork climpred-feedstock repository
    - Clone this fork and edit recipe:

```
$ git clone git@github.com:username/climpred-feedstock.git
$ cd climpred-feedstock
$ cd recipe
$ # edit meta.yaml
```

- Update version
- Get sha256 from pypi.org for climpred
- Check that `requirements.txt` from the main `climpred` repo is accounted for in `meta.yaml` from the feedstock.
- Fill in the rest of information as described here
- Commit and submit a PR

2.20 Contributors

2.20.1 Core Developers

- Riley X. Brady (github)
- Aaron Spring (github)

2.20.2 Contributors

- Andrew Huang (github)
- Kathy Pegion (github)

For a list of all the contributions, see the github contribution graph.

Bibliography

[Jolliffe2011] Ian T. Jolliffe and David B. Stephenson. Forecast Verification: A Practitioner's Guide in Atmospheric Science. John Wiley & Sons, Ltd, Chichester, UK, December 2011. ISBN 978-1-119-96000-3 978-0-470-66071-3. URL: http://doi.wiley.com/10.1002/9781119960003.

[Murphy1988] Allan H. Murphy. Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient. Monthly Weather Review, 116(12):2417–2424, December 1988. https://doi.org/10/fc7mxd.

[Boer2016] Boer, G. J., D. M. Smith, C. Cassou, F. Doblas-Reyes, G. Danabasoglu, B. Kirtman, Y. Kushnir, et al. "The Decadal Climate Prediction Project (DCPP) Contribution to CMIP6." Geosci. Model Dev. 9, no. 10 (October 25, 2016): 3751–77. https://doi.org/10/f89qdf.

[Bushuk2018] Mitchell Bushuk, Rym Msadek, Michael Winton, Gabriel Vecchi, Xiaosong Yang, Anthony Rosati, and Rich Gudgel. Regional Arctic sea–ice prediction: potential versus operational seasonal forecast skill. Climate Dynamics, June 2018. https://doi.org/10/gd7hfq.

[Griffies1997] S. M. Griffies and K. Bryan. A predictability study of simulated North Atlantic multidecadal variability. Climate Dynamics, 13(7-8):459–487, August 1997. https://doi.org/10/ch4kc4.

[Seferian2018] Roland Séférian, Sarah Berthet, and Matthieu Chevallier. Assessing the Decadal Predictability of Land and Ocean Carbon Uptake. Geophysical Research Letters, March 2018. https://doi.org/10/gdb424.

[Goddard2013] Goddard, L., A. Kumar, A. Solomon, D. Smith, G. Boer, P. Gonzalez, V. Kharin, et al. "A Verification Framework for Interannual-to-Decadal Predictions Experiments." Climate Dynamics 40, no. 1–2 (January 1, 2013): 245–72. https://doi.org/10/f4jjvf.

[Wilks2016] Wilks, D. S. "'The Stippling Shows Statistically Significant Grid Points': How Research Results Are Routinely Overstated and Overinterpreted, and What to Do about It." Bulletin of the American Meteorological Society 97, no. 12 (March 9, 2016): 2263–73. https://doi.org/10/f9mvth.

[Griffies1997] Griffies, S. M., and K. Bryan. "A Predictability Study of Simulated North Atlantic Multidecadal Variability." Climate Dynamics 13, no. 7–8 (August 1, 1997): 459–87. https://doi.org/10/ch4kc4

[Boer2016] Boer, G. J., Smith, D. M., Cassou, C., Doblas-Reyes, F., Danabasoglu, G., Kirtman, B., Kushnir, Y., Kimoto, M., Meehl, G. A., Msadek, R., Mueller, W. A., Taylor, K. E., Zwiers, F., Rixen, M., Ruprich-Robert, Y., and Eade, R.: The Decadal Climate Prediction Project (DCPP) contribution to CMIP6, Geosci. Model Dev., 9, 3751-3777, https://doi.org/10.5194/gmd-9-3751-2016, 2016.

[Jolliffe2011] Ian T. Jolliffe and David B. Stephenson. Forecast Verification: A Practitioner's Guide in Atmospheric Science. John Wiley & Sons, Ltd, Chichester, UK, December 2011. ISBN 978-1-119-96000-3 978-0-470-66071-3. URL: http://doi.wiley.com/10.1002/9781119960003.

[Meehl2013] Meehl, G. A., Goddard, L., Boer, G., Burgman, R., Branstator, G., Cassou, C., … & Karspeck, A. (2014). Decadal climate prediction: an update from the trenches. Bulletin of the American Meteorological Society, 95(2), 243-267. https://doi.org/10.1175/BAMS-D-12-00241.1.

[Murphy1985] Murphy, Allan H., and Daan, H. "Forecast evaluation." Probability, Statistics, and Decision Making in the Atmospheric Sciences, A. H. Murphy and R. W. Katz, Eds., Westview Press, 379-437. https://doi.org.

[Murphy1988] Murphy, Allan H. "Skill Scores Based on the Mean Square Error and Their Relationships to the Correlation Coefficient." Monthly Weather Review 116, no. 12 (December 1, 1988): 2417–24. https://doi.org/10/fc7mxd.

[Pegion2017] Pegion, K., T. Delsole, E. Becker, and T. Cicerone (2019). "Assessing the Fidelity of Predictability Estimates.", Climate Dynamics, 53, 7251–7265 https://doi.org/10.1007/s00382-017-3903-7.

[Jolliffe2012] Jolliffe, Ian T., and David B. Stephenson, eds. Forecast verification: a practitioner's guide in atmospheric science. John Wiley & Sons, 2012.

[Yuan2016] Yuan, Xiaojun, et al. "Arctic sea ice seasonal prediction by a linear Markov model." Journal of Climate 29.22 (2016): 8151-8173.

Index

Symbols

__init__() (*climpred.classes.HindcastEnsemble method*), 115, 117
__init__() (*climpred.classes.PerfectModelEnsemble method*), 119, 120
__init__() (*climpred.comparisons.Comparison method*), 134
__init__() (*climpred.metrics.Metric method*), 133
_bias_slope() (*in module climpred.metrics*), 102
_brier_score() (*in module climpred.metrics*), 107
_conditional_bias() (*in module climpred.metrics*), 100
_crps() (*in module climpred.metrics*), 103
_crpss() (*in module climpred.metrics*), 104
_crpss_es() (*in module climpred.metrics*), 106
_effective_sample_size() (*in module climpred.metrics*), 84
_get_norm_factor() (*in module climpred.metrics*), 133
_mae() (*in module climpred.metrics*), 91
_mape() (*in module climpred.metrics*), 97
_median_absolute_error() (*in module climpred.metrics*), 92
_mse() (*in module climpred.metrics*), 89
_msess() (*in module climpred.metrics*), 96
_msess_murphy() (*in module climpred.metrics*), 102
_nmae() (*in module climpred.metrics*), 93
_nmse() (*in module climpred.metrics*), 92
_nrmse() (*in module climpred.metrics*), 95
_pearson_r() (*in module climpred.metrics*), 82
_pearson_r_eff_p_value() (*in module climpred.metrics*), 85
_pearson_r_p_value() (*in module climpred.metrics*), 83
_rmse() (*in module climpred.metrics*), 90
_smape() (*in module climpred.metrics*), 98
_spearman_r() (*in module climpred.metrics*), 86
_spearman_r_eff_p_value() (*in module climpred.metrics*), 88
_spearman_r_p_value() (*in module climpred.metrics*), 87
_std_ratio() (*in module climpred.metrics*), 100
_threshold_brier_score() (*in module climpred.metrics*), 108
_uacc() (*in module climpred.metrics*), 98
_unconditional_bias() (*in module climpred.metrics*), 101

A

add_control() (*climpred.classes.PerfectModelEnsemble method*), 120
add_observations() (*climpred.classes.HindcastEnsemble method*), 117
add_uninitialized() (*climpred.classes.HindcastEnsemble method*), 117
autocorr() (*in module climpred.stats*), 135

B

bootstrap() (*climpred.classes.PerfectModelEnsemble method*), 121
bootstrap_compute() (*in module climpred.bootstrap*), 123
bootstrap_hindcast() (*in module climpred.bootstrap*), 125
bootstrap_perfect_model() (*in module climpred.bootstrap*), 127
bootstrap_uninit_pm_ensemble_from_control_cftime() (*in module climpred.bootstrap*), 128
bootstrap_uninitialized_ensemble() (*in module climpred.bootstrap*), 129

C

Comparison (*class in climpred.comparisons*), 134
compute_hindcast() (*in module climpred.prediction*), 130
compute_metric() (*climpred.classes.PerfectModelEnsemble method*), 122

compute_perfect_model() (*in module climpred.prediction*), 130
compute_persistence() (*climpred.classes.PerfectModelEnsemble method*), 122
compute_persistence() (*in module climpred.reference*), 131
compute_uninitialized() (*climpred.classes.PerfectModelEnsemble method*), 122
compute_uninitialized() (*in module climpred.reference*), 132
control (*climpred.classes.PerfectModelEnsemble attribute*), 119, 120
corr() (*in module climpred.stats*), 135

D

decorrelation_time() (*in module climpred.stats*), 136
dpp() (*in module climpred.stats*), 136
dpp_threshold() (*in module climpred.bootstrap*), 129

G

generate_uninitialized() (*climpred.classes.PerfectModelEnsemble method*), 123
get_control() (*climpred.classes.PerfectModelEnsemble method*), 121
get_initialized() (*climpred.classes.HindcastEnsemble method*), 117
get_initialized() (*climpred.classes.PerfectModelEnsemble method*), 120
get_observations() (*climpred.classes.HindcastEnsemble method*), 117
get_path() (*in module climpred.preprocessing.mpi*), 140
get_uninitialized() (*climpred.classes.HindcastEnsemble method*), 117
get_uninitialized() (*climpred.classes.PerfectModelEnsemble method*), 121

H

HindcastEnsemble (*class in climpred.classes*), 115

L

load_dataset() (*in module climpred.tutorial*), 138
load_hindcast() (*in module climpred.preprocessing.shared*), 139

M

Metric (*class in climpred.metrics*), 133

O

observations (*climpred.classes.HindcastEnsemble attribute*), 116, 117

P

PerfectModelEnsemble (*class in climpred.classes*), 119

R

rename_SLM_to_climpred_dims() (*in module climpred.preprocessing.shared*), 140
rename_to_climpred_dims() (*in module climpred.preprocessing.shared*), 140
rm_poly() (*in module climpred.stats*), 137
rm_trend() (*in module climpred.stats*), 137

S

smooth() (*climpred.classes.HindcastEnsemble method*), 118
smooth_kws (*climpred.classes.HindcastEnsemble attribute*), 118

U

uninitialized (*climpred.classes.HindcastEnsemble attribute*), 116
uninitialized (*climpred.classes.PerfectModelEnsemble attribute*), 119
uninitialized (*in module climpred.classes*), 117, 120

V

varweighted_mean_period() (*in module climpred.stats*), 138
varweighted_mean_period_threshold() (*in module climpred.bootstrap*), 129
verify() (*climpred.classes.HindcastEnsemble method*), 118