Appendix C. API Documentation

This section describes the methods available in the *Climate Econometrics Toolkit API*.

Appendix C.1. Methods for Data Preprocessing

`extract_raster_data(`
`raster_file, shape_file=None, weights=None, weight_file=None):`
Gridded raster data extraction via provided shape file and optional weights argument. Gridded weights will be reprojected to match provided raster file if necessary.
raster_file: a string representing the path to a gridded raster data file
shape_filepath (optional): a string representing the path to a Shapes file (.shp). If None, a default country shapes file will be used.
weights (optional): a string indicating the use of one of the built-in weight files. Must be one of: 'popweighted', 'cropweighted', 'maizeweighted', 'riceweighted', 'soybeanweighted', 'wheatweighted'.
weight_file (optional): a string representing the path to a gridded raster file of weights
Returns: a dictionary containing extracted results

`aggregate_raster_data(extracted_climate_data, climate_var_name, aggregation_func, subperiods_per_time_unit, starting_year, shape_file=None, geo_identifier=None, subperiods_to_use=None, crop=None):` Aggregates sub-yearly output of *extract_raster_data* into a DataFrame based on the aggregation of the data to the yearly level.
extracted_climate_data: the output from *extract_raster_data*
climate_var_name: String representing the name to assign to the extracted variable
aggregation_func: String, either 'mean' or 'sum'
superiods_per_year: An integer representing the number of observations in *extracted_climate_data* that comprise a year. For example, if the data is monthly, the integer 12 should be entered. Observations at the daily or sub-daily levels (e.g. > 365) must be divisible by either 365 or 366. Leap years will be automatically applied using the Python calendar module.
starting_year: an integer representing the first year for which there is data in *extracted_climate_data*
shape_file (optional): The same shape file that was provided to the call to

extract_raster_data, if provided. For use only to retrieve names of geographical entities. If no shape file is provided, the built-in country-level shape file will be used.
geo_identifier (optional): A string representing the geographic identifier column in the shape file to use for naming geographic entities in the output. Must be present as a column in the provided shape file. If a shape file is provided, this argument should also be provided, otherwise it is not needed.
subperiods_to_use (optional): a dictionary representing geographic entity-specific subperiods to use to filter the aggregation, with string keys representing geographic entities and a list of integer values representing subperiods to include in the aggregation. For example, assuming a monthly to yearly aggregation, if one wanted to restrict the aggregated observations to only include February for country A, one can add the following to the dictionary: A:[2]
crop (optinal): a string indicating to automatically filter the aggregation to only include data that occurred during the specified crop's growing season. Only works if one's data is at the country level and uses ISO3/GMI identifiers, which is the case if the built-in shape file is used during extraction. Must be one of: 'maize','rice','soybeans','wheat.spring','wheat.winter'
Returns: a Pandas DataFrame with the aggregated data

`compute_degree_days(years, countries, threshold, mode='above', panel_column_name='ISO3', time_column_name='year', crop=None, second_threshold=None):` Computes temperature degree days for given countries and years, optionally limited to crop-specific growing seasons
years: an iterable of years for which to calculate degree days
countries: an iterable of ISO3 country codes
threshold: a float indicating the degree day temperature threshold in Celsius
mode: a string indicating how to evaluate the threshold ("above", "below", or "between")
panel_column_name: the name of the column identifying countries or regions in the result dataframe
time_column_name: the name of the column identifying time in the result dataframe
crop (optional): a string representing the crop type to use for growing season limits (e.g., "maize")
second_threshold (optional): a float used as the upper bound when `mode='between'` is specified

Returns: a pandas DataFrame containing degree day values by country and year

`add_degree_days_to_dataframe(dataframe, threshold, panel_column='ISO3', time_column='year', mode='above', crop=None, second_threshold=None):` Adds a new column with computed degree days to a dataframe based on country and year
dataframe: a pandas DataFrame with panel and time columns
threshold: a float specifying the temperature threshold
panel_column: the name of the column identifying panel units (e.g., "ISO3")
time_column: the name of the column representing time (e.g., "year")
mode: how to compute degree days relative to the threshold ("above", "below", or "between")
crop (optional): restricts computation to a specific crops growing season
second_threshold (optional): used when `mode='between'` to define an upper threshold
Returns: a merged pandas DataFrame with the degree day column added

`integrate(dataframes, keep_na=False, panel_column='ISO3', time_column='year'):` Merges multiple panel datasets on panel and time dimensions
dataframes: a list of pandas DataFrames to merge
keep_na: a boolean indicating whether to keep all rows (outer join) or only overlapping rows (inner join)
panel_column: the name of the column identifying the panel unit
time_column: the name of the column identifying the time unit
Returns: a pandas DataFrame with merged contents

`convert_between_administrative_levels(data, from_code, to_code):` Converts a list of administrative location names between levels using internal lookup tables
data: a list or pandas Series of administrative region names to convert
from_code: a string indicating the current administrative level ("admin1" or "admin2")
to_code: a string indicating the target administrative level ("admin1" or "country")
Returns: a pandas Series with converted administrative level names

`load_climate_data(weight='unweighted')`: Loads climate data from the preprocessed dataset
weight: a string specifying the temperature weighting method to use; must be one of: 'popweighted', 'cropweighted', 'maizeweighted', 'riceweighted', 'soybeanweighted', 'wheatweighted'
Returns: a pandas DataFrame containing gridded climate data

`load_temperature_humidity_index_data(weight='unweighted')`: Loads temperature-humidity index data
weight: a string specifying the temperature weighting method to use; must be one of: 'popweighted', 'cropweighted', 'maizeweighted', 'riceweighted', 'soybeanweighted', 'wheatweighted'
Returns: a pandas DataFrame with temperature-humidity index data

`load_ndvi_data(weight='unweighted')`: Loads NDVI (Normalized Difference Vegetation Index) data from preprocessed datasets
weight: a string specifying the temperature weighting method to use; must be one of: 'popweighted', 'cropweighted', 'maizeweighted', 'riceweighted', 'soybeanweighted', 'wheatweighted'
Returns: a pandas DataFrame containing global NDVI time series

`load_emdat_data()`: Loads disaster occurrence data from the EM-DAT dataset
Returns: a pandas DataFrame of disaster events by country and year

`load_faostat_data()`: Loads FAOSTAT food production index data
Returns: a pandas DataFrame with agricultural production indices by country and year

`load_usda_fda_data()`: Loads USDA-FDA data on global total factor productivity (TFP) in agriculture
Returns: a pandas DataFrame with country-level TFP data from USDA FDA

`load_worldbank_gdp_data()`: Loads GDP data from the World Bank by country and year
Returns: a pandas DataFrame containing GDP values at the country level

`load_spei_data(weight='unweighted')`: Loads the Standardized Precipitation-

Evapotranspiration Index (SPEI) dataset
weight: a string indicating the weighting method; must be one of: 'pop-weighted', 'cropweighted', 'maizeweighted', 'riceweighted', 'soybeanweighted', 'wheatweighted'
Returns: a pandas DataFrame with SPEI values

`get_temperature_humidity_index(temp_data, relative_humidity_data):` Computes the temperature-humidity index from supplied temperature and relative humidity data
temp_data: a numeric array or pandas Series of daily average temperature values (in Celsius)
relative_humidity_data: a numeric array or pandas Series of daily relative humidity values (percent, from 0 to 100)
Returns: a numeric array of computed temperature-humidity index values

Appendix C.2. Methods for Econometric Model Analysis

`reset_model()`: Resets the global model instance to a new `ClimateEconometricsModel` object
Returns: None

`evaluate_model(std_error_type='nonrobust')`: Fit the current model with OLS, including an evaluation with out-of-sample cross-validation.
std_error_type: The type of standard error to compute during model fitting. Must be one of: "nonrobust", "whitehuber", "neweywest", "clusteredtime", "clusteredspace", "driscollkraay"
Returns: a string representing the model ID assigned to the current model

`get_best_model(metric='r2')`: Get the best model from the current session, based on the supplied metric. *metric*: One of "out_sample_mse_reduction", "out_sample_mse","out_sample_pred_int_cov","rmse","r2".
Returns: an instance of class ClimateEconometricModel

`get_all_model_ids()`: Get IDs of all models in the current session.
Returns: a list of strings representing the model IDs

`get_model_by_id(model_id)`: Get the model object corresponding to the given model ID

model_id: a string representing a model ID
Returns: a ClimateEconometricsModel

`get_all_models_from_cache()`: Loads and returns all models stored in the cache for the currently loaded dataset
Returns: a list of model objects from the cache

`load_dataset_from_file(datafile)`: Loads the given filepath as a CSV into a Pandas DataFrame and adds it to the current model
datafile: a string representing a local CSV file
Returns: None

`set_dataset(dataframe, dataset_name)`: Adds the given pandas DataFrame to the current model
dataframe: a pandas DataFrame containing panel data
dataset_name: a string supplying a name for the dataset
Returns: None

`view_current_model()`: Prints details of the current model
Returns: None

`set_target_variable(var, existence_check=True)`: Sets the dependent (target) variable in the current model
var: a string representing a column in the loaded dataset
existence_check: Boolean indicating whether to check to see if a dataset is loaded and if *var* exists in the loaded dataset before attempting to add to model
Returns: None

`set_time_column(var)`: Sets the time column in the current model
var: a string representing a column in the loaded dataset
Returns: None

`set_panel_column(var)`: Sets the panel column in the current model
var: a string representing a column in the loaded dataset
Returns: None

`add_transformation(var, transformations, keep_original_var=True)`:

add a transformation or transformations to a single model variable
var: a string representing a column in the loaded dataset
transformations: a list of transformations to be applied to the specified model variable. All list items should be one of "fd","sq","cu","ln","lag1","lag2","lag3".
keep_original_var: Boolean indicating whether the non-transformed variable should be kept or removed from the model after the transformation is applied to it
Returns: None

`add_covariates(vars, existence_check=True)`: add a covariate or covariates to the current model
vars: a list of strings representing columns in the loaded dataset
existence_check: Boolean indicating whether to check to see if a dataset is loaded and if *var* exists in the loaded dataset before attempting to add to model
Returns: None

`add_fixed_effects(vars)`: add a fixed-effect or fixed-effects to the current model
vars: a list of strings representing columns in the loaded dataset
Returns: None

`add_random_effect(var, group)`: add a random-effect to the coefficient of the specified variable, based on the specified group
var: a string representing a model covariate for which the coefficient should be stratified by `group`
group: a string representing the group to stratify the specified covariate by. Typically should be a time- or geography-based column in the loaded dataset.
Returns: None

`add_time_trend(vars, exp)`: add a time trend or time trends to the current model
vars: a list of strings representing columns in the loaded dataset
exp: integer representing the power of the time trend. For instance, 1 means that the time trend will be linear, 2 will be quadratic, 3 will be cubic, etc.
Returns: None

`remove_covariates(vars)`: remove a covariate or covariates from the

current model
vars: a list of strings representing covariates in the current model
Returns: None

`remove_time_trend(var, exp)`: remove a time trend from the current model
var: a string representing a variable with a time trend
exp: an integer representing the power of the time trend to remove
Returns: None

`remove_fixed_effect(fe)`: remove a fixed-effect from the current model
var: a string representing a variable with a fixed-effect
Returns: None

`remove_random_effect(add_to_covariate_list=True)`: remove the random-effect from the current model
add_to_covariate_list: a Boolean indicating whether to keep the variable to which the random-effect applied as a covariate in the model or remove it (default True)
Returns: None

`remove_transformation(var, transformations)`: remove a transformed variable from the current model
var: a transformed variable in the current model
transformations: the list of transformations applied to the transformed variable to remove
Returns: None

`run_bayesian_regression(model, num_samples=1000`: Run Bayesian Inference against the current model and dataset. Results will be saved to the subdirectory `bayes_samples` in the CET home directory. This command can be long running.
model: a ClimateEconometricsModel
num_samples: the number of posterior samples to generate for each sampling chain
Returns: None

`run_block_bootstrap(model, num_samples)`: Run bootstrapping against

the current model and dataset. Results will be saved to the subdirectory `bootstrap_samples` in the CET home directory. This command can be long running.
model: a ClimateEconometricsModel
num_samples: the number of bootstrap samples to generate
Returns: None

`run_specification_search(metric='out_sample_mse_reduction')`: Performs a grid search over possible model specifications and returns the best model based on the chosen evaluation metric
metric: a string indicating the evaluation metric to use; must be one of: 'out_sample_mse_reduction','out_sample_mse','out_sample_pred_int_cov','rmse','r2'
Returns: the best model object found during the search

`run_spatial_regression(reg_type, std_error_type='nonrobust', geometry_column=None, k=5, num_lags=1)`: Run a spatial lag or error regression on the current model using the specified regression type and optional geometry column. Output is saved in *cet_home/spatial_regression_output*.
reg_type: a string specifying the type of spatial regression to run. Must be one of: "lag" or "error"
std_error_type: A string representing the type of standard error to compute with model fitting. Must be one of: "nonrobust", "whitehuber", "neweywest". Ignored if reg_type is "error". (default "nonrobust")
geometry_column (optional): a string specifying the geometry column name to use for spatial weights. The column should contain spatial polygons for each geographical identifier in the data. If not specified, geometry column will be automatically generated based on the geographical identifiers in the data. Note that this only works if data contains ISO3/GMI identifiers; otherwise an error will be thrown (default None)
k: integer identifying the number of nearest neighbors to consider in the computation of the k-nearest-neighbors spatial weight matrix (default 5)
num_lags: integer identifying the number of orders of the spatial weight matrix to include. Ignored if reg_type is "error". (default 1)
Returns: None

`run_quantile_regression(q, std_error_type='nonrobust')`: Run quantile regression(s) on the current model for one or multiple quantiles. Output is saved in *cet_home/quantile_regression_output*.

q: a float or list of floats specifying the quantile(s) to run regression for (e.g., 0.25, 0.5, 0.75)
std_error_type: A string representing the type of standard error to compute with model fitting. Must be one of: "nonrobust", "greene". Note that, for quantile regression, the underlying Statsmodels implementation lists the robust standard error available as "heteroskedasticity robust standard errors (as suggested in Greene 6th edition)", which is specified here with the 'greene' argument value. (default "nonrobust")
Returns: None

`run_adf_panel_unit_root_tests()`: Run Augmented Dickey-Fuller panel unit root tests on the current model
Returns: a pandas DataFrame containing the results of the panel unit root tests

`run_engle_granger_cointegration_check()`: Run Engle-Granger cointegration tests on the current model
Returns: a pandas DataFrame containing the results of the cointegration tests

`run_pesaran_cross_sectional_dependence_check()`: Run Pesaran cross-sectional dependence tests on the current model
Returns: a pandas DataFrame containing the results of the cross-sectional dependence tests

`transform_data(data, model, include_target_var=True, demean=False)`: Transform the supplied dataset according to the model configuration, optionally including the target variable and applying demeaning
data: a pandas DataFrame containing the data to be transformed
model: an instance of the model used to guide the transformation
include_target_var (optional): a boolean indicating whether to include the target variable in the transformation (default True)
demean (optional): a boolean indicating whether to demean variables during the transformation (default False)
Returns: a pandas DataFrame containing the transformed data

Appendix C.3. Methods for Computation of Impacts

`predict_out_of_sample(model, data, transform_data=False, var_map=None)`: use the fitted model to generate predictions on out-of-sample data

model: a ClimateEconometricsModel

data: a Pandas DataFrame with columns for all variables in the model

transform_data: a Boolean indicating whether to apply the data transformations (e.g. square, log, first difference) to the out-of-sample data

var_map (optional): A dictionary of variable names, in the case that the out-of-sample data has column names that do not match the variable names in the model

Returns: a Pandas DataFrame containing the predictions

`cumulative_sum_of_predictions_by_geolocation(model, predictions, geo_weights=None, prediction_columns=None)`: apply the yearly cumulative sum to predictions for all geolocations.

Called by invoking the API with `call_user_prediction_function( 'geotemporal_cumulative_sum', [predictions, geo_weights, prediction_columns])`

model: a ClimateEconometricsModel

prediction: a Pandas DataFrame containing predictions generated by the model

geo_weights (optional): a dictionary containing weights for each geolocation, which will be multiplied with the cumulative sum of the impacts for each geolocation

prediction_columns (optional): a list of columns in the predict DataFrame to include. This is useful if you want to restrict the number of prediction samples generated by bootstrapping or Bayesian inference post-hoc.

`multiply_geo_coefficients_by_data_column(group_column, data, coefficients, multiplier_column)`: Multiplies coefficients by values from a specified column for each geolocation group in the data and sums the results.

Called by invoking the API with `call_user_prediction_function( 'multiply_geo_coefficients_by_data_column', [group_column, data, coefficients, multiplier_column])`

group_column: a string representing the column name used to group the data by geolocation or other grouping

data: a DataFrame containing the data to be multiplied by coefficients
coefficients: a dictionary where keys are coefficient names in the format `multiplier_column_geoLoc` and values are the coefficient values
multiplier_column: a string representing the column name in `data` whose values are multiplied by the corresponding coefficients
Returns: a dictionary mapping geolocations to the summed multiplied results

`convert_geo_log_loss_to_percent(effect_by_geo_loc):` Converts log loss values for each geolocation into percentage effects using the exponential minus one transformation.
Called by invoking the API with `call_user_prediction_function( 'convert_geo_log_loss_to_percent', [effect_by_geo_loc])`
effect_by_geo_loc: a dictionary mapping geolocations to arrays or lists of log loss values
Returns: a dictionary mapping geolocations to arrays of percentage effects