wass2s: A python-based tool for seasonal climate forecast in West Africa and the Sahel.

Release 0.1.1

Mandela C. M. HOUNGNIBO

Jun 25, 2025

CONTENTS:

A python-based tool for seasonal climate forecast in West Africa and the Sahel.

The wass2s tool is designed to facilitate implementation of the new generation of seasonal forecasts in West Africa and the Sahel using various statistical and machine learning methods. New generation of seasonal forecasts aligns with the World Meteorological Organization's (WMO) guidelines for objective, operational, and scientifically rigorous seasonal forecasting methods. wass2s helps forecaster to download GCM, reanalysis, and satellite/observation data, build statistical or machine learning models, verify the models using cross-validation, and forecast. A user-friendly jupyter-lab notebook streamlines the forecasting process .

Features

- Automated Forecasting
- Reproducibility
- Modularity
- Exploration of Machine Learning Models.

CHAPTER

ONE

INSTALLATION

1. Create an environment and activate it

- For Windows: download yaml here and run

```
conda env create -f WAS_S2S_windows.yml
conda activate WASS2S
```

- For Linux: download yaml here and run

```
conda env create -f WAS_S2S_linux.yml
conda activate WASS2S
```

2. Install the wass2s package

```
pip install wass2s
```

3. Upgrade the wass2s package

```
pip install --upgrade wass2s
```

CHAPTER TWO

USAGE

Comprehensive usage guidelines, including data download, processing, models description, configuration and execution, cross-validation, and verification, are available in the Training Documentation. But for a quick start, use the example notebooks.

Download example notebooks:

```
git clone https://github.com/hmandela/WASS2S_notebooks.git
```

or download the zip file:

```
wget https://github.com/hmandela/WASS2S_notebooks/archive/refs/heads/main.zip -O WASS2S_
↪notebooks.zip
```

2.1 Download module

Three types of data can be downloaded with wass2s:

- GCM data on seasonal time scales
- Reanalysis data
- Observational data (satellite data, products combining satellite and observational data)

For some data, for instance C3S, it requires creating an account, accepting the terms of use, and configuring an API key (CDS API key). Please refer also to the CDS documentation for more instructions on how to set up the API key. For more information on C3S seasonal data, browse the MetaData.

2.1.1 Download GCM data

The WAS_Download_Models method allows downloading seasonal forecast model data from various centers for specified variables, initialization months, lead times, and years.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- center_variable (list): List of center-variable identifiers, e.g., ["ECMWF_51.PRCP", "UKMO_604.TEMP"].
- month_of_initialization (int): Initialization month as an integer (1-12).
- lead_time (list): List of lead times in months.
- year_start_hindcast (int): Start year for hindcast data.
- year_end_hindcast (int): End year for hindcast data.
- area (list): Bounding box as [North, West, South, East] for clipping.

- year_forecast (int, optional): Forecast year if downloading forecast data. Defaults to None.
- ensemble_mean (str, optional): Can be "median", "mean", or None. Defaults to None.
- force_download (bool): If True, forces download even if file exists.

Available centers and variables:

- **Centers:** BOM_2, ECMWF_51, UKMO_604, UKMO_603, METEOFRANCE_8, METEOFRANCE_9, DWD_21, DWD_22, CMCC_35, NCEP_2, JMA_3, ECCC_4, ECCC_5, CFSV2_1, CMC1_1, CMC2_1, GFDL_1, NASA_1, NCAR_CCSM4_1, NMME_1
- **Variables:** PRCP, TEMP, TMAX, TMIN, UGRD10, VGRD10, SST, SLP, DSWR, DLWR, HUSS_1000, HUSS_925, HUSS_850, UGRD_1000, UGRD_925, UGRD_850, VGRD_1000, VGRD_925, VGRD_850

Note: Some models are part of the NMME (North American Multi-Model Ensemble) project. For more information, see the NMME documentation. If year_forecast is not specified, hindcast data is downloaded; otherwise, forecast data for the specified year is retrieved.

Example:

```
from wass2s import *

downloader = WAS_Download()

downloader.WAS_Download_Models(
    dir_to_save="/path/to/save",
    center_variable=["ECMWF_51.PRCP"],
    month_of_initialization="03",
    lead_time=["01", "02", "03"],
    year_start_hindcast=1993,
    year_end_hindcast=2016,
    area=[60, -180, -60, 180],
    force_download=False
)
```

2.1.2 Download daily GCM data

The WAS_Download_Models_Daily method allows downloading daily or sub-daily seasonal forecast model data from various centers for specified variables, initialization dates, lead times, and years.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- center_variable (list): List of center-variable identifiers, e.g., ["ECMWF_51.PRCP", "UKMO_604.TEMP"].
- month_of_initialization (int): Initialization month as an integer (1-12).
- day_of_initialization (int): Initialization day as an integer (1-31).
- leadtime_hour (list): List of lead times in hours, e.g., ["24", "48", …, "5160"].
- year_start_hindcast (int): Start year for hindcast data.
- year_end_hindcast (int): End year for hindcast data.
- area (list): Bounding box as [North, West, South, East] for clipping.
- year_forecast (int, optional): Forecast year if downloading forecast data. Defaults to None.
- ensemble_mean (str, optional): Can be "mean", "median", or None. Defaults to None.

- force_download (bool): If True, forces download even if file exists.

Available centers and variables:

- **Centers:** ECMWF_51, UKMO_604, UKMO_603, METEOFRANCE_8, DWD_21, DWD_22, CMCC_35, NCEP_2, JMA_3, ECCC_4, ECCC_5
- **Variables:** PRCP, TEMP, TMAX, TMIN, UGRD10, VGRD10, SST, SLP, DSWR, DLWR, HUSS_1000, HUSS_925, HUSS_850, UGRD_1000, UGRD_925, UGRD_850, VGRD_1000, VGRD_925, VGRD_850

Example:

```
from wass2s import *

downloader = WAS_Download()
downloader.WAS_Download_Models_Daily(
    dir_to_save="/path/to/save",
    center_variable=["ECMWF_51.PRCP"],
    month_of_initialization="01",
    day_of_initialization="01",
    leadtime_hour=["24", "48", "72"],
    year_start_hindcast=1993,
    year_end_hindcast=2016,
    area=[60, -180, -60, 180],
    force_download=False
)
```

2.1.3 Download reanalysis data

The WAS_Download_Reanalysis method downloads reanalysis data for specified center-variable combinations, years, and months, handling cross-year seasons.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- center_variable (list): List of center-variable identifiers, e.g., [“ERA5.PRCP”, “MERRA2.TEMP”].
- year_start (int): Start year for the data to download.
- year_end (int): End year for the data to download.
- area (list): Bounding box as [North, West, South, East] for clipping.
- seas (list): List of month strings representing the season, e.g., [“11”, “12”, “01”] for NDJ.
- force_download (bool): If True, forces download even if file exists.
- run_avg (int): Number of months for running average (default=3).

Available centers and variables:

- **Centers:** ERA5, MERRA2, NOAA (for SST)
- **Variables:** PRCP, TEMP, TMAX, TMIN, UGRD10, VGRD10, SST, SLP, DSWR, DLWR, HUSS_1000, HUSS_925, HUSS_850, UGRD_1000, UGRD_925, UGRD_850, VGRD_1000, VGRD_925, VGRD_850

Example:

```
from wass2s import *

downloader = WAS_Download()
```

(continues on next page)

(continued from previous page)

```
downloader.WAS_Download_Reanalysis(
    dir_to_save="/path/to/save",
    center_variable=["ERA5.PRCP"],
    year_start=1993,
    year_end=2016,
    area=[60, -180, -60, 180],
    seas=["11", "12", "01"],
    force_download=False
)
```

2.1.4 Download observational data

Observational data includes agro-meteorological indicators and satellite-based precipitation data like CHIRPS.

Agro-meteorological indicators

The WAS_Download_AgroIndicators method downloads agro-meteorological indicators for specified variables, years, and months, handling cross-year seasons.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- variables (list): List of shorthand variables, e.g., [“AGRO.PRCP”, “AGRO.TMAX”].
- year_start (int): Start year for the data to download.
- year_end (int): End year for the data to download.
- area (list): Bounding box as [North, West, South, East] for clipping.
- seas (list): List of month strings representing the season, e.g., [“11”, “12”, “01”] for NDJ.
- force_download (bool): If True, forces download even if file exists.

Available variables:

- AGRO.PRCP: precipitation_flux
- AGRO.TMAX: 2m_temperature (24_hour_maximum)
- AGRO.TEMP: 2m_temperature (24_hour_mean)
- AGRO.TMIN: 2m_temperature (24_hour_minimum)

Example:

```
from wass2s import *

downloader = WAS_Download()
downloader.WAS_Download_AgroIndicators(
    dir_to_save="/path/to/save",
    variables=["AGRO.PRCP"],
    year_start=1993,
    year_end=2016,
    area=[60, -180, -60, 180],
    seas=["11", "12", "01"],
    force_download=False
)
```

Download daily agro-meteorological indicators

The WAS_Download_AgroIndicators_daily method downloads daily agro-meteorological indicators for specified variables and years.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- variables (list): List of shorthand variables, e.g., ["AGRO.PRCP", "AGRO.TMAX"].
- year_start (int): Start year for the data to download.
- year_end (int): End year for the data to download.
- area (list): Bounding box as [North, West, South, East] for clipping.
- force_download (bool): If True, forces download even if file exists.

Available variables:

- AGRO.PRCP: precipitation_flux
- AGRO.TMAX: 2m_temperature (24_hour_maximum)
- AGRO.TEMP: 2m_temperature (24_hour_mean)
- AGRO.TMIN: 2m_temperature (24_hour_minimum)

Example:

```
from wass2s import *

downloader = WAS_Download()
downloader.WAS_Download_AgroIndicators_daily(
    dir_to_save="/path/to/save",
    variables=["AGRO.PRCP"],
    year_start=1993,
    year_end=2016,
    area=[60, -180, -60, 180],
    force_download=False
)
```

CHIRPS precipitation data

The WAS_Download_CHIRPSv3 method downloads CHIRPS v3.0 monthly precipitation data for a specified cross-year season.

Parameters:

- dir_to_save (str): Directory to save the downloaded files.
- variables (list): List of variables, typically ["PRCP"].
- year_start (int): Start year for the data to download.
- year_end (int): End year for the data to download.
- area (list, optional): Bounding box as [North, West, South, East] for clipping.
- season_months (list): List of month strings representing the season, e.g., ["03", "04", "05"] for MAM.
- force_download (bool): If True, forces download even if file exists.

Note: CHIRPS data is available for land areas between 50°S and 50°N.

Example:

```
from wass2s import *

downloader = WAS_Download()
downloader.WAS_Download_CHIRPSv3(
    dir_to_save="/path/to/save",
    variables=["PRCP"],
    year_start=1993,
    year_end=2016,
    area=[15, -20, -5, 20],  # Example for Africa
    season_months=["03", "04", "05"],
    force_download=False
)
```

2.2 Processing Modules

The Processing modules provide tools for computing various climate indices or predictands from daily data, such as onset and cessation of the rainy season, dry and wet spells, number of rainy days, extreme precipitation indices, and heat wave indices. Additionally, it offers methods for merging or adjusting gridded data with station observations to correct biases.

These modules are divided into two main parts:

1. **Computing Predictands**: Classes for calculating different climate indices from daily data.
2. **Merging and Adjusting Data**: Classes for combining gridded data with station observations to improve accuracy.

Prerequisites

- **Dask**: Required for parallel processing in gridded data computations.
- **Data Formats**: Gridded data should be in xarray DataArray format with coordinates (T, Y, X). Station data should be in CDT format for daily data or CPT format for seasonal aggregation before merging.

Climate Data Tools (CDT): Format for daily data.

ID	ALLADA	APLAHOUE
LON	2.133333	1.666667
LAT	6.65	6.916667
DAILY/ELEV	92.0	153.0
19810101	0.0	0.0
19810102	0.0	0.0
19810103	0.0	0.0
19810104	0.0	0.0
19810105	0.0	0.0
19810106	0.0	0.0
19810107	0.0	0.0
19810108	0.0	0.0
19810109	0.0	0.0
19810110	0.0	0.0
…		

Climate Prediction Tools (CPT): Format for seasonal aggregation (used in climate prediction tools) before merging.

STATION	ABEO	ABUJ	ADEK
LAT	7.2	7.6	9.0
LON	3.3	5.2	7.2
1991	514.9	715.1	934.3
1992	503.6	736.4	714.6
1993	414.6	891.0	709.6
1994	345.6	1034.7	491.7
1995	492.2	837.6	938.8
…			

2.2.1 Computing Predictands

This section includes classes for computing various climate indices:

- WAS_compute_onset: Computes the onset of the rainy season.
- WAS_compute_cessation: Computes the cessation of the rainy season.
- WAS_compute_onset_dry_spell: Computes the longest dry spell after the onset.
- WAS_compute_cessation_dry_spell: Computes the longest dry spell in flourishing period.
- WAS_count_wet_spells: Computes the number of wet spells between onset and cessation.
- WAS_count_dry_spells: Computes the number of dry spells between onset and cessation.
- WAS_count_rainy_days: Computes the number of rainy days between onset and cessation.
- WAS_r95_99p: Computes extreme precipitation indices R95p and R99p.
- WAS_compute_HWSDI: Computes the Heat Wave Severity Duration Index.

Each class has methods for computing the index from gridded data (compute) and, where applicable, from station data in CDT format (compute_insitu).

Onset Computation

The WAS_compute_onset class computes the onset of the rainy season based on user-defined or default criteria for different zones.

Initialization

- __init__(self, user_criteria=None): Initializes the class with user-defined criteria. If not provided, default criteria are used.
- Dictionaries onset_criteria, cessation_criteria, onset_dryspell_criteria, cessation_dryspell_criteria show how to define the criteria for onset, cessation, onset dry spell and cessation dry spell computations.

Methods

- compute(self, daily_data, nb_cores): Computes onset dates for gridded daily rainfall data. * daily_data: xarray DataArray with daily rainfall data (coords: T, Y, X). * nb_cores: Number of CPU cores for parallel processing. * Returns: xarray DataArray with onset dates.
- compute_insitu(self, daily_df): Computes onset dates for station data in CDT format. * daily_df: pandas DataFrame in CDT format. * Returns: pandas DataFrame in CPT format with onset dates.

Criteria Dictionary

The criteria dictionary defines parameters for onset computation:

```
{
    0: {"zone_name": "Sahel100_0mm", "start_search": "06-01", "cumulative": 10, "number_dry_days
↪": 25, "thrd_rain_day": 0.85, "end_search": "08-30"},
    1: {"zone_name": "Sahel200_100mm", "start_search": "05-15", "cumulative": 15, "number_dry_days
↪": 25, "thrd_rain_day": 0.85, "end_search": "08-15"},
    ...
}
```

- zone_name: Name of the zone.
- start_search: Start date for searching the onset (e.g., "06-01").
- cumulative: Cumulative rainfall threshold (mm).
- number_dry_days: Maximum number of dry days allowed after onset.
- thrd_rain_day: Rainfall threshold to consider a day as rainy (mm).
- end_search: End date for searching the onset.

Example

```
from wass2s import *
# Download daily rainfall data
downloader = WAS_Download()
downloader.WAS_Download_AgroIndicators_daily(
    dir_to_save="/path/to/save",
    variables=["AGRO.PRCP"],
    year_start=1993,
    year_end=2016,
    area=[60, -180, -60, 180],
    force_download=False
)

# Load daily rainfall data
rainfall = prepare_predictand(dir_to_save, variables, year_start, year_end, daily=True, ds=False)
## NB: prepare_predictand is a utility function that loads the data and prepares it for the computation␣
↪of the predictand.
## ds is set to False because the data will be loaded as dataarray.

# Print predefined  onset criteria
onset_criteria
# Define user criteria
user_criteria = onset_criteria
# adjust user criteria
user_criteria[0]["start_search"] = "06-15"
user_criteria[1]["end_search"] = "09-01"
# Compute onset
was_onset = WAS_compute_onset(user_criteria)
onset = was_onset.compute(daily_data=rainfall, nb_cores=4)
# Plot the mean onset date to check the results
plot_date(onset.mean(dim='T'))
```

Cessation Computation

The WAS_compute_cessation class computes the cessation of the rainy season based on soil moisture balance criteria.

- Similar initialization and methods as WAS_compute_onset with criteria including: * date_dry_soil: Date when soil is assumed dry (e.g., "01-01"). * ETP: Evapotranspiration rate (mm/day). * Cap_ret_maxi: Maximum soil water retention capacity (mm).

Dry Spell Computation

The WAS_compute_onset_dry_spell class computes the longest dry spell after the onset.

- Includes an additional nbjour parameter in the criteria for the number of days to check after onset.

The WAS_compute_cessation_dry_spell class computes the longest dry spell in flourishing period.

- Includes an additional nbjour parameter in the criteria for the number of days to check after cessation.

The WAS_count_dry_spells class computes the number of dry spells between onset and cessation. Requires onset and cessation dates as inputs.

Wet Spell Computation

The WAS_count_wet_spells class computes the number of wet spells between onset and cessation. Requires onset and cessation dates as inputs.

Rainy Days Computation

The WAS_count_rainy_days class computes the number of rainy days between onset and cessation. Requires onset and cessation dates as inputs.

Extreme Precipitation Indices

The WAS_r95_99p class computes R95p and R99p indices. Initialization with a base period (e.g., slice("1991-01-01", "2020-12-31")) and optional season (list of months).

- Methods: * compute_r95p and compute_r99p for gridded data. * compute_insitu_r95p and compute_insitu_r99p for station data.

Heat Wave Indices

The WAS_compute_HWSDI class computes the Heat Wave Severity Duration Index. Computes TXin90 (90th percentile of daily max temperature) and counts heatwave days with at least 6 consecutive hot days.

2.2.2 Merging and Adjusting Data

The WAS_Merging class provides methods for merging gridded data with station observations to adjust for biases.

Initialization

- __init__(self, df, da, date_month_day="08-01"): Initializes with station data DataFrame (CPT format), gridded data DataArray, and a date string.

Methods

- simple_bias_adjustment(self, missing_value=-999.0, do_cross_validation=False): Adjusts gridded data using kriging of residuals.
- regression_kriging(self, missing_value=-999.0, do_cross_validation=False): Uses linear regression followed by kriging of residuals.
- neural_network_kriging(self, missing_value=-999.0, do_cross_validation=False): Uses a neural network followed by kriging of residuals.
- multiplicative_bias(self, missing_value=-999.0, do_cross_validation=False): Applies a multiplicative bias correction.

Each method returns the adjusted gridded data as an xarray DataArray and optionally cross-validation results as a DataFrame.

- plot_merging_comparaison(self, df_Obs, da_estimated, da_corrected, missing_value=-999.0): Visualizes the comparison between observations, original estimates, and corrected data.

Example: Merging Onset with Station Observations

```
# Load station onset data in CPT format
cpt_input_file_path = "./path/to/cpt_file.csv"
df = pd.read_csv(cpt_input_file_path, na_values=-999.0, encoding="latin1")

# Filter for relevant years and stations
year_start, year_end = 1981, 2020  # Example years
onset_df = df[(df['STATION'] == 'LAT') | (df['STATION'] == 'LON') |
             (pd.to_numeric(df['STATION'], errors='coerce').between(year_start, year_end))]

# Verify station network
verify_station_network(onset_df, area)
## NB: verify_station_network is a utility function that verifies the station network. area is the extent␣
↪of the gridded onset domain.

# Instantiate WAS_Merging
data_merger = WAS_Merging(onset_df, onset, date_month_day='02-01')
## NB: date_month_day is set to '02-01' because the onset start_search criteria is set to the month of␣
↪February.
## Important to verify the T dimension in the gridded onset computed. the month and day must match␣
↪the date_month_day.

# Perform simple bias adjustment
onset_adjusted, _ = data_merger.simple_bias_adjustment(do_cross_validation=False)

# Plot comparison
data_merger.plot_merging_comparaison(onset_df, onset, onset_adjusted)
## NB: plot_merging_comparaison is a utility function that plots the comparison between the station␣
↪onset, the gridded onset and the adjusted onset.
```

2.3 Quantifying uncertainty via cross-validation

Cross-validation schemes are used to assess model performance and to quantify uncertainty. *wass2s* uses a cross-validation scheme that splits the data into training, omit, and test periods. The scheme is a variation of the *K-Fold* cross-validation scheme, but it is tailored for time series data throughout *CustomTimeSeriesSplit* and *WAS_Cross_Validator* class. The scheme is illustrated in the figure below (Figure 1).

The figure shows how we split our data (1981–2010) to validate the model. Each row is a "fold" or a test run.

- **Pink (Training)**: Years we use to train the model. For example, in the first row, we train on 1986–2010.
- **Yellow (Omit)**: A buffer years we skip to avoid cheating. Climate data has patterns over time, so we don't want to train on a years right after/before the one we're predicting, which would make the model look better than it really is. In this case we've omitted four years (in the first row, we skip 1982-1985).
- **White (Predict)**: The year we predict. In the first row, we predict 1981.

CustomTimeSeriesSplit

A custom splitter for time series data that accounts for temporal dependencies.

Initialization

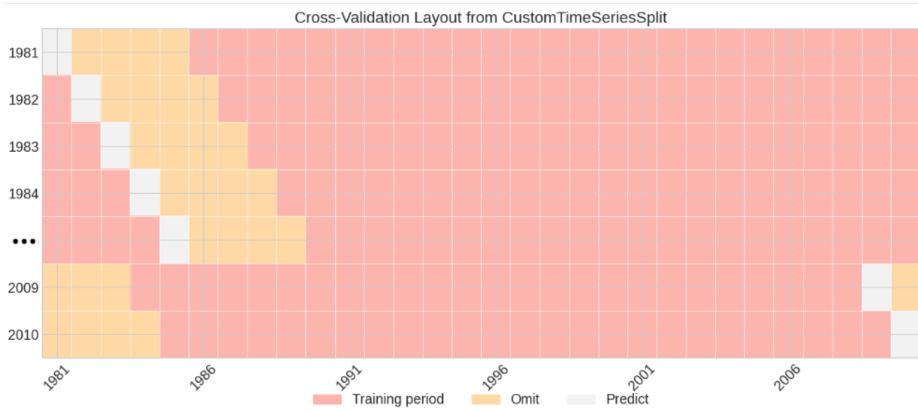

Fig. 1: Cross-validation scheme used in wass2s

- *n_splits*: Number of splits for cross-validation.

Methods

- *split*: Generates indices for training and test sets, omitting a specified number of samples after the test index.
- *get_n_splits*: Returns the number of splits.

WAS_Cross_Validator

A wrapper class that uses the custom splitter to perform cross-validation with various models.

Initialization

- *n_splits*: Number of splits for cross-validation.
- *nb_omit*: Number of samples to omit from training after the test index.

Methods

- *get_model_params*: Retrieves parameters for the model's *compute_model* method.
- *cross_validate*: Performs cross-validation and computes deterministic hindcast and tercile probabilities.

Example Usage

```
from wass2s.was_cross_validate import WAS_Cross_Validator

# Initialize the cross-validator
cv = WAS_Cross_Validator(n_splits=30, nb_omit=4)
```

A better example will be provided in the next sections.

2.3.1 Estimating Prediction Uncertainty

The cross-validation makes out-of-sample predictions for each fold's prediction period, and errors are calculated by comparing predictions to actual values. These errors are collected across all folds. Running the statistical models—e.g. multiple linear regression—yields the most likely value of the predictand (best-guess) for the coming season. Because seasonal outlooks are inherently probabilistic, we must go beyond this single best-guess and quantify the likelihood of other possible outcomes. wass2s does so by analysing the cross-validation errors described earlier. The method explicitly takes the statistical distribution of the predictand into account. If, for instance, the predictand is approximately Gaussian, we assume the predicted values follow a normal distribution whose mean is the single best-guess and whose variance equals the cross-validated error variance. Comparing that forecast probability-density function with the climatological density (see the example in Figure 2) lets us integrate the areas that fall below-normal (values below the 1st tercile), near-normal (values between the 1st and 3rd terciles), and above-normal (values above the 3rd tercile). These integrals are the tercile probabilities ultimately delivered to users.

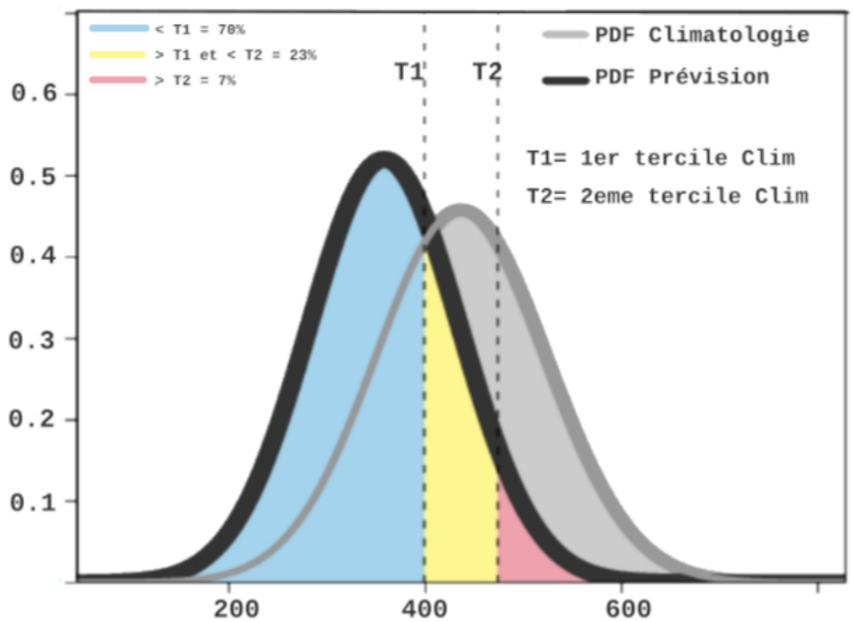

Fig. 2: Figure 2: Generation of probabilistic forecasts

Important

Classification-based statistical models—such as logistic regression, extended logistic regression, and support vector classification—do **not** generate continuous probabilistic forecasts over a full distribution of outcomes as indicated above. Instead, they classify the predictand into discrete categories based on climatological terciles (below-normal, near-normal, above-normal) and estimate the probability associated with each class.

2.4 Models Modules

The Models modules provide a comprehensive suite of statistical and machine learning models for climate prediction, including linear models, EOF-based models, canonical correlation analysis (CCA), analog methods, and multi-model ensemble (MME) techniques. These models are designed to handle both deterministic and probabilistic forecasts, with support for hyperparameter tuning. Models are evaluated using cross-validation schemes.

The models modules are organized into several classes, each implementing a specific type of model:

1. **Machine Learning Models**: This includes linear models such as multiple linear regression, logistic regression and regularized models like ridge, lasso, elastic-net. Additionally, more advanced models are available, including support vector regression, random forests, XGBoost, and neural networks.
2. **EOF and PCR Models**: For dimensionality reduction and regression using principal components.
3. **CCA Models**: For identifying relationships between two multivariate datasets.
4. **Analog Methods**: For finding historical analogs to current conditions.
5. **Multi-Model Ensemble (MME) Techniques**: For combining predictions from multiple models.

2.4.1 Machine Learning Models

The available models are:

- *WAS_LinearRegression_Model*:

Standard Multiple Linear Regression. * *WAS_Ridge_Model*: Ridge regression with L2 regularization. * *WAS_Lasso_Model*: Lasso regression with L1 regularization. * *WAS_LassoLars_Model*: Lasso regression using the LARS algorithm. * *WAS_ElasticNet_Model*: Elastic net regression combining L1 and L2 regularization. * *WAS_LogisticRegression_Model*: Logistic regression for classification. * *WAS_SVR*: Support vector regression. * *WAS_PolynomialRegression*: Polynomial regression. * *WAS_PoissonRegression*: Poisson regression. * *WAS_RandomForest_XGBoost_ML_Stacking*: Random forest and XGBoost regression with stacking. * *WAS_MLP*: Multi-Layer Perceptron regression. * *WAS_RandomForest_XGBoost_Stacking_MLP*: Random forest, XGBoost, and MLP regression with stacking. * *WAS_Stacking_Ridge*: Random forest, XGBoost, MLP, and Ridge regression with stacking.

Except for *WAS_LogisticRegression_Model*, each model class includes methods for:

- *compute_model*: Training the model and making predictions.
- *compute_prob*: Computing tercile probabilities for the predictions.
- *forecast*: Making forecasts for new data.

2.4.2 EOF and PCR Models

The *was_eof.py* and *was_pcr.py* modules provide classes for EOF analysis and Principal Component Regression (PCR), with support for multiple EOF zones:

- *WAS_EOF*: Performs EOF analysis with options for cosine latitude weighting, standardization, and L2 normalization.
- *WAS_PCR*: Combines EOF analysis with a regression model for prediction, supporting multiple EOF zones.

WAS_EOF

Initialization

- *n_modes*: Number of EOF modes to retain.
- *use_coslat*: Apply cosine latitude weighting (default: True).
- *standardize*: Standardize the input data (default: False).
- *opti_explained_variance*: Target cumulative explained variance to determine modes.
- *L2norm*: Normalize components and scores to have L2 norm (default: True).

Methods

- *fit*: Fits the EOF model to the data, supporting multiple zones by applying EOF analysis to the entire dataset.
- *transform*: Projects new data onto the EOF modes.
- *inverse_transform*: Reconstructs data from principal components (PCs).
- *plot_EOF*: Plots the EOF spatial patterns with explained variance.

WAS_PCR

Initialization

- *regression_model*: The regression model (e.g., *WAS_Ridge_Model*) to use with PCs.
- *n_modes*: Number of EOF modes to retain.
- *use_coslat*: Apply cosine latitude weighting (default: True).

- *standardize*: Standardize the input data (default: False).
- *opti_explained_variance*: Target cumulative explained variance.
- *L2norm*: Normalize EOF components and scores (default: True).

Methods

- *compute_model*: Fits the EOF model, transforms data to PCs, and applies the regression model.
- *compute_prob*: Computes tercile probabilities using the regression model.
- *forecast*: Makes forecasts using EOF-transformed data.

Example Usage: Seasonal Forecasting Based on Observational Data

```
from wass2s import *
## Define the directory to save the data
dir_to_save_reanalysis = "/path/to/save_reanalysis"
dir_to_save_agroindicators = "/path/to/save_agroindicators"

## Define the climatology  year range and the season
clim_year_start = 1991
clim_year_end = 2020
seas_reanalysis = ["01", "02", "03"]
seas_agroindicators = ["05", "06", "07"]

## Define the variables to download
variables = ["AGRO.PRCP"]

## Define the center and the predictor variables
center_variable = ["ERA5.SST"]:

## Define the extent for reanalysis
extent = [45, -180, -45, 180] # [North, West, South, East]

## Define the extent for Observation
extent_obs = [30, -25, 0, 30] # [North, West, South, East]

## Download the predictors and the predictand
downloader = WAS_Download()

## Download the predictors
downloader.WAS_Download_Reanalysis(
    dir_to_save=dir_to_save_reanalysis,
    center_variable=center_variable,
    year_start=1991,
    year_end=2025,
    area=extent,
    seas=seas_reanalysis,
    force_download=False
)

## Download the predictand
downloader.WAS_Download_AgroIndicators(
    dir_to_save=dir_to_save_agroindicators,
    variables=["AGRO.PRCP"],
```

(continues on next page)

(continued from previous page)

```
    year_start=1991,
    year_end=2024,
    area=extent_obs,
    seas=seas_agroindicators,
    force_download=False
)
```

Case 1: Used SST index as a predictor

```
# Prepare predictand and predictors
predictand = prepare_predictand(dir_to_save_agroindicators, variables, year_start, year_end, seas_
↪agroindicators, ds=False, daily=False)

# Prepare predictors
## Print available SST indices
print(list(sst_indices_name.keys()))

## Choose yours
sst_index_name = ['NINO34','TNA', 'TSA', 'DMI']

## Plot the SST index zone
plot_map([extent[1],extent[3],extent[2],extent[0]], sst_indices = sst_index_name, title="Index Zone",fig_
↪size=(7,4))

## Compute the SST indices
predictors = compute_sst_indices(dir_to_save_reanalysis, sst_index_name, center_variable[0], year_
↪start, year_end, seas_reanalysis)

## Compute variance inflation factor to see multicolinearity between predictors

vif_data = pd.DataFrame()
vif_data["feature"] = predictors.to_dataframe().columns
vif_data["VIF"] = [VIF(predictors.to_dataframe(), i) for i in range(predictors.to_dataframe().shape[1])]
## Print VIF values
print(vif_data)

## Set a threshold for VIF
vif_threshold = 5
# Remove features with VIF greater than the threshold
low_vif_predictors = vif_data[vif_data["VIF"] < vif_threshold]["feature"].tolist()
filtered_predictors = predictors[low_vif_predictors].to_array()
filtered_predictors = filtered_predictors.rename({"variable": "features"}).transpose('T', 'features')

# Initialize the model class
model = WAS_LinearRegression_Model(nb_cores=2, dist_method="lognormal")
# Assuming predictand follows a lognormal distribution. otherwise, normal, student-t or gamma are␣
↪available. used dist_method="normal" or dist_method="t" or dist_method="gamma".

# Perform cross-validation
was_cv = WAS_Cross_Validator(n_splits=len(predictand.get_index("T")), nb_omit=2)
hindcast_det, hindcast_prob = was_cv.cross_validate(model, predictand, filtered_
↪predictorsisel(T=slice(None,-1)), clim_year_start, clim_year_end)
```

(continues on next page)

(continued from previous page)

```
# clim_year_start and clim_year_end are the years used to compute the climatology.

# Initialize the model class
model = WAS_Ridge_Model(n_clusters=6, alpha_range=np.logspace(-4, 0.1, 20), nb_cores = 2)

# Compute alpha parameters
alpha, clusters = model.compute_hyperparameters(predictand, filtered_predictors)

# Perform cross-validation
was_cv = WAS_Cross_Validator(n_splits=len(predictand.get_index("T")), nb_omit=2)
hindcast_det_Ridge, hindcast_prob_Ridge = was_cv.cross_validate(model, predictand, filtered_
↪predictors.isel(T=slice(None,-1)), clim_year_start, clim_year_end, alpha=alpha)

# Make a forecast
forecast_det_Ridge, forecast_prob_Ridge = model.forecast(predictand, clim_year_start, clim_year_
↪end, filtered_predictors.isel(T=slice(None,-1)), hindcast_det_Ridge, filtered_predictors.isel(T=[-1]),
↪alpha=alpha, l1_ratio=l1_ratio)
```

Case 2: Used PCRs as a predictor

```
# Set your own zones ( zones not available in built-in)
# define zone as dict : {'zone_name_key': ('Explicit_Zone_name', lon_min, lon_max, lat_min, lat_
↪max)}
zones_for_PCR = {'A': ('A', -150, 150, -45, 45)}

# Set number of modes
n_modes = 6

# ElasticNet hyperparameters range
alpha_range = np.logspace(-4, 0.1, 20)
l1_ratio_range = [0.5, 0.9999]

# Initialize the model class
model = WAS_PCR_Model(n_clusters=6, alpha_range=np.logspace(-4, 0.1, 20), nb_cores = 2)
plot_map([extent[1],extent[3],extent[2],extent[0]], sst_indices = zones_for_PCR, title="Predictors Area",
↪fig_size=(8,6))

# Retrieve predictor data for the defined zone
predictor = retrieve_single_zone_for_PCR(dir_to_save_Reanalysis, zones_for_PCR, variables_
↪reanalysis[0], year_start, year_end, season, clim_year_start, clim_year_end)

# Load WAS_EOF Class
eof_model = WAS_EOF(n_modes=n_modes, use_coslat=True, standardize=True)

# Load predictor, compute EOFs and retrieve component, scores and explained variances
s_eofs, s_pcs, s_expvar, _ = eof_model.fit(predictor, dim="T",  clim_year_start=clim_year_start,
↪clim_year_end=clim_year_end)

# Plot EOFs and explained variances
eof_model.plot_EOF(s_eofs, s_expvar)

# Perform Cross-validation with elastic-net
```

(continues on next page)

(continued from previous page)

```
## Load class for model
regression_model = WAS_ElasticNet_Model(alpha_range = alpha_range, l1_ratio_range = l1_ratio_
↪range, nb_cores = 2, dist_method="lognormal")
pcr_model = WAS_PCR(regression_model=regression_model, n_modes=n_modes, standardize=False)

## Compute alpha parameters
alpha, l1_ratio, clusters = regression_model.compute_hyperparameters(predictand, s_pcs.
↪isel(T=slice(None,-1)).rename({"mode": "features"}).transpose('T', 'features'))
## Perform cross-validation
was_cv = WAS_Cross_Validator(n_splits=len(predictand.get_index("T")), nb_omit=2)
hindcast_det, hindcast_prob = was_cv.cross_validate(pcr_model, predictand, s_pcs.isel(T=slice(None,-
↪1)).rename({"mode": "features"}).transpose('T', 'features'), clim_year_start, clim_year_end,
↪alpha=alpha, l1_ratio=l1_ratio)
```

2.4.3 CCA Models

The *was_cca.py* module provides classes for Canonical Correlation Analysis (CCA):

- *WAS_CCA*: Performs CCA to identify relationships between two multivariate datasets.

Initialization

- *n_modes*: Number of CCA modes to retain.
- *n_pca_modes*: Number of PCA modes to use for dimensionality reduction.
- *dist_method*: distribution method for probability computations.

Methods

- *compute_model*: Fits the CCA model and makes predictions.
- *compute_prob*: Computes tercile probabilities for the predictions.

Example Usage: Recalibrating Seasonal Forecast Outputs from Global Climate Models (GCMs)

2.4.4 Analog Forecasting Methods

The *was_analog.py* module provides the *WAS_Analog* class for analog-based forecasting using various techniques to identify historical analogs to current conditions for prediction, particularly for seasonal rainfall forecasts using sea surface temperature (SST) data.

Initialization Parameters

- `dir_to_save` (str): Directory path to save downloaded and processed data files.
- `year_start` (int): Starting year for historical data.
- `year_forecast` (int): Target forecast year.
- `reanalysis_name` (str): Reanalysis dataset name (e.g., “ERA5.SST” or “NOAA.SST”).
- `model_name` (str): Forecast model name (e.g., “ECMWF_51.SST”).
- `method_analog` (str, default=”som”): Analog method to use (“som”, “cor_based”, “pca_based”).
- `best_prcp_models` (list, optional): List of best precipitation models. Default is None.
- `month_of_initialization` (int, optional): Forecast initialization month. Default is None (uses current month).
- `lead_time` (list, optional): Lead times in months. Default is None (uses [1, 2, 3, 4, 5]).

- ensemble_mean (str, default="mean"): Ensemble mean method ("mean" or "median").
- clim_year_start (int, optional): Start year for climatology period.
- clim_year_end (int, optional): End year for climatology period.
- define_extent (tuple, optional): Bounding box as (lon_min, lon_max, lat_min, lat_max) for regional analysis.
- index_compute (list, optional): Climate indices to compute (e.g., ["NINO34", "DMI"]).
- some_grid_size (tuple, default=(None, None)): SOM grid dimensions (rows, cols); None uses automatic sizing.
- some_learning_rate (float, default=0.5): Learning rate for SOM training.
- some_neighborhood_function (str, default="gaussian"): Neighborhood function for SOM ("gaussian", etc.).
- some_sigma (float, default=1.0): Initial neighborhood radius for SOM.
- dist_method (str, default="gamma"): Probability method ("gamma", "t", "normal", "lognormal", "non-param").

Key Methods

- download_sst_reanalysis(): Downloads and processes SST reanalysis data from the specified center for the given years and area.
- download_models(): Downloads seasonal forecast model data for the specified model, initialization month, and lead times.
- standardize_timeseries(): Standardizes time series data over a specified climatology period.
- calc_index(): Computes specified climate indices (e.g., NINO34, DMI) from SST data.
- compute_model(): Identifies historical analogs using the specified method and computes deterministic forecasts.
- compute_prob(): Calculates tercile probabilities (Below Normal, Near Normal, Above Normal) using the specified distribution method.
- forecast(): Generates deterministic and probabilistic forecasts for the target year, returning processed SST data, similar years, deterministic forecast, and probabilistic forecast.
- composite_plot(): Creates composite plots of forecast results, optionally including the predictor (SST) visualization.

Example Usage

Basic analog forecast setup:

```python
from wass2s.was_analog import WAS_Analog

# Initialize analog model
analog_model = WAS_Analog(
    dir_to_save="./s2s_data/analog",
    year_start=1990,
    year_forecast=2025,
    reanalysis_name="NOAA.SST",
    model_name="ECMWF_51.SST",
    method_analog="som",
    month_of_initialization=3,
    clim_year_start=1991,
    clim_year_end=2020,
    define_extent=(-180, 180, -45, 45),
```

(continues on next page)

(continued from previous page)

```
    index_compute=["NINO34", "DMI"],
    dist_method="gamma"
)

# Download and process data
sst_hist, sst_for = analog_model.download_and_process()

# Generate forecast
ddd, similar_years, forecast_det, forecast_prob = analog_model.forecast(
    predictant=rainfall_data,
    clim_year_start=1991,
    clim_year_end=2020,
    hindcast_det=hindcast_data,
    forecast_year=2025
)

# Create composite plot
similar_years = analog_model.composite_plot(
    predictant=rainfall_data,
    clim_year_start=1991,
    clim_year_end=2020,
    hindcast_det=hindcast_data,
    plot_predictor=True
)
```

Cross-Validation Example

```
from wass2s.was_analog import WAS_Cross_Validator

# Perform cross-validation
was_analog_cv = WAS_Cross_Validator(n_splits=len(rainfall.get_index("T")), nb_omit=2)
hindcast_analog_det, hindcast_analog_prob = was_analog_cv.cross_validate(
    analog_model,
    rainfall,
    clim_year_start=1991,
    clim_year_end=2020
)

# Generate forecast using cross-validated hindcast
ddd, similar_years, forecast_det, forecast_prob = analog_model.forecast(
    predictant=rainfall,
    clim_year_start=1991,
    clim_year_end=2020,
    hindcast_det=hindcast_analog_det,
    forecast_year=2025
)
```

Note

Ensure *WAS_Cross_Validator* is correctly imported from the *wass2s.was_analog* module and that the *rainfall* variable is an xarray DataArray with appropriate dimensions (T, Y, X).

2.5 Verification Module

The Verification module provides tools for evaluating the performance of climate forecasts using a variety of deterministic, probabilistic, and ensemble-based metrics. It is implemented in the *was_verification.py* module and leverages the *WAS_Verification* class to compute metrics such as Kling-Gupta Efficiency (KGE), Pearson Correlation, Ranked Probability Skill Score (RPSS), and Continuous Ranked Probability Score (CRPS). The module also includes visualization utilities for plotting scores, reliability diagrams, and ROC curves.

This module is designed to work with gridded climate data, typically stored in *xarray* DataArrays, and supports parallel computation using *dask* for efficiency with large datasets.

The *WAS_Verification* class is the core of the Verification module, providing methods to compute and visualize various performance metrics for climate forecasts.

Initialization

```
from wass2s.was_verification import WAS_Verification

# Initialize with a distribution method for probabilistic forecasts
verifier = WAS_Verification(dist_method="gamma")
```

Parameters

- *dist_method*: Specifies the distribution method for computing tercile probabilities. Options include: - *"t"*: Student's t-based method. - *"gamma"*: Gamma distribution-based method (default). - *"normal"*: Normal distribution-based method. - *"lognormal"*: Lognormal distribution-based method. - *"weibull_min"*: Weibull minimum distribution-based method. - *"nonparam"*: Non-parametric method using historical errors.

Available Metrics

The class defines a dictionary of scoring metrics with metadata, including:

- **Deterministic Metrics**: - *KGE*: Kling-Gupta Efficiency (-1 to 1). - *Pearson*: Pearson Correlation Coefficient (-1 to 1). - *IOA*: Index of Agreement (0 to 1). - *MAE*: Mean Absolute Error (0 to 100). - *RMSE*: Root Mean Square Error (0 to 100). - *NSE*: Nash-Sutcliffe Efficiency (None to 1). - *TAYLOR_DIAGRAM*: Taylor Diagram (visualization).
- **Probabilistic Metrics**: - *GROC*: Generalized Receiver Operating Characteristic (0 to 1). - *RPSS*: Ranked Probability Skill Score (-1 to 1). - *IGS*: Ignorance Score (0 to None). - *RES*: Resolution Score (0 to None). - *REL*: Reliability Score (None to None). - *RELIABILITY_DIAGRAM*: Reliability Diagram (visualization). - *ROC_CURVE*: Receiver Operating Characteristic Curve (visualization).
- **Ensemble Metrics**: - *CRPS*: Continuous Ranked Probability Score (0 to 100).

Metadata Access

```
metadata = verifier.get_scores_metadata()
```

This returns a dictionary containing the name, range, type, colormap, and computation function for each metric.

2.5.1 Deterministic Metrics

Deterministic metrics evaluate the performance of point forecasts against observations. They are computed using the *compute_deterministic_score* method, which applies a scoring function over *xarray* DataArrays.

Example Usage

```
# Compute Pearson Correlation
pearson_score = verifier.compute_deterministic_score(
```

(continues on next page)

(continued from previous page)

```
    verifier.pearson_corr, obs_data, model_data
)

# Plot the score
verifier.plot_model_score(pearson_score, "Pearson", dir_save_score="./scores", figure_name=
↪"Pearson_Score")
```

Key Methods

- *kling_gupta_efficiency*: Computes KGE, balancing correlation, bias, and variability.
- *pearson_corr*: Computes Pearson Correlation Coefficient.
- *index_of_agreement*: Computes IOA, measuring agreement between predictions and observations.
- *mean_absolute_error*: Computes MAE, the average absolute difference.
- *root_mean_square_error*: Computes RMSE, the square root of mean squared differences.
- *nash_sutcliffe_efficiency*: Computes NSE, comparing prediction errors to the mean of observations.
- *taylor_diagram*: Placeholder for Taylor Diagram visualization (to be implemented).

Plotting

The *plot_model_score* method visualizes deterministic scores on a map using *cartopy*.

```
verifier.plot_model_score(score_result, "KGE", dir_save_score="./scores", figure_name="KGE_Model
↪")
```

The *plot_models_score* method plots multiple model scores in a grid.

```
model_metrics = {
    "model1": score_result1,
    "model2": score_result2
}
verifier.plot_models_score(model_metrics, "Pearson", dir_save_score="./scores")
```

2.5.2 Probabilistic Metrics

Probabilistic metrics evaluate the performance of forecasts that provide probabilities for tercile categories (below-normal, near-normal, above-normal). These are computed using the *compute_probabilistic_score* method.

Example Usage

```
# Compute tercile probabilities
proba_forecast = verifier.gcm_compute_prob(obs_data, clim_year_start=1981, clim_year_end=2010,␣
↪hindcast_det=model_data)

# Compute RPSS
rpss_score = verifier.compute_probabilistic_score(
    verifier.calculate_rpss, obs_data, proba_forecast, clim_year_start=1981, clim_year_end=2010
)
```

Key Methods

- *classify*: Classifies data into terciles based on climatology.
- *compute_class*: Computes tercile class labels for observations.

- *calculate_groc*: Computes GROC, averaging AUC across tercile categories.
- *calculate_rpss*: Computes RPSS, comparing forecast probabilities to climatology.
- *ignorance_score*: Computes Ignorance Score per Weijs (2010).
- *resolution_score_grid*: Computes Resolution Score, measuring how forecasts differ from climatology.
- *reliability_score_grid*: Computes Reliability Score, assessing forecast calibration.
- *reliability_diagram*: Plots Reliability Diagrams for each tercile category.
- *plot_roc_curves*: Plots ROC Curves with confidence intervals for each tercile.

Visualization

Reliability Diagrams and ROC Curves are generated for probabilistic forecasts.

```
# Plot Reliability Diagram
verifier.reliability_diagram(
    modelname="Model1", dir_to_save_score="./scores", y_true=obs_data, y_probs=proba_forecast,
    clim_year_start=1981, clim_year_end=2010
)

# Plot ROC Curves with 95% confidence intervals
verifier.plot_roc_curves(
    modelname="Model1", dir_to_save_score="./scores", y_true=obs_data, y_probs=proba_forecast,
    clim_year_start=1981, clim_year_end=2010, n_bootstraps=1000, ci=0.95
)
```

2.5.3 Ensemble Metrics

Ensemble metrics evaluate forecasts with multiple members, such as those from GCMs. The primary metric is CRPS, computed using *xskillscore*.

Example Usage

```
# Compute CRPS for ensemble forecast
crps_score = verifier.compute_crps(obs_data, model_data, member_dim='number', dim="T")
```

Key Methods

- *compute_crps*: Computes CRPS for ensemble forecasts, measuring the difference between predicted and observed distributions.

2.5.4 Tercile Probability Computation

The module provides multiple methods to compute tercile probabilities for probabilistic forecasts, based on different distributional assumptions.

Key Methods

- *calculate_tercile_probabilities*: Uses Student's t-distribution.
- *calculate_tercile_probabilities_gamma*: Uses Gamma distribution.
- *calculate_tercile_probabilities_normal*: Uses Normal distribution.
- *calculate_tercile_probabilities_lognormal*: Uses Lognormal distribution.
- *calculate_tercile_probabilities_weibull_min*: Uses Weibull minimum distribution.
- *calculate_tercile_probabilities_nonparametric*: Uses historical errors for a non-parametric approach.

Example Usage

```
# Compute probabilities using Gamma distribution
hindcast_prob = verifier.gcm_compute_prob(
    Predictant=obs_data, clim_year_start=1981, clim_year_end=2010, hindcast_det=model_data
)
```

The *gcm_compute_prob* method selects the appropriate distribution based on the *dist_method* parameter.

2.5.5 GCM Validation

The module includes methods to validate General Circulation Model (GCM) forecasts against observations, supporting both deterministic and probabilistic metrics.

Key Methods

- *gcm_validation_compute*: Validates GCM forecasts for multiple models, computing specified metrics.
- *weighted_gcm_forecasts*: Combines forecasts from multiple models using weights based on a performance metric (e.g., GROC).

Example Usage

```
# Validate GCM forecasts
models_files_path = {
    "model1": "path/to/model1.nc",
    "model2": "path/to/model2.nc"
}
x_metric = verifier.gcm_validation_compute(
    models_files_path=models_files_path, Obs=obs_data, score="Pearson",
    month_of_initialization=3, clim_year_start=1981, clim_year_end=2010,
    dir_to_save_roc_reliability="./scores", lead_time=[1]
)

# Compute weighted GCM forecasts
hindcast_det, hindcast_prob, forecast_prob = verifier.weighted_gcm_forecasts(
    Obs=obs_data, best_models={"model1_MarIc_JFM_1": score1}, scores={"GROC": x_metric},
    lead_time=[1], model_dir="./models", clim_year_start=1981, clim_year_end=2010, variable=
↪"PRCP"
)
```

2.5.6 Annual Year Validation

The module provides utilities to validate forecasts for a specific year, including ratio-to-average classification and RPSS computation.

Key Methods

- *ratio_to_average*: Classifies forecast data relative to the climatological mean into categories (e.g., Well Above Average, Near Average).
- *compute_one_year_rpss*: Computes RPSS for a specific year and visualizes it on a map.

Example Usage

```
# Classify ratio to average for a specific year
verifier.ratio_to_average(predictant=obs_data, clim_year_start=1981, clim_year_end=2010,
↪year=2020)
```

(continues on next page)

(continued from previous page)

```
# Compute RPSS for a specific year
verifier.compute_one_year_rpss(
    obs=obs_data, prob_pred=proba_forecast, clim_year_start=1981, clim_year_end=2010, year=2020
)
```

- **Placeholder Functions**: Some methods (e.g., *taylor_diagram*) are placeholders and require implementation based on specific needs.
- **Gridded Data**: The module currently supports only gridded data validation. Non-gridded validation is not implemented.
- **Performance**: The use of *dask* ensures efficient computation for large datasets, but users should ensure proper chunking of *xarray* DataArrays.
- **Visualization**: Plots are saved to the specified directory and displayed using *matplotlib*. Ensure the output directory exists.

This documentation provides an overview of the Verification module's capabilities, along with example usage for key methods. For detailed information on each method, refer to the source code and docstrings in *was_verification.py*.

2.6 Multi-Model Ensemble (MME) Techniques

The *was_mme.py* module provides classes for combining predictions from multiple models, including:

- *WAS_mme_ELM*: Extreme Learning Machine for MME.
- *WAS_mme_EPOELM*: Enhanced Parallel Online Extreme Learning Machine.
- *WAS_mme_MLP*: Multi-Layer Perceptron for MME.
- *WAS_mme_GradientBoosting*: Gradient Boosting for MME.
- *WAS_mme_XGBoosting*: XGBoost for MME.
- *WAS_mme_AdaBoost*: AdaBoost for MME.
- *WAS_mme_LGBM_Boosting*: LightGBM Boosting for MME.
- *WAS_mme_Stack_MLP_RF*: Stacking model with MLP and Random Forest.
- *WAS_mme_Stack_Lasso_RF_MLP*: Stacking model with Lasso, Random Forest, and MLP.
- *WAS_mme_Stack_MLP_Ada_Ridge*: Stacking model with MLP, AdaBoost, and Ridge.
- *WAS_mme_Stack_RF_GB_Ridge*: Stacking model with Random Forest, Gradient Boosting, and Ridge.
- *WAS_mme_Stack_KNN_Tree_SVR*: Stacking model with KNN, Decision Tree, and SVR.
- *WAS_mme_GA*: Genetic Algorithm for MME.

Each MME class includes methods for computing the ensemble model and, where applicable, computing probabilities.

Example Usage with WAS_mme_ELM

```
from wass2s.was_mme import WAS_mme_ELM

# Define ELM parameters
elm_kwargs = {
    'regularization': 10,
    'hidden_layer_size': 4,
```

(continues on next page)

(continued from previous page)

```
    'activation': 'lin',  # Options: 'sigm', 'tanh', 'lin', 'relu'
    'preprocessing': 'none',  # Options: 'minmax', 'std', 'none'
    'n_estimators': 10,
}

# Initialize the MME ELM model
model = WAS_mme_ELM(elm_kwargs=elm_kwargs, dist_method="euclidean")

# Process datasets for MME (user-defined function)
all_model_hdcst, all_model_fcst, obs, best_score = process_datasets_for_mme(
    rainfall.sel(T=slice(str(year_start), str(year_end))),
    gcm=True, ELM_ELR=True, dir_to_save_model="./models",
    best_models=[], scores=[], year_start=1990, year_end=2020,
    model=True, month_of_initialization=3, lead_time=1, year_forecast=2021
)

# Initialize cross-validator
was_mme_gcm = WAS_Cross_Validator(
    n_splits=len(rainfall.sel(T=slice(str(year_start), str(year_end))).get_index("T")),
    nb_omit=2
)

# Perform cross-validation
hindcast_det_gcm, hindcast_prob_gcm = was_mme_gcm.cross_validate(
    model, obs, all_model_hdcst, clim_year_start, clim_year_end
)
```

CHAPTER

THREE

WASS2S SUBMODULES

3.1 wass2s.was_download module

3.2 wass2s.was_compute_predictand module

3.3 wass2s.was_merge_predictand module

3.4 wass2s.was_cross_validate module

3.5 wass2s.was_linear_models module

3.6 wass2s.was_eof module

3.7 wass2s.was_pcr module

3.8 wass2s.was_cca module

3.9 wass2s.was_machine_learning module

3.10 wass2s.was_analog module

3.11 wass2s.was_verification module

3.12 wass2s.was_mme module

3.13 wass2s.utils module