


climada_seasonal_forecast_sr / create_seasonal_forecast_hazard.py



 DahyannAraya Update_25

5517486 · 13 minutes ago

 History

Code

Blame

972 lines (831 loc) · 34.6 KB

Raw



```
1  """
2  This file is part of CLIMADA.
3
4  Copyright (C) 2017 ETH Zurich, CLIMADA contributors listed in AUTHORS.
5
6  CLIMADA is free software: you can redistribute it and/or modify it under the
7  terms of the GNU General Public License as published by the Free
8  Software Foundation, version 3.
9
10 CLIMADA is distributed in the hope that it will be useful, but WITHOUT ANY
11 WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
12 PARTICULAR PURPOSE. See the GNU General Public License for more details.
13
14 You should have received a copy of the GNU General Public License along
15 with CLIMADA. If not, see <https://www.gnu.org/licenses/>.
16
17 ---
18 Core interface for managing seasonal climate forecasts in CLIMADA.
19
20 This module provides the SeasonalForecast class, which enables:
21 - Downloading Copernicus seasonal forecast data for selected years and months.
22 - Processing raw GRIB or NetCDF data into standardized daily format.
23 - Computing user-defined climate indices (e.g., Heatwaves, Tropical Nights, Tmax).
24 - Converting the calculated indices into CLIMADA-compatible Hazard objects.
25 - Organizing outputs by forecast system, initialization time, and spatial domain.
26
27 The interface integrates several submodules under copernicus_interface:
28 - create_seasonal_forecast_hazard.py: implements the core SeasonalForecast class
29   that coordinates the entire workflow.
30 - downloader.py: handles forecast data retrieval from the CDS API.
31 - index_definitions.py: climate index definitions and variable handling.
32 - heat_index.py: calculate different thermal indices.
33 - seasonal_statistics.py: provides statistical postprocessing and index calculations
34 - path_utils.py: standardizes and validates file and folder structures.
35 - time_utils.py: computes lead times and handles month name conversions.
36 - forecast_skill.py: manages access and plotting of seasonal forecast skill scores f
37
38 All inputs and outputs are consistently managed through a pipeline structure that en
```

```

39     modularity, traceability, and ease of integration into CLIMADA workflows.
40
41     """
42     import calendar
43     import logging
44     from datetime import date
45     from pathlib import Path, PosixPath
46     from typing import List
47
48     import cartopy.crs as ccrs
49     import cartopy.feature as cfeature
50     import matplotlib.pyplot as plt
51     import numpy as np
52     import pandas as pd
53     import xarray as xr
54
55     from climada import CONFIG
56     from climada.hazard import Hazard
57     from climada_petals.hazard.copernicus_interface.downloader import download_data
58     from climada_petals.hazard.copernicus_interface.index_definitions import (
59         IndexSpecEnum,
60         get_short_name_from_variable,
61     )
62     import climada_petals.hazard.copernicus_interface.seasonal_statistics as seasonal_st
63     from climada_petals.hazard.copernicus_interface.time_utils import (
64         month_name_to_number,
65         calculate_leadtimes,
66     )
67     from climada_petals.hazard.copernicus_interface.path_utils import (
68         check_existing_files, get_file_path
69     )
70
71
72     # set path to store data
73     DATA_OUT = CONFIG.hazard.copernicus.seasonal_forecasts.dir()
74     LOGGER = logging.getLogger(__name__)
75
76
77     # ----- Main Class -----
78     ✓ class SeasonalForecast:
79         """
80         Class for managing the download, processing, and analysis of seasonal climate fo
81         """
82
83     ✓     def __init__(
84         self,
85         index_metric,
86         year_list,
87         forecast_period,
88         initiation_month,
89         bounds,
90         data_format,
91         originating_centre,

```

```

92         system,
93         data_out=None,
94     ):
95         """
96         Initialize the SeasonalForecast instance with user-defined parameters for in
97
98         Parameters
99         -----
100         index_metric : str
101             Climate index to calculate (e.g., "HW", "TR", "Tmax").
102         year_list : list of int
103             List of years for which data should be downloaded and processed.
104         lead_time_months : list of str or int
105             List specifying the start and end month (given as integers or strings)
106             of the valid forecast period. Must contain exactly two elements.
107         initiation_month : list of str
108             List of initiation months for the forecast (e.g., ["March", "April"]).
109         bounds : list of float
110             Bounding box values in EPSG 4326 format: (min_lon, min_lat, max_lon, max
111         data_format : str
112             Format of the downloaded data. Either "grib" or "netcdf".
113         originating_centre : str
114             Data provider (e.g., "dwd").
115         system : str
116             Forecast system configuration (e.g., "21").
117         data_out : pathlib.Path, optional
118             Output directory for storing downloaded and processed data. If None,
119             uses a default directory specified in the configuration.
120
121         Raises
122         -----
123         ValueError
124             If the valid period does not contain exactly two months.
125         """
126         # initiate initiation month, valid period, and leadtimes
127         valid_period = forecast_period
128         if not isinstance(initiation_month, list):
129             initiation_month = [initiation_month]
130         if not isinstance(valid_period, list) or len(valid_period) != 2:
131             raise ValueError("Valid period must be a list of two months.")
132         self.initiation_month_str = [
133             f"{month_name_to_number(month):02d}" for month in initiation_month
134         ]
135         self.valid_period = [month_name_to_number(month) for month in valid_period]
136         self.valid_period_str = "-".join(
137             [f"{month:02d}" for month in self.valid_period]
138         )
139
140         self.index_metric = index_metric
141         self.year_list = year_list
142         self.bounds = bounds
143         self.bounds_str = (
144             f"{bounds[0][0]:.1f}, {bounds[0][1]:.1f}, {bounds[1][0]:.1f}, {bounds[1][1]:.1f}"

```

```

144         f"boundsN{int(self.bounds[0])}_S{int(self.bounds[1])}_N"
145         f"E{int(self.bounds[2])}_W{int(self.bounds[3])}"
146     )
147     self.data_format = data_format
148     self.Originating_centre = originating_centre
149     self.system = system
150
151     # initialize base directory
152     self.data_out = Path(data_out) if data_out else DATA_OUT
153
154     # Get index specifications
155     index_spec = IndexSpecEnum.get_info(self.index_metric)
156     self.variables = index_spec.variables
157     self.variables_short = [
158         get_short_name_from_variable(var) for var in self.variables
159     ]
160
161
162     ##### Index Metadata Utilities #####
163
164     ✓ def explain_index(self, index_metric=None, print_flag=False):
165         """
166         Retrieve and display information about a specific climate index.
167
168         This function provides an explanation and the required input variables for
169         the selected climate index. If no index is provided, the instance's
170         `index_metric` is used.
171
172         Parameters
173         -----
174         index_metric : str, optional
175             Climate index to explain (e.g., 'HW', 'TR', 'Tmax'). If None, uses the
176             instance's index_metric.
177         print_flag : bool, optional
178             If True, prints the explanation. Default is False.
179
180         Returns
181         -----
182         str
183             Text description of the index explanation and required input variables.
184
185         Notes
186         -----
187         The index information is retrieved from `IndexSpecEnum.get_info`.
188         """
189         index_metric = index_metric or self.index_metric
190         response = (
191             f"Explanation for {index_metric}: "
192             f"{IndexSpecEnum.get_info(index_metric).explanation} "
193         )
194         response += (
195             "Required variables: "
196             f"{', '.join(IndexSpecEnum.get_info(index_metric).variables)}"

```

```

197         )
198         if print_flag:
199             print(response)
200         return response
201
202
203     ##### Path Utilities #####
204
205     ✓ def get_pipeline_path(self, year, initiation_month_str, data_type):
206         """
207         Provide (and possibly create) file paths for forecast pipeline.
208
209         Parameters
210         -----
211         year : int
212             Year of the forecast initiation.
213         init_month : str
214             Initiation month as two-digit string (e.g., '03' for March).
215         data_type : str
216             Type of data to access ('downloaded_data', 'processed_data', 'indices',
217
218         Returns
219         -----
220         Path or dict of Path
221             Path to the requested file(s). For 'indices', returns a dictionary with
222             'daily', 'monthly', 'stats'.
223
224         Raises
225         -----
226         ValueError
227             If unknown data_type is provided.
228
229         Notes
230         -----
231         File structure:
232         {base_dir}/{originating_centre}/sys{system}/{year}/init{init_month}/valid{va
233         /{data_type}
234         """
235         file_path = get_file_path(
236             self.data_out,
237             self.originating_centre,
238             year,
239             initiation_month_str,
240             self.valid_period_str,
241             data_type,
242             self.index_metric,
243             self.bounds_str,
244             self.system,
245             self.data_format,
246         )
247
248         # create directory if not existing
249         if data_type == "indices":

```

```

247         if data_type == 'indices':
248             file_path["monthly"].parent.mkdir(parents=True, exist_ok=True)
249         else:
250             file_path.parent.mkdir(parents=True, exist_ok=True)
251
252     return file_path
253
254 ##### Download and process #####
255
256 ✓ def _download(self, overwrite=False):
257     """
258     Download seasonal forecast data for the specified years and initiation month.
259
260     This function downloads the raw forecast data files for each year and initiation month
261     defined in the instance configuration. The data is downloaded in the specified format
262     ('grib' or 'netcdf') and stored in the configured directory structure.
263
264     Parameters
265     -----
266     overwrite : bool, optional
267         If True, existing downloaded files will be overwritten. Default is False
268
269     Returns
270     -----
271     dict
272         Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
273         and values corresponding to the downloaded data file paths.
274
275     Notes
276     -----
277     The data is downloaded using the `_download_data` function and follows the directory
278     structure defined in `get_pipeline_path`. The bounding box is automatically converted
279     to CDS (Climate Data Store) format before download.
280
281     """
282     output_files = {}
283     bounds_cds_order = [
284         self.bounds[3],
285         *self.bounds[:3],
286     ] # convert bounds to CDS order
287     for year in self.year_list:
288         for month_str in self.initiation_month_str:
289             leadtimes = calculate_leadtimes(year, int(month_str), self.valid_period_str)
290
291             # Generate output file name
292             downloaded_data_path = self.get_pipeline_path(
293                 year, month_str, "downloaded_data"
294             )
295
296             output_files[f"{year}_init{month_str}_valid{self.valid_period_str}"] =
297                 _download_data(
298                     downloaded_data_path,
299                     overwrite,
300                     self.variables,
301

```

```

302         year,
303         month_str,
304         self.data_format,
305         self.Originating_centre,
306         self.system,
307         bounds_cds_order,
308         leadtimes,
309     )
310 )
311
312     return output_files
313
314 ✓ def _process(self, overwrite=False):
315     """
316     Process the downloaded forecast data into daily NetCDF format.
317
318     This function processes the raw downloaded data files into a standardized
319     daily NetCDF format, applying basic aggregation operations (mean, max, min).
320     The processed files are saved in the configured output directory.
321
322     Parameters
323     -----
324     overwrite : bool, optional
325         If True, existing processed files will be overwritten. Default is False.
326
327     Returns
328     -----
329     dict
330         Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
331         and values corresponding to the processed NetCDF file paths.
332
333     Notes
334     -----
335     The processing applies a daily coarsening operation and aggregates the data.
336     The processed data is saved in NetCDF format in the directory defined by
337     `get_pipeline_path`. Processing is performed using the `_process_data` function.
338     """
339     processed_files = {}
340     for year in self.year_list:
341         for month_str in self.initiation_month_str:
342             # Locate input file name
343             downloaded_data_path = self.get_pipeline_path(
344                 year, month_str, "downloaded_data"
345             )
346             # Generate output file name
347             processed_data_path = self.get_pipeline_path(
348                 year, month_str, "processed_data"
349             )
350
351             processed_files[
352                 f"{year}_init{month_str}_valid{self.valid_period_str}"
353             ] = _process_data(
354                 processed_data_path,

```

```

355         processed_data_path,
356         overwrite,
357         downloaded_data_path,
358         self.variables_short,
359         self.data_format,
360     )
361
362     return processed_files
363
364 ✓ def download_and_process_data(self, overwrite=False):
365     """
366     Download and process seasonal climate forecast data.
367
368     This function performs the complete data pipeline by first downloading
369     the raw forecast data for the specified years and initiation months,
370     and then processing the downloaded data into a daily NetCDF format.
371
372     Parameters
373     -----
374     overwrite : bool, optional
375         If True, existing downloaded and processed files will be overwritten. De
376
377     Returns
378     -----
379     dict
380         Dictionary containing two keys:
381         - "downloaded_data": dict with file paths to downloaded raw data.
382         - "processed_data": dict with file paths to processed NetCDF data.
383
384     Raises
385     -----
386     Exception
387         If an error occurs during download or processing, such as invalid input
388         or file system issues.
389
390     Notes
391     -----
392     This is a high-level method that internally calls `_download()` and `_proces
393     The file structure and naming follow the configuration defined in `get_pipel
394     """
395
396     # Call high-level methods for downloading and processing
397     created_files = {}
398     try:
399         # 1) Attempt downloading data
400         created_files["downloaded_data"] = self._download(overwrite=overwrite)
401         # 2) Attempt processing data
402         created_files["processed_data"] = self._process(overwrite=overwrite)
403     except Exception as error:
404         # Catch reversed valid_period or any other ValueError from calculate_lea
405         raise RuntimeError(f"Download/process aborted: {error}") from error
406
407     return created_files

```



```

407
408 ##### Calculate index #####
409
410 ✓ def calculate_index(
411     self,
412     overwrite=False,
413     hw_threshold=27,
414     hw_min_duration=3,
415     hw_max_gap=0,
416     tr_threshold=20,
417 ):
418     """
419     Calculate the specified climate index based on the downloaded forecast data.
420
421     This function processes the downloaded or processed forecast data to compute
422     the selected climate index (e.g., Heatwave days, Tropical Nights) according
423     to the parameters defined for the index.
424
425     Parameters
426     -----
427     overwrite : bool, optional
428         If True, existing index files will be overwritten. Default is False.
429     hw_threshold : float, optional
430         Temperature threshold for heatwave days index calculation. Default is 27
431     hw_min_duration : int, optional
432         Minimum duration (in days) of consecutive conditions for a heatwave event
433     hw_max_gap : int, optional
434         Maximum allowable gap (in days) between conditions to still
435         consider as a single heatwave event. Default is 0.
436     tr_threshold : float, optional
437         Temperature threshold for tropical nights index calculation. Default is 20
438
439     Returns
440     -----
441     dict
442         Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
443         and values corresponding to the output NetCDF index files (daily, monthly)
444
445     Raises
446     -----
447     Exception
448         If index calculation fails due to missing files or processing errors.
449
450     Notes
451     -----
452     The input files used depend on the index:
453     - For 'TX30', 'TR', and 'HW', the raw downloaded GRIB data is used.
454     - For other indices, the processed NetCDF data is used.
455
456     The calculation is performed using the `_calculate_index` function and results
457     are saved in the configured output directory structure.
458     """
459     index_outputs = {}

```

```

460
461 # Iterate over each year and initiation month
462 for year in self.year_list:
463     for month_str in self.initiation_month_str:
464         LOGGER.info(
465             "Processing index %s for year %, initiation month %s.",
466             self.index_metric,
467             year,
468             month_str,
469         )
470
471     # Determine the input file based on index type
472     if self.index_metric in ["TX30", "TR", "HW"]: # Metrics using GRIB
473         input_data_path = self.get_pipeline_path(
474             year, month_str, "downloaded_data"
475         )
476     else: # Metrics using processed NC files
477         input_data_path = self.get_pipeline_path(
478             year, month_str, "processed_data"
479         )
480
481     # Generate paths for index outputs
482     index_data_paths = self.get_pipeline_path(year, month_str, "indices"
483
484     # Process the index and handle exceptions
485     try:
486         outputs = _calculate_index(
487             index_data_paths,
488             overwrite,
489             input_data_path,
490             self.index_metric,
491             tr_threshold=tr_threshold,
492             hw_min_duration=hw_min_duration,
493             hw_max_gap=hw_max_gap,
494             hw_threshold=hw_threshold,
495         )
496         index_outputs[
497             f"{year}_init{month_str}_valid{self.valid_period_str}"
498         ] = outputs
499
500     except FileNotFoundError:
501         msg = (
502             f"[Index Calculation] Skipped {self.index_metric} for "
503             f"year={year}, month={month_str} – input file not found. "
504             f"Expected: {input_data_path}"
505         )
506         LOGGER.warning(msg)
507
508     except Exception as error:
509         raise RuntimeError(
510             f"Error processing index {self.index_metric} for "
511             f"{year}-{month_str}: {error}"

```

```

512         ) from error
513
514     return index_outputs
515
516     ##### Calculate hazard #####
517
518     ✓ def save_index_to_hazard(self, overwrite=False):
519         """
520         Convert the calculated climate index to a CLIMADA Hazard object and save it
521
522         This function reads the monthly aggregated index NetCDF files and converts them
523         into a CLIMADA Hazard object. The resulting hazard files are saved in HDF5 format.
524
525         Parameters
526         -----
527         overwrite : bool, optional
528             If True, existing hazard files will be overwritten. Default is False.
529
530         Returns
531         -----
532         dict
533             Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
534             and values corresponding to the saved Hazard HDF5 file paths.
535
536         Raises
537         -----
538         Exception
539             If the hazard conversion fails due to missing input files or processing errors.
540
541         Notes
542         -----
543         The hazard conversion is performed using the `_convert_to_hazard` function.
544         The function expects that the index files (monthly NetCDF) have already been
545         calculated and saved using `calculate_index()`.
546
547         The resulting Hazard objects follow CLIMADA's internal structure and can be
548         used for further risk assessment workflows.
549         """
550         hazard_outputs = {}
551
552         for year in self.year_list:
553             for month_str in self.initiation_month_str:
554                 LOGGER.info(
555                     "Creating hazard for index %s for year %s, initiation month %s." %
556                     self.index_metric,
557                     year,
558                     month_str,
559                 )
560                 # Get input index file paths and hazard output file paths
561                 index_data_path = self.get_pipeline_path(year, month_str, "indices")
562                 ["monthly"]
563             ]
564             hazard_data_path = self.get_pipeline_path(year, month_str, "hazard")

```

```

565
566         try:
567             # Convert index file to Hazard
568             hazard_outputs[
569                 f"{year}_init{month_str}_valid{self.valid_period_str}"
570             ] = _convert_to_hazard(
571                 hazard_data_path,
572                 overwrite,
573                 index_data_path,
574                 self.index_metric,
575             )
576
577         except FileNotFoundError:
578             msg = (
579                 f"[Hazard Conversion] Skipped {self.index_metric} for year={year},
580                 f"month={month_str} - monthly index file not found."
581             )
582             LOGGER.warning(msg)
583
584         except Exception as error:
585             raise RuntimeError(
586                 f"Hazard creation failed for {year}-{month_str}: {error}"
587             ) from error
588
589         return hazard_outputs
590
591
592     ##### Utility Functions #####
593
594     ✓ def handle_overwriting(function):
595         """
596         Decorator to handle file overwriting during data processing.
597
598         This decorator checks if the target output file(s) already exist and
599         whether overwriting is allowed. If the file(s) exist and overwriting
600         is disabled, the existing file paths are returned without executing
601         the decorated function.
602
603         Parameters
604         -----
605         function : callable
606             Function to be decorated. Must have the first two arguments:
607             - output_file_name : Path or dict of Path
608             - overwrite : bool
609
610         Returns
611         -----
612         callable
613             Wrapped function with added file existence check logic.
614
615         Notes
616         -----
617         If 'output_file_name' is a 'Path' object and 'overwrite' is checked

```

```

617         - If `output_file_name` is a `Path`, its existence is checked.
618         - If `output_file_name` is a `dict` of `Path`, the existence of any file is checked.
619         - If `overwrite` is False and the file(s) exist, the function is skipped and the
620           existing path(s) are returned.
621         - The function must accept `overwrite` as the second argument.
622         """
623
624     def wrapper(output_file_name, overwrite, *args, **kwargs):
625         # if data exists and we do not want to overwrite
626         if isinstance(output_file_name, PosixPath):
627             if not overwrite and output_file_name.exists():
628                 LOGGER.info("%s already exists.", output_file_name)
629                 return output_file_name
630             elif isinstance(output_file_name, dict):
631                 if not overwrite and any(
632                     path.exists() for path in output_file_name.values()
633                 ):
634                     existing_files = [str(path) for path in output_file_name.values()]
635                     LOGGER.info("One or more files already exist: %s", existing_files)
636                     return output_file_name
637
638                 return function(output_file_name, overwrite, *args, **kwargs)
639
640     return wrapper
641
642
643     ##### Decorated Functions #####
644
645     @handle_overwriting
646     def _download_data(
647         output_file_name,
648         overwrite,
649         variables,
650         year,
651         initiation_month,
652         data_format,
653         originating_centre,
654         system,
655         bounds_cds_order,
656         leadtimes,
657     ):
658         """
659         Download seasonal forecast data for a specific year and initiation month.
660
661         This function downloads raw seasonal forecast data from the Copernicus
662         Climate Data Store (CDS) based on the specified forecast configuration
663         and geographical domain. The data is saved in the specified format and
664         location.
665
666         Parameters
667         -----
668         output_file_name : Path
669             Path to save the downloaded data file.

```

```

670     overwrite : bool
671         If True, existing files will be overwritten. If False and the file exists,
672         the download is skipped.
673     variables : list of str
674         List of variable names to download (e.g., ['tasmax', 'tasmin']).
675     year : int
676         Year of the forecast initiation.
677     initiation_month : int
678         Month of the forecast initiation (1-12).
679     data_format : str
680         File format for the downloaded data ('grib' or 'netcdf').
681     originating_centre : str
682         Forecast data provider (e.g., 'dwd' for German Weather Service).
683     system : str
684         Model system identifier (e.g., '21').
685     bounds_cds_order : list of float
686         Geographical bounding box in CDS order: [north, west, south, east].
687     leadtimes : list of int
688         List of forecast lead times in hours.
689
690     Returns
691     -----
692     Path
693         Path to the downloaded data file.
694
695     Notes
696     ----
697     The function uses the `download_data` method from the Copernicus interface module.
698     The downloaded data is stored following the directory structure defined by the p
699     """
700     # Prepare download parameters
701     download_params = {
702         "data_format": data_format,
703         "originating_centre": originating_centre,
704         "area": bounds_cds_order,
705         "system": system,
706         "variable": variables,
707         "month": initiation_month,
708         "year": year,
709         "day": "01",
710         "leadtime_hour": leadtimes,
711     }
712
713     # Perform download
714     downloaded_file = download_data(
715         "seasonal-original-single-levels",
716         download_params,
717         output_file_name,
718         overwrite=overwrite,
719     )
720
721     return downloaded_file
722

```

```

722
723
724     @handle_overwriting
725     def _process_data(output_file_name, overwrite, input_file_name, variables, data_format):
726         """
727         Process a downloaded forecast data file into daily NetCDF format.
728
729         This function reads the downloaded forecast data (in GRIB or NetCDF format),
730         applies a temporal coarsening operation (aggregation over 4 time steps),
731         and saves the resulting daily data as a NetCDF file. For each variable,
732         daily mean, maximum, and minimum values are computed.
733
734         Parameters
735         -----
736         output_file_name : Path
737             Path to save the processed NetCDF file.
738         overwrite : bool
739             If True, existing processed files will be overwritten. If False and the file
740             the processing is skipped.
741         input_file_name : Path
742             Path to the input downloaded data file.
743         variables : list of str
744             List of short variable names to process (e.g., ['tasmax', 'tasmin']).
745         data_format : str
746             Format of the input file ('grib' or 'netcdf').
747
748         Returns
749         -----
750         Path
751             Path to the saved processed NetCDF file.
752
753         Raises
754         -----
755         FileNotFoundError
756             If the input file does not exist.
757         Exception
758             If an error occurs during data processing.
759
760         Notes
761         -----
762         The function performs a temporal aggregation by coarsening the data over 4 time steps,
763         resulting in daily mean, maximum, and minimum values for each variable.
764         The processed data is saved in NetCDF format and can be used for index calculation.
765         """
766         try:
767             with xr.open_dataset(
768                 input_file_name,
769                 engine="cfgrib" if data_format == "grib" else None,
770             ) as input_dataset:
771                 # Coarsen the data
772                 ds_mean = input_dataset.coarsen(step=4, boundary="trim").mean()
773                 ds_max = input_dataset.coarsen(step=4, boundary="trim").max()
774                 ds_min = input_dataset.coarsen(step=4, boundary="trim").min()

```

```

775
776     # Create a new dataset combining mean, max, and min values
777     combined_ds = xr.Dataset()
778     for var in variables:
779         combined_ds[f"{var}_mean"] = ds_mean[var]
780         combined_ds[f"{var}_max"] = ds_max[var]
781         combined_ds[f"{var}_min"] = ds_min[var]
782
783     # Save the combined dataset to NetCDF
784     combined_ds.to_netcdf(str(output_file_name))
785     LOGGER.info("Daily file saved to %s", output_file_name)
786
787     return output_file_name
788
789 except FileNotFoundError as error:
790     raise FileNotFoundError(
791         f"Input file {input_file_name} does not exist. Processing failed."
792     ) from error
793 except Exception as error:
794     raise RuntimeError(
795         f"Error during processing for {input_file_name}: {error}"
796     ) from error
797
798
799 @handle_overwriting
800 ✓ def _calculate_index(
801     output_file_names,
802     overwrite,
803     input_file_name,
804     index_metric,
805     tr_threshold=20,
806     hw_threshold=27,
807     hw_min_duration=3,
808     hw_max_gap=0,
809 ):
810     """
811     Calculate and save climate indices based on the input data.
812
813     Parameters
814     -----
815     output_file_names : dict
816         Dictionary containing paths for daily, monthly, and stats output files.
817     overwrite : bool
818         Whether to overwrite existing files.
819     input_file_name : Path
820         Path to the input file.
821     index_metric : str
822         Climate index to calculate (e.g., 'HW', 'TR').
823     threshold : float, optional
824         Threshold for the index calculation (specific to the index type).
825     min_duration : int, optional
826         Minimum duration for events (specific to the index type).
827     max_gap : int, optional

```



```

827         max_gap : int, optional
828             Maximum gap allowed between events (specific to the index type).
829     tr_threshold : float, optional
830         Threshold for tropical nights (specific to the 'TR' index).
831
832     Returns
833     -----
834     dict
835         Paths to the saved index files.
836     """
837     # Define output paths
838     daily_output_path = output_file_names["daily"]
839     monthly_output_path = output_file_names["monthly"]
840     stats_output_path = output_file_names["stats"]
841
842     ds_daily, ds_monthly, ds_stats = seasonal_statistics.calculate_heat_indices_metric(
843         input_file_name,
844         index_metric,
845         tr_threshold=tr_threshold,
846         hw_threshold=hw_threshold,
847         hw_min_duration=hw_min_duration,
848         hw_max_gap=hw_max_gap,
849     )
850
851     # Save outputs
852     if ds_daily is not None:
853         ds_daily.to_netcdf(daily_output_path)
854         LOGGER.info("Saved daily index to %s", daily_output_path)
855     if ds_monthly is not None:
856         ds_monthly.to_netcdf(monthly_output_path)
857         LOGGER.info("Saved monthly index to %s", monthly_output_path)
858     if ds_stats is not None:
859         ds_stats.to_netcdf(stats_output_path)
860         LOGGER.info("Saved stats index to %s", stats_output_path)
861
862     return {
863         "daily": daily_output_path,
864         "monthly": monthly_output_path,
865         "stats": stats_output_path,
866     }
867
868
869     @handle_overwriting
870     ✓ def _convert_to_hazard(output_file_name, overwrite, input_file_name, index_metric):
871         """
872         Convert a climate index file to a CLIMADA Hazard object and save it as HDF5.
873
874         This function reads a processed climate index NetCDF file, converts it to a
875         CLIMADA Hazard object, and saves it in HDF5 format. The function supports
876         ensemble members and concatenates them into a single Hazard object.
877
878     Parameters
879     -----

```

```

880     output_file_name : Path
881         Path to save the generated Hazard HDF5 file.
882     overwrite : bool
883         If True, existing hazard files will be overwritten. If False and the file exists,
884         the conversion is skipped.
885     input_file_name : Path
886         Path to the input NetCDF file containing the calculated climate index.
887     index_metric : str
888         Climate index metric used for hazard creation (e.g., 'HW', 'TR', 'Tmax').
889
890     Returns
891     -----
892     Path
893         Path to the saved Hazard HDF5 file.
894
895     Raises
896     -----
897     KeyError
898         If required variables (e.g., 'step' or index variable) are missing in the dataset.
899     Exception
900         If the hazard conversion process fails.
901
902     Notes
903     -----
904     - The function uses `Hazard.from_xarray_raster()` to create Hazard objects
905       from the input dataset.
906     - If multiple ensemble members are present, individual Hazard objects are
907       created for each member and concatenated.
908     - The function determines the intensity unit based on the selected index:
909       - '%' for relative humidity (RH)
910       - 'days' for duration indices (e.g., 'HW', 'TR', 'TX30')
911       - '°C' for temperature indices
912     """
913     try:
914         with xr.open_dataset(str(input_file_name)) as input_dataset:
915             if "step" not in input_dataset.variables:
916                 raise KeyError(
917                     f"Missing 'step' variable in dataset for {input_file_name}."
918                 )
919
920             input_dataset["step"] = xr.DataArray(
921                 [f"{date}-01" for date in input_dataset["step"].values],
922                 dims=["step"],
923             )
924             input_dataset["step"] = pd.to_datetime(input_dataset["step"].values)
925
926             ensemble_members = input_dataset.get("number", [0]).values
927             hazards = []
928
929             # Determine intensity unit and variable
930             intensity_unit = (
931                 "%"
932                 if index_metric == "RH"

```

```

933         else "days" if index_metric in ["TR", "TX30", "HW"] else "°C"
934     )
935     intensity_variable = index_metric
936
937     if intensity_variable not in input_dataset.variables:
938         raise KeyError(
939             f"No variable named '{intensity_variable}' in the dataset. "
940             f"Available variables: {list(input_dataset.variables)}"
941         )
942
943     # Create Hazard objects
944     for member in ensemble_members:
945         ds_subset = input_dataset.sel(number=member) if "number" in input_da
946         hazard = Hazard.from_xarray_raster(
947             data=ds_subset,
948             hazard_type=index_metric,
949             intensity_unit=intensity_unit,
950             intensity=intensity_variable,
951             coordinate_vars={
952                 "event": "step",
953                 "longitude": "longitude",
954                 "latitude": "latitude",
955             },
956         )
957         hazard.event_name = [
958             f"member{member}" for _ in range(len(hazard.event_name))
959         ]
960         hazards.append(hazard)
961
962     hazard = Hazard.concat(hazards)
963     hazard.check()
964     hazard.write_hdf5(str(output_file_name))
965
966     LOGGER.info("Hazard file saved to %s.", output_file_name)
967     return output_file_name
968
969 except Exception as error:
970     raise RuntimeError(
971         f"Hazard conversion failed for {input_file_name}: {error}"
972     ) from error

```