copro Documentation

Release 0.1.1

Jannis M. Hoch

Jul 02, 2021

CONTENTS

This is the documentation of CoPro, a machine-learning tool for conflict risk projections.

A software description paper was published in JOSS.

ONE

MAIN GOAL

With CoPro it is possible to apply machine-learning techniques to make projections of future areas at risk. CoPro was developed with a rather clear application in mind, unravelling the interplay of socio-economic development, climate change, and conflict occurrence. Nevertheless, we put a lot of emphasis on making it flexible. We hope that other, related questions about climate and conflict can be tackled as well, and that process understanding is deepened further.

CONTENTS

2.1 Installation

2.1.1 From GitHub

To install CoPro from GitHub, first clone the code. It is advised to create a separate environment first.

Note: We recommend to use Anaconda or Miniconda to install CoPro as this was used to develop and test the model. For installation instructions, see here.

```
$ git clone https://github.com/JannisHoch/copro.git
$ cd path/to/copro
$ conda env create -f environment.yml
```

It is now possible to activate this environment with

```
$ conda activate copro
```

To install CoPro in editable mode in this environment, run this command next in the CoPro-folder:

```
$ pip install -e .
```

2.1.2 From PyPI

To install CoPro directly from PyPI, use the following command.

```
pip install copro
```

2.1.3 From conda

Todo: This is not yet supported. Feel invited to provide a pull request enabling installation via conda.

2.2 Model execution

To be able to run the model, the conda environment has to be activated first.

```
$ conda activate copro
```

2.2.1 Runner script

To run the model, a command line script is provided. The usage of the script is as follows:

```
Usage: copro_runner [OPTIONS] CFG

Main command line script to execute the model.
All settings are read from cfg-file.
One cfg-file is required argument to train, test, and evaluate the model.
Multiple classifiers are trained based on different train-test data combinations.
Additional cfg-files for multiple projections can be provided as optional arguments,
→whereby each file corresponds to one projection to be made.
Per projection, each classifiers is used to create separate projection outcomes per time
→step (year).
All outcomes are combined after each time step to obtain the common projection outcome.

Args:     CFG (str): (relative) path to cfg-file

Options:
-plt, --make_plots      add additional output plots
-v, --verbose           command line switch to turn on verbose mode
```

Help information can be accessed with

```
$ copro_runner --help
```

All data and settings are retrieved from the configuration-file (`cfg-file`, see *Settings*) which needs to be provided as command line argument. In the cfg-file, the various settings of the simulation are defined.

A typical command would thus look like this:

```
$ copro_runner settings.cfg
```

In case issues occur, updating `setuptools` may be required.

```
$ pip3 install --upgrade pip setuptools
```

2.2.2 Binder

There is also a notebook running on Binder.

Please check it out to go through the model execution step-by-step and interactively explore the functionalities of CoPro.

2.3 Settings

2.3.1 The cfg-file

The main model settings need to be specified in a configuration file (`cfg-file`). This file looks like this.

```
[general]
input_dir=./path/to/input_data
output_dir=./path/to/store/output
# 1: all data. 2: leave-one-out model. 3: single variable model. 4: dubbelsteenmodel
# Note that only 1 supports sensitivity_analysis
model=1
verbose=True

[settings]
# start year
y_start=2000
# end year
y_end=2012

[PROJ_files]
# cfg-files
proj_nr_1=./path/to/projection/settings_proj.cfg

[pre_calc]
# if nothing is specified, the XY array will be stored in output_dir
# if XY already pre-calculated, then provide path to npy-file
XY=

[extent]
shp=folder/with/polygons.shp

[conflict]
# either specify path to file or state 'download' to download latest PRIO/UCDP dataset
conflict_file=folder/with/conflict_data.csv
min_nr_casualties=1
# 1=state-based armed conflict. 2=non-state conflict. 3=one-sided violence
type_of_violence=1,2,3

[climate]
shp=folder/with/climate_zones.shp
# define either one or more classes (use abbreviations!) or specify nothing for not␣
→filtering
zones=
code2class=folder/with/classification_codes.txt
```

(continues on next page)

```
[data]
# specify the path to the nc-file, whether the variable shall be log-transformed (True,␣
↪False), and which statistical function should be applied
# these three settings need to be separated by a comma
# NOTE: variable name here needs to be identical with variable name in nc-file
# NOTE: only statistical functions supported by rasterstats are valid
precipitation=folder/with/precipitation_data.nc,False,mean
temperature=folder/with/temperature_data.nc,False,mean
population=folder/with/population_data.nc,True,sum

[machine_learning]
# choose from: MinMaxScaler, StandardScaler, RobustScaler, QuantileTransformer
scaler=QuantileTransformer
# choose from: NuSVC, KNeighborsClassifier, RFClassifier
model=RFClassifier
train_fraction=0.7
# number of repetitions
n_runs=10
```

Note: All paths for `input_dir`, `output_dir`, and in `[PROJ_files]` are relative to the location of the cfg-file.

Important: Empty spaces should be avoided in the cfg-file, besides for those lines commented out with '#'.

2.3.2 The sections

Here, the different sections are explained briefly.

[general]

`input_dir`: (relative) path to the directory where the input data is stored. This requires all input data to be stored in one main folder, sub-folders are possible.

`output_dir`: (relative) path to the directory where output will be stored. If the folder does not exist yet, it will be created. CoPro will automatically create the sub-folders _REF for output for the reference run, and _PROJ for output from the (various) projection runs.

`model`: the type of simulation to be run can be specified here. Currently, for different models are available:

1. 'all data': all variable values are used to fit the model and predict results.

2. 'leave one out': values of each variable are left out once, resulting in n-1 runs with n being the number of variables. This model can be used to identify the relative influence of one variable within the variable set/

3. 'single variables': each variable is used as sole predictor once. With this model, the explanatory power of each variable on its own can be assessed.

4. 'dubbelsteen': the relation between variables and conflict are abolished by shuffling the binary conflict data randomly. By doing so, the lower boundary of the model can be estimated.

Note: All model types except 'all_data' will be deprecated in a future release.

`verbose`: if True, additional messages will be printed.

[settings]

`y_start`: the start year of the reference run.

`y_end`: the end year of the reference run. The period between `y_start` and `y_end` will be used to train and test the model.

`y_proj`: the end year of the projection run. The period between `y_end` and `y_proj` will be used to make annual projections.

[PROJ_files]

A key section. Here, one (slightly different) cfg-file per projection needs to be provided. This way, multiple projection runs can be defined from within the "main" cfg-file.

The conversion is that the projection name is defined as value here. For example, the projections "SSP1" and "SSP2" would be defined as

```
SSP1=/path/to/ssp1.cfg
SSP2=/path/to/ssp2.cfg
```

A cfg-file for a projection is shorter than the main cfg-file used as command line argument and looks like this:

```
[general]
input_dir=./path/to/input_data
verbose=True

[settings]
# year for which projection is to be made
y_proj=2050

[data]
# specify the path to the nc-file, whether the variable shall be log-transformed (True,
→False), and which statistical function should be applied
# these three settings need to be separated by a comma
# NOTE: variable name here needs to be identical with variable name in nc-file
# NOTE: only statistical functions supported by rasterstats are valid
precipitation=folder/with/precipitation_data.nc,False,mean
temperature=folder/with/temperature_data.nc,False,mean
population=folder/with/population_data.nc,True,sum
```

[pre_calc]

`XY`: if the XY-data was already pre-computed in a previous run and stored as npy-file, it can be specified here and will be loaded from file to save time. If nothing is specified, the model will save the XY-data by default to the output directory as `XY.npy`.

[extent]

`shp`: the provided shape-file defines the boundaries for which the model is applied. At the same time, it also defines at which aggregation level the output is determined.

Note: The shp-file can contain multiple polygons covering the study area. Their size defines the output aggregation level. It is also possible to provide only one polygon, but model behaviour is not well tested for this case.

[conflict]

`conflict_file`: path to the csv-file containing the conflict dataset. It is also possible to define `download`, then the latest conflict dataset (currently version 20.1) is downloaded and used as input.

`min_nr_casualties`: minimum number of reported casualties required for a conflict to be considered in the model.

`type_of_violence`: the types of violence to be considered can be specified here. Multiple values can be specified. Types of violence are:

1. 'state-based armed conflict': a contested incompatibility that concerns government and/or territory where the use of armed force between two parties, of which at least one is the government of a state, results in at least 25 battle-related deaths in one calendar year.

2. 'non-state conflict': the use of armed force between two organized armed groups, neither of which is the government of a state, which results in at least 25 battle-related deaths in a year.

3. 'one-sided violence': the deliberate use of armed force by the government of a state or by a formally organized group against civilians which results in at least 25 deaths in a year.

Important: CoPro currently only works with UCDP data.

[climate]

`shp`: the provided shape-file defines the areas of the different Köppen-Geiger climate zones.

`zones`: abbreviations of the climate zones to be considered in the model. Can either be 'None' or one or multiple abbreviations.

`code2class`: converting the abbreviations to class-numbers used in the shp-file.

Warning: The code2class-file should not be altered!

[data]

In this section, all variables to be used in the model need to be provided. The paths are relative to `input_dir`. Only netCDF-files with annual data are supported.

The main convention is that the name of the file agrees with the variable name in the file. For example, if the variable `precipitation` is provided in a nc-file, this should be noted as follows

```
[data]
precipitation=folder/with/precipitation_data.nc
```

CoPro furthermore requires information whether the values sampled from a file are ought to be log-transformed.

Besides, it is possible to define a statistical function that is applied when sampling from file per polygon of the `shp-file`. CoPro makes use of the `zonal_stats` function available within rasterstats.

To determine the log-scaled mean value of precipitation per polygon, the following notation is required:

```
[data]
precipitation=folder/with/precipitation_data.nc,False,mean
```

[machine_learning]

`scaler`: the scaling algorithm used to scale the variable values to comparable scales. Currently supported are

- MinMaxScaler;
- StandardScaler;
- RobustScaler;
- QuantileTransformer.

`model`: the machine learning algorithm to be applied. Currently supported are

- NuSVC;
- KNeighborsClassifier;
- RFClassifier.

`train_fraction`: the fraction of the XY-data to be used to train the model. The remaining data (1-train_fraction) will be used to predict and evaluate the model.

`n_runs`: the number of classifiers to use.

2.4 Workflow

This page provides a short example workflow in Jupyter Notebooks. It is designed such that the main steps, features, assumptions, and outcomes of CoPro become clear.

As model input data, the data set downloadable from Zenodo was used.

Even though the model can be perfectly executed using notebooks, the main (and more convenient) way of model execution is the command line script (see *Runner script*).

An interactive version of the content shown here can be accessed via Binder.

2.4.1 Model initialization and selection procedure

In this notebook, we will show how CoPro is initialized and the selection procedure of spatial aggregation units and conflicts works.

Model initialization

Start with loading the required packages.

```
[1]: from copro import utils, selection, plots, data

     %matplotlib inline

     import matplotlib.pyplot as plt
     import numpy as np
     import seaborn as sns
     import os, sys
     import warnings
     warnings.simplefilter("ignore")
```

For better reproducibility, the version numbers of all key packages used to run this notebook are provided.

```
[2]: utils.show_versions()
```

```
Python version: 3.7.8 | packaged by conda-forge | (default, Jul 31 2020, 01:53:57) [MSC
→v.1916 64 bit (AMD64)]
copro version: 0.0.8
geopandas version: 0.9.0
xarray version: 0.15.1
rasterio version: 1.1.0
pandas version: 1.0.3
numpy version: 1.18.1
scikit-learn version: 0.23.2
matplotlib version: 3.2.1
seaborn version: 0.11.0
rasterstats version: 0.14.0
```

The configurations-file (cfg-file)

In the configurations-file (cfg-file), all the settings for the analysis are defined. The cfg-file contains, amongst others, all paths to input files, settings for the machine-learning model, and the various selection criteria for spatial aggregation units and conflicts. Note that the cfg-file can be stored anywhere, not per se in the same directory where the model data is stored (as in this example case). Make sure that the paths in the cfg-file are updated if you use relative paths and change the folder location of th cfg-file!

```
[3]: settings_file = 'example_settings.cfg'
```

Based on this cfg-file, the set-up of the run can be initialized. Here, the cfg-file is parsed (i.e. read) and all settings and paths become 'known' to the model. Also, the output folder is created (if it does not exist yet) and the cfg-file is copied to the output folder for improved reusability.

If you set `verbose=True`, then additional statements are printed during model execution. This can help to track the behaviour of the model.

```
[4]: main_dict, root_dir = utils.initiate_setup(settings_file, verbose=False)
```

```
#### CoPro version 0.0.8 ####
#### For information about the model, please visit https://copro.readthedocs.io/ ####
#### Copyright (2020-2021): Jannis M. Hoch, Sophie de Bruin, Niko Wanders ####
#### Contact via: j.m.hoch@uu.nl ####
#### The model can be used and shared under the MIT license ####

INFO: reading model properties from example_settings.cfg
INFO: verbose mode on: False
INFO: saving output to main folder C:\Users\hoch0001\Documents\_code\copro\example\./OUT
```

One of the outputs is a dictionary (here `main_dict`) containing the parsed configurations (they are stored in computer memory, therefore the slighly odd specification) as well as output directories of both the reference run and the various projection runs specified in the cfg-file.

For the reference run, only the respective entries are required.

```
[5]: config_REF = main_dict['_REF'][0]
print('the configuration of the reference run is {}'.format(config_REF))
out_dir_REF = main_dict['_REF'][1]
print('the output directory of the reference run is {}'.format(out_dir_REF))
```

```
the configuration of the reference run is <configparser.RawConfigParser object at␣
→0x000001FBC8FA9D08>
the output directory of the reference run is C:\Users\hoch0001\Documents\_code\copro\
→example\./OUT\_REF
```

Filter conflicts and spatial aggregation units

Background

As conflict database, we use the UCDP Georeferenced Event Dataset. Not all conflicts of the database may need to be used for a simulation. This can be, for example, because they belong to a non-relevant type of conflict we are not interested in, or because it is simply not in our area-of-interest. Therefore, it is possible to filter the conflicts on various properties:

1. *min_nr_casualties: minimum number of casualties of a reported conflict;*

2. *type_of_violence: 1=state-based armed conflict; 2=non-state conflict; 3=one-sided violence.*

To unravel the interplay between climate and conflict, it may be beneficial to run the model only for conflicts in particular climate zones. It is hence also possible to select only those conflcits that fall within a climate zone following the Koeppen-Geiger classification.

Selection procedure

In the selection procedure, we first load the conflict database and convert it to a georeferenced dataframe (geo-dataframe). To define the study area, a shape-file containing polygons (in this case water provinces) is loaded and converted to geo-dataframe as well.

We then apply the selection criteria (see above) as specified in the cfg-file, and keep the remaining data points and associated polygons.

```
[7]: conflict_gdf, selected_polygons_gdf, global_df = selection.select(config_REF, out_dir_
     ↪REF, root_dir)

     INFO: reading csv file to dataframe C:\Users\hoch0001\Documents\_code\copro\example\./
     ↪example_data\UCDP/ged201.csv
     INFO: filtering based on conflict properties.
```

With the chosen settings, the following picture of polygons and conflict data points is obtained.

```
[8]: fig, ax= plt.subplots(1, 1, figsize=(20,10))
     conflict_gdf.plot(ax=ax, c='r', column='best', cmap='magma',
                       vmin=int(config_REF.get('conflict', 'min_nr_casualties')),
     ↪vmax=conflict_gdf.best.mean(),
                       legend=True,
                       legend_kwds={'label': "# casualties", 'orientation': "vertical", 'pad':
     ↪ 0.05})
     selected_polygons_gdf.boundary.plot(ax=ax);
```
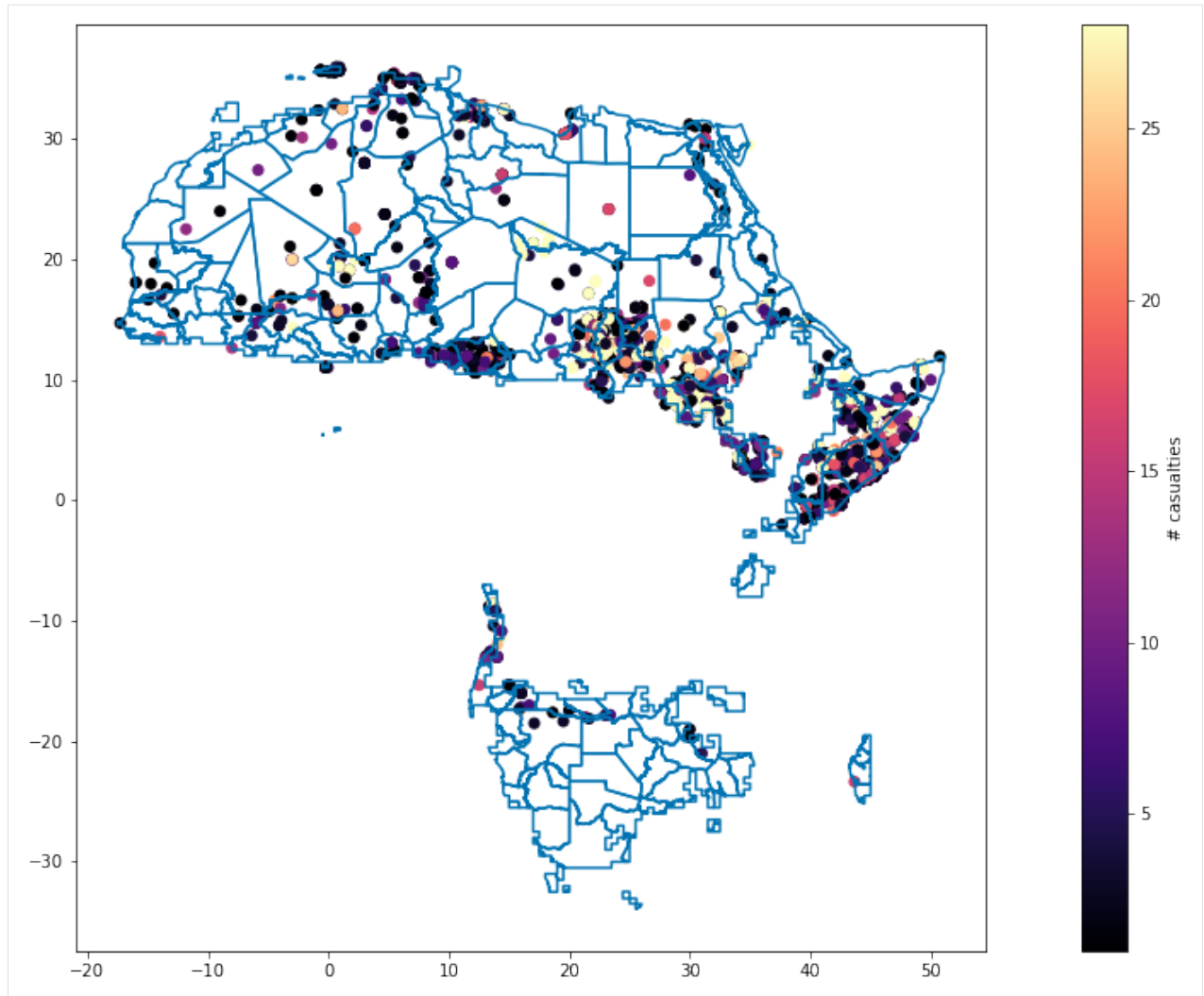
It's nicely visible that for this example-run, not all provinces are considered but we focus on specified climate zones only.

Temporary files

To be able to also run the following notebooks, some of the data has to be written to file temporarily. This is **not** part of the CoPro workflow but merely needed to split up the workflow in different notebooks outlining the main steps to go through when using CoPro.

```
[9]: if not os.path.isdir('temp_files'):
         os.makedirs('temp_files')
```

```
[10]: conflict_gdf.to_file(os.path.join('temp_files', 'conflicts.shp'))
      selected_polygons_gdf.to_file(os.path.join('temp_files', 'polygons.shp'))
```

```
[11]: global_df['ID'] = global_df.index.values
      global_arr = global_df.to_numpy()
      np.save(os.path.join('temp_files', 'global_df'), global_arr)
```

2.4.2 Obtaining samples matrix and target values

In this notebook, we will show how CoPro reads the input data and derives the samples matrix and target values needed to establish a machine-learning model.

Preparations

Start with loading the required packages.

```
[1]: from copro import utils, pipeline, data

     %matplotlib inline

     import matplotlib.pyplot as plt
     import pandas as pd
     import geopandas as gpd
     import os, sys
     import warnings
     from shutil import copyfile
     warnings.simplefilter("ignore")
```

For better reproducibility, the version numbers of all key packages used to run this notebook are provided.

```
[2]: utils.show_versions()
```

```
Python version: 3.7.8 | packaged by conda-forge | (default, Jul 31 2020, 01:53:57) [MSC
→v.1916 64 bit (AMD64)]
copro version: 0.0.8
geopandas version: 0.9.0
xarray version: 0.15.1
rasterio version: 1.1.0
pandas version: 1.0.3
numpy version: 1.18.1
scikit-learn version: 0.23.2
matplotlib version: 3.2.1
seaborn version: 0.11.0
rasterstats version: 0.14.0
```

To be able to also run this notebooks, some of the previously saved temporary files need to be loaded.

```
[3]: conflict_gdf = gpd.read_file(os.path.join('temp_files', 'conflicts.shp'))
     selected_polygons_gdf = gpd.read_file(os.path.join('temp_files', 'polygons.shp'))
```

The configurations-file (cfg-file)

To be able to continue the simulation with the same settings as in the previous notebook, the cfg-file has to be read again and the model needs to be initialised subsequently. This is not needed if CoPro is run from command line. Please see the previous notebook for additional information.

```
[4]: settings_file = 'example_settings.cfg'
```

```
[5]: main_dict, root_dir = utils.initiate_setup(settings_file, verbose=False)
```

```
#### CoPro version 0.0.8 ####
#### For information about the model, please visit https://copro.readthedocs.io/ ####
#### Copyright (2020-2021): Jannis M. Hoch, Sophie de Bruin, Niko Wanders ####
#### Contact via: j.m.hoch@uu.nl ####
#### The model can be used and shared under the MIT license ####

INFO: reading model properties from example_settings.cfg
INFO: verbose mode on: False
INFO: saving output to main folder C:\Users\hoch0001\Documents\_code\copro\example\./OUT
```

```
[6]: config_REF = main_dict['_REF'][0]
     out_dir_REF = main_dict['_REF'][1]
```

Reading the files and storing the data

Background

This is an essential part of CoPro. For a machine-learning model to work, it requires a samples matrix (X), representing here the socio-economic and hydro-climatic 'drivers' of conflict, and target values (Y) representing the (observed) conflicts themselves. By fitting a machine-learning model, a relation between X and Y is established, which in turn can be used to make projections.

Additional information can be found on scikit-learn.

Since CoPro simulates conflict risk spatially explicit for each polygons (here water provinces), it is furthermore needed to be able to associate each polygon with the corresponding data points in X and Y. We therefore also keep a polygon-ID and its geometry, and track it throughout the modelling chain.

Implementation

CoPro goes through all model years as specified in the cfg-file. Per year, CoPro loops over all polygons remaining after the selection procedure (see previous notebook) and does the following to obtain the **X-data**.

1. *Assign ID to polygon and retrieve geometry information;*

2. *Calculate a statistical value per polygon from each input file specified in the cfg-file in section 'data'. Which statistical value is ought to be computed needs to be specified in the cfg-file. In the cfg-file, it's furthermore possible to specify whether values are ought to be log-transformed.*

Note that CoPro applies a 1 year time-lag by default. That means that for a given year *J*, the data from year *J-1* is read. This is to avoid issues with backwards reversibility, i.e. we assume that the driver results in conflict (or not) with one year delay and not immediately in the same year.

And to obtain the **Y-data**:

1. *Assign a Boolean value whether a conflict took place in a polygon or not - the number of casualties or conflicts per year is not relevant in thise case.*

All information is stored in a X-array and a Y-array. The X-array has *2+n* columns whereby *n* denotes the number of samples provided. The Y-array has obviously only 1 column, consisting of zeros and ones. In both arrays is the number of rows determined as number of years times the number of polygons. In case a row contains a missing value (e.g. because one input data does not cover this polygon), the entire row is removed from the XY-array.

Note that the sample values can still range a lot depending on their units, measurement, etc. In the next notebook, the X-data will be scaled to be able to compare the different values in the samples matrix.

```
[7]: X, Y = pipeline.create_XY(config_REF, out_dir_REF, root_dir, selected_polygons_gdf,
     ↪conflict_gdf)
```

```
INFO: reading data for period from 2000 to 2012
INFO: skipping first year 2000 to start up model
INFO: entering year 2001
INFO: entering year 2002
INFO: entering year 2003
INFO: entering year 2004
INFO: entering year 2005
INFO: entering year 2006
INFO: entering year 2007
INFO: entering year 2008
INFO: entering year 2009
INFO: entering year 2010
INFO: entering year 2011
INFO: entering year 2012
```

Saving data to file

Depending on sample and file size, obtaining the X-array and Y-array can be time-consuming. Therefore, CoPro automatically stores a combined XY-array as `npy-file` to the output folder if not specified otherwise in the cfg-file. With this file, future runs using the same data but maybe different machine-learning settings can be executed in less time.

Let's check if this files exists.

```
[8]: os.path.isfile(os.path.join(out_dir_REF, 'XY.npy'))
```

```
[8]: True
```

Temporary files

By default, a binary map of conflict per polygon for the last year of the simulation period is stored to the output directory. Since the output directory is created from scratch at each model initalisation, we need to temporarily store this map in another folder to be used in subsequent notebooks.

```
[9]: %%capture

     for root, dirs, files in os.walk(os.path.join(out_dir_REF, 'files')):
         for file in files:
             fname = file
             print(fname)
             copyfile(os.path.join(out_dir_REF, 'files', str(fname)),
                     os.path.join('temp_files', str(fname)))
```

2.4.3 Initializing and executing the machine-learning model

In this notebook, we will show how CoPro creates, trains, and tests a machine-learning model based on the settings and data shown in the previous notebooks.

Preparations

Start with loading the required packages.

```
[1]: from copro import utils, pipeline, evaluation, plots, machine_learning

     %matplotlib inline

     import matplotlib.pyplot as plt
     import numpy as np
     import pandas as pd
     import geopandas as gpd
     import seaborn as sbs
     import os, sys
     from sklearn import metrics
     from shutil import copyfile
     import warnings
     warnings.simplefilter("ignore")
```

For better reproducibility, the version numbers of all key packages used to run this notebook are provided.

```
[2]: utils.show_versions()
```
```
Python version: 3.7.8 | packaged by conda-forge | (default, Jul 31 2020, 01:53:57) [MSC
→v.1916 64 bit (AMD64)]
copro version: 0.0.8
geopandas version: 0.9.0
xarray version: 0.15.1
rasterio version: 1.1.0
pandas version: 1.0.3
numpy version: 1.18.1
scikit-learn version: 0.23.2
matplotlib version: 3.2.1
seaborn version: 0.11.0
rasterstats version: 0.14.0
```

To be able to also run this notebooks, some of the previously saved data needs to be loaded.

```
[3]: conflict_gdf = gpd.read_file(os.path.join('temp_files', 'conflicts.shp'))
     selected_polygons_gdf = gpd.read_file(os.path.join('temp_files', 'polygons.shp'))
```

```
[4]: global_arr = np.load(os.path.join('temp_files', 'global_df.npy'), allow_pickle=True)
     global_df = pd.DataFrame(data=global_arr, columns=['geometry', 'ID'])
     global_df.set_index(global_df.ID, inplace=True)
     global_df.drop(['ID'] , axis=1, inplace=True)
```

The configurations-file (cfg-file)

To be able to continue the simulation with the same settings as in the previous notebook, the cfg-file has to be read again and the model needs to be initialised subsequently. This is not needed if CoPro is run from command line. Please see the first notebook for additional information.

```
[5]: settings_file = 'example_settings.cfg'
```

```
[6]: main_dict, root_dir = utils.initiate_setup(settings_file, verbose=False)
```

```
#### CoPro version 0.0.8 ####
#### For information about the model, please visit https://copro.readthedocs.io/ ####
#### Copyright (2020-2021): Jannis M. Hoch, Sophie de Bruin, Niko Wanders ####
#### Contact via: j.m.hoch@uu.nl ####
#### The model can be used and shared under the MIT license ####

INFO: reading model properties from example_settings.cfg
INFO: verbose mode on: False
INFO: saving output to main folder C:\Users\hoch0001\Documents\_code\copro\example\./OUT
```

```
[7]: config_REF = main_dict['_REF'][0]
     out_dir_REF = main_dict['_REF'][1]
```

Loading the XY-data

To avoid reading the XY-data again (see previous notebook for this), we can load the data directly from a `XY.npy` file which is automatically written to the output path. We saw that this file was created, but since no XY-data is specified in the config-file initially, we have to set the path manually. Note that this de-tour is only necessary due to the splitting of the workflow in different notebooks!

```
[8]: config_REF.set('pre_calc', 'XY', str(os.path.join(out_dir_REF, 'XY.npy')))
```

To double-check, see if the file manually specifid actually exists.

```
[9]: os.path.isfile(config_REF.get('pre_calc', 'XY'))
```

```
[9]: True
```

The scence is set now and we can read the X-array and Y-array from file.

```
[10]: X, Y = pipeline.create_XY(config_REF, out_dir_REF, root_dir, selected_polygons_gdf,␣
      ↪conflict_gdf)
```

```
INFO: loading XY data from file C:\Users\hoch0001\Documents\_code\copro\example\./OUT\_
↪REF\XY.npy
```

Scaler and classifier

Background

In principle, one can put all kind of data into the samples matrix X, leading to a wide spread of orders of magnitude, units, distributions etc. It is therefore needed to scale (or transform) the data in the X-array such that sensible comparisons and computations are possible. To that end, a scaling technique is applied.

Once there is a scaled X-array, a machine-learning model can be fitted with it together with the target values Y.

Implementation

CoPro supports four different scaling techniques. For more info, see the scikit-learn documentation.

1. MinMaxScaler;

2. StandardScaler;

3. RobustScaler;

4. QuantileTransformer.

From the wide range of machine-learning model, CoPro employs three different ones from the categorie of supervised learning.

1. NuSVC;

2. KNeighborsClassifier;

3. RFClassifier.

Note that CoPro uses pretty much the default parameterization of the scalers and models. An extensive GridSearchCV did not show any significant improvements when changing the parameters. There is currently no way to provide other parameters than those currently set.

Let's see which scaling technique and which supervised classifiers is specified for the example-run.

```
[11]: scaler, clf = pipeline.prepare_ML(config_REF)
      print('As scaling technique, it is used: {}'.format(scaler))
      print('As supervised classifing technique, it is used: {}'.format(clf))
```

```
As scaling technique, it is used: QuantileTransformer(random_state=42)
As supervised classifing technique, it is used: RandomForestClassifier(class_weight={1:
→100}, n_estimators=1000,
                       random_state=42)
```

Output initialization

Since the model is run multiple times to test various random train-test data combinations, we need to initialize a few lists first to append the output per run.

```
[12]: out_X_df = evaluation.init_out_df()
      out_y_df = evaluation.init_out_df()
```

```
[13]: out_dict = evaluation.init_out_dict()
```

```
[14]: trps, aucs, mean_fpr = evaluation.init_out_ROC_curve()
```

ML-model execution

The pudels kern! This is where the magic happens, and not only once. To make sure that any conincidental results are ruled out, we run the model multiple times. Thereby, always different parts of the XY-array are used for training and testing. By using a sufficient number of runs and averaging the overall results, we should be able to get a good picture of what the model is capable of. The number of runs as well as the split between training and testing data needs to be specified in the cfg-file.

Per repetition, the model is evaluated. The main evaluation metric is the mean ROC-score and **ROC-curve**, plotted at the end of all runs. Additional evaluation metrics are computed as described below.

```
[15]: # #- create plot instance
      fig, (ax1) = plt.subplots(1, 1, figsize=(20,10))

      #- go through all n model executions
      for n in range(config_REF.getint('machine_learning', 'n_runs')):

          print('INFO: run {} of {}'.format(n+1, config_REF.getint('machine_learning', 'n_runs
      ↪')))

          #- run machine learning model and return outputs
          X_df, y_df, eval_dict = pipeline.run_reference(X, Y, config_REF, scaler, clf, out_
      ↪dir_REF, run_nr=n+1)

          #- select sub-dataset with only datapoints with observed conflicts
          X1_df, y1_df = utils.get_conflict_datapoints_only(X_df, y_df)

          #- append per model execution
          out_X_df = evaluation.fill_out_df(out_X_df, X_df)
          out_y_df = evaluation.fill_out_df(out_y_df, y_df)
          out_dict = evaluation.fill_out_dict(out_dict, eval_dict)

          #- plot ROC curve per model execution
          tprs, aucs = plots.plot_ROC_curve_n_times(ax1, clf, X_df.to_numpy(), y_df.y_test.to_
      ↪list(),
                                                    trps, aucs, mean_fpr)

      #- plot mean ROC curve
      plots.plot_ROC_curve_n_mean(ax1, tprs, aucs, mean_fpr)

      plt.savefig('../docs/_static/roc_curve.png', dpi=300, bbox_inches='tight')
```

```
INFO: run 1 of 10
```

```
No handles with labels found to put in legend.
```

```
INFO: run 2 of 10
```

```
No handles with labels found to put in legend.
```

```
INFO: run 3 of 10
```

```
No handles with labels found to put in legend.
```

| INFO: run 4 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 5 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 6 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 7 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 8 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 9 of 10 |
| No handles with labels found to put in legend. |
| INFO: run 10 of 10 |
| No handles with labels found to put in legend. |
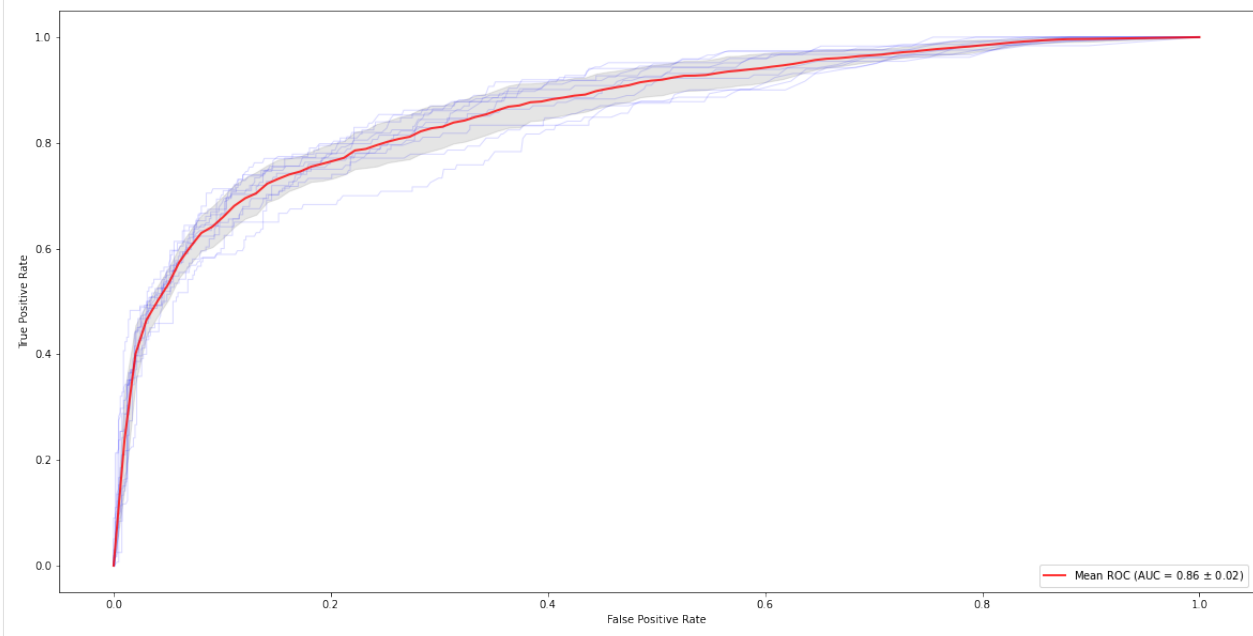
Model evaluation

For all data points

During the model runs, the computed model evaluation scores per model execution were stored to a dictionary. Currently, the evaluation scores used are:

- **Accuracy**: the fraction of correct predictions;
- **Precision**: the ratio $tp / (tp + fp)$ where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative;

- ****Recall****: the ratio *tp / (tp + fn)* where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples;

- ****F1 score****: the F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0;

- ****Cohen-Kappa score****: is used to measure inter-rater reliability. It is generally thought to be a more robust measure than simple percent agreement calculation, as takes into account the possibility of the agreement occurring by chance.

- ****Brier score****: the smaller the Brier score, the better, hence the naming with "loss". The lower the Brier score is for a set of predictions, the better the predictions are calibrated. Note that the Brier loss score is relatively sensitive for imbalanced datasets;

- ****ROC score****: a value of 0.5 suggests no skill, e.g. a curve along the diagonal, whereas a value of 1.0 suggests perfect skill, all points along the left y-axis and top x-axis toward the top left corner. A value of 0.0 suggests perfectly incorrect predictions. Note that the ROC score is relatively insensitive for imbalanced datasets.

- ****AP score****: the average_precision_score function computes the average precision (AP) from prediction scores. The value is between 0 and 1 and higher is better.

Let's check the mean scores over all runs:

```
[16]: for key in out_dict:

    print('average {0} of run with {1} repetitions is {2:0.3f}'.format(key, config_REF.
    ↪getint('machine_learning', 'n_runs'), np.mean(out_dict[key])))
```

```
average Accuracy of run with 10 repetitions is 0.883
average Precision of run with 10 repetitions is 0.717
average Recall of run with 10 repetitions is 0.496
average F1 score of run with 10 repetitions is 0.585
average Cohen-Kappa score of run with 10 repetitions is 0.520
average Brier loss score of run with 10 repetitions is 0.090
average ROC AUC score of run with 10 repetitions is 0.863
average AP score of run with 10 repetitions is 0.636
```

So how are, e.g. accuracy, precision, and recall distributed?

```
[17]: plots.metrics_distribution(out_dict, metrics=['Accuracy', 'Precision', 'Recall'],␣
    ↪figsize=(20, 5));
```
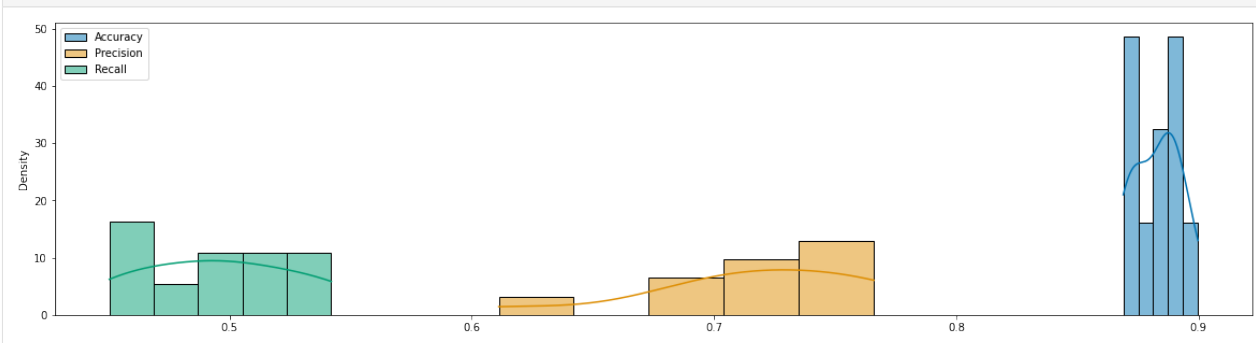

Based on all data points, the ****confusion matrix**** can be plotted. This is a relatively straightforward way to visualize how good (i.e. correct) the observations are predicted by the model. Ideally, all `True label` and `Predicted label` pairs have the highest values.

```
[18]: fig, ax = plt.subplots(1, 1, figsize=(8, 8))
      metrics.plot_confusion_matrix(clf, out_X_df.to_numpy(), out_y_df.y_test.to_list(),
      ↪ax=ax);
```
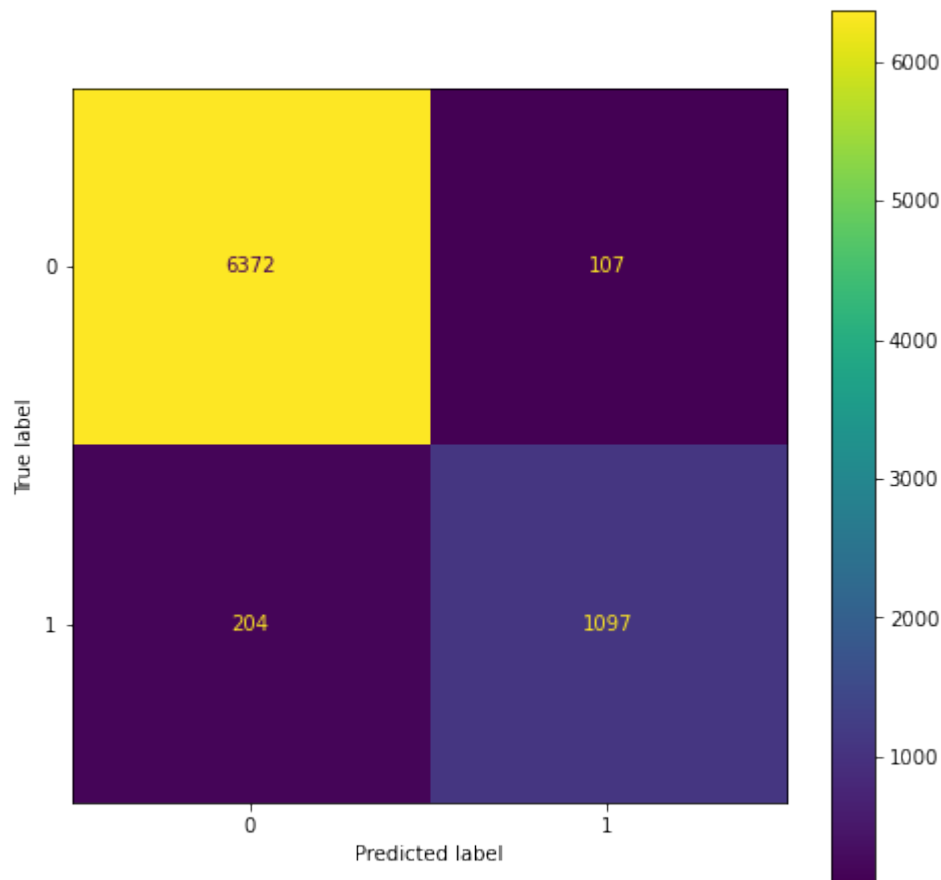

In `out_y_df`, all predictions are stored. This includes the actual value `y_test` (ie. whether a conflict was observed or not) and the predicted outcome `y_pred` together with the probabilities of this outcome. Additionally, CoPro adds a column with a Boolean indicator whether the predictions was correct (`y_test=y_pred`) or not.

```
[19]: out_y_df.head()
```

```
[19]:       ID                                           geometry y_test y_pred  \
      0   1009   POLYGON ((29 6.696147705436432, 29.05159624587...      0      0
      1   1525   POLYGON ((0.5770535604073238 6, 0.578418291470...      0      0
      2   1307   (POLYGON ((-14.94260162796269 16.6312412609754...      0      0
      3    118   POLYGON ((25.29046121561265 -18.03749999982506...      0      0
      4     45   POLYGON ((9.821052248962189 28.22336190952456,...      0      0

         y_prob_0 y_prob_1   correct_pred
      0     0.989    0.011              1
      1     0.998    0.002              1
      2     0.976    0.024              1
      3     0.914    0.086              1
      4     0.966    0.034              1
```

Per unique polygon

Thus far, we merely looked at numerical scores for all predcitions. This of course tells us a lot about the quality of the machine-learning model, but not so much about how this looks like spatially. We therefore combine the observations and predictions made with the associated polygons based on a 'global' dataframe functioning as a look-up table. By this means, each model prediction (ie. each row in `out_y_df`) can be connected to its polygon using a unique polygon-ID.

```
[20]: df_hit, gdf_hit = evaluation.polygon_model_accuracy(out_y_df, global_df)
```

First, let's have a look at how often each polygon occurs in the all test samples, i.e. those obtained by appending the test samples per model execution. Besides, the overall relative distribution is visualized.

```
[21]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
      gdf_hit.plot(ax=ax1, column='nr_predictions', legend=True, cmap='Blues')
      selected_polygons_gdf.boundary.plot(ax=ax1, color='0.5')
      ax1.set_title('number of predictions made per polygon')
      sbs.distplot(df_hit.nr_predictions.values, ax=ax2)
      ax2.set_title('distribution of predictions');
```
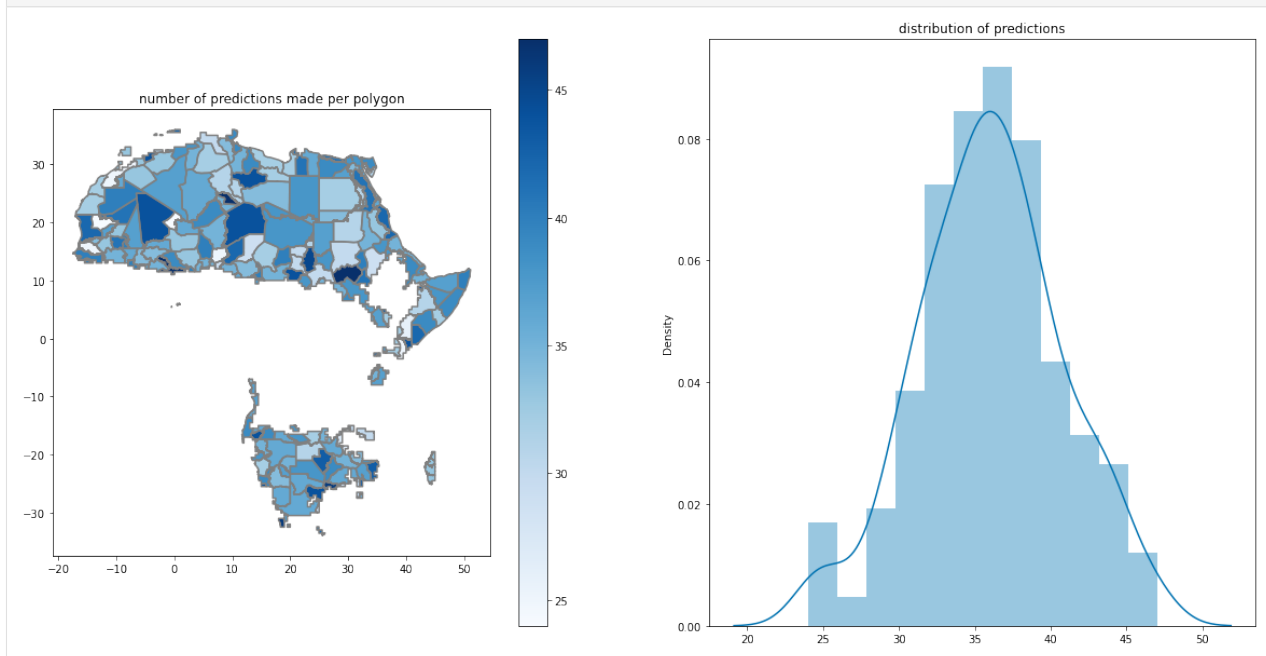

By repeating the model *n* times, the aim is to represent all polygons in the resulting test sample. The fraction is computed below.

Note that is should be close to 100 % but may be slightly less. This can happen if input variables have no data for one polygon, leading to a removal of those polygons from the analysis. Or because some polygons and input data may not overlap.

```
[22]: print('{0:0.2f} % of all active polygons are considered in test sample'.format(len(gdf_
      ↪hit)/len(selected_polygons_gdf)*100))
```

```
100.00 % of all active polygons are considered in test sample
```

By aggregating results per polygon, we can now assess model output spatially. Three main aspects are presented here:

1. The total number of conflict events per water province;

2. The chance of a correct prediction, defined as the ratio of number of correct predictions made to overall number of predictions made;

3. The mean conflict probability, defined as the mean value of all probabilites of conflict to occur (y_prob_1) in a polygon.

```
[24]: fig, axes = plt.subplots(1, 3, figsize=(20, 20), sharex=True, sharey=True)

gdf_hit.plot(ax=axes[0], column='nr_observed_conflicts', legend=True, cmap='Reds',
             legend_kwds={'label': "nr_observed_conflicts", 'orientation': "horizontal",
→'fraction': 0.045, 'pad': 0.05})
selected_polygons_gdf.boundary.plot(ax=axes[0], color='0.5')

gdf_hit.plot(ax=axes[1], column='fraction_correct_predictions', legend=True,
             legend_kwds={'label': "fraction_correct_predictions", 'orientation':
→"horizontal", 'fraction': 0.045, 'pad': 0.05})
selected_polygons_gdf.boundary.plot(ax=axes[1], color='0.5')

gdf_hit.plot(ax=axes[2], column='probability_of_conflict', legend=True, cmap='Blues',␣
→vmin=0, vmax=1,
             legend_kwds={'label': "mean conflict probability", 'orientation':
→"horizontal", 'fraction': 0.045, 'pad': 0.05})
selected_polygons_gdf.boundary.plot(ax=axes[2], color='0.5')

plt.tight_layout();
```
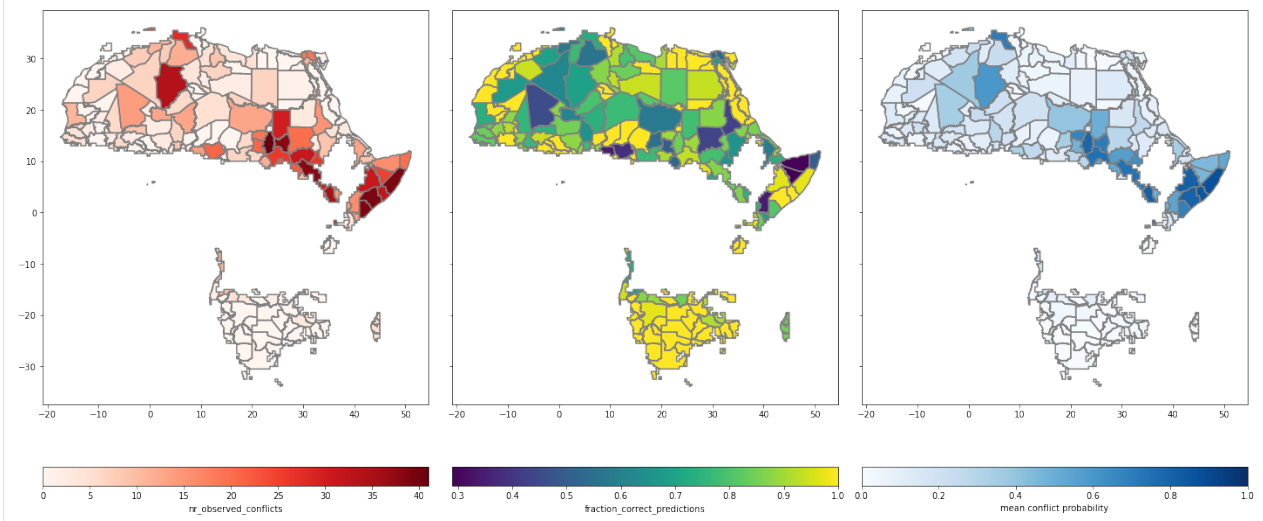

Preparing for projections

In this notebook, we have trained and tested our model with various combinations of data. Subsequently, the average performance of the model was evaluated with a range of metrics.

If we want to re-use our model for the future and want to make projections, it is necessary to save the model (that is, the *n* fitted classifiers). They can then be loaded and one or more projections can be made with other variable values than those used for this reference run.

To that end, the classifier is fitted again, but then with all data, i.e. without a split-sample test. That way, the classifier fit is most robust.

```
[25]: %%capture

      for root, dirs, files in os.walk(os.path.join(out_dir_REF, 'clfs')):
          for file in files:
              fname = file
              print(fname)
              copyfile(os.path.join(out_dir_REF, 'clfs', str(fname)),
                       os.path.join('temp_files', str(fname)))
```

```
[ ]:
```

2.4.4 Projecting conflict risk

In this notebook, we will show how CoPro uses a number of previously fitted classifiers and projects conflict risk forward in time. Eventually, these forward predictions based on multiple classifiers can be merged into a robust estimate of future conflict risk.

Preparations

Start with loading the required packages.

```
[1]: from copro import utils, pipeline, evaluation, plots, machine_learning

     %matplotlib inline

     import matplotlib.pyplot as plt
     import numpy as np
     import pandas as pd
     import geopandas as gpd
     import seaborn as sbs
     import os, sys
     from sklearn import metrics
     from shutil import copyfile
     import warnings
     import glob
     warnings.simplefilter("ignore")
```

For better reproducibility, the version numbers of all key packages are provided.

```
[2]: utils.show_versions()
```

```
Python version: 3.7.8 | packaged by conda-forge | (default, Jul 31 2020, 01:53:57) [MSC
→v.1916 64 bit (AMD64)]
copro version: 0.0.8
geopandas version: 0.9.0
xarray version: 0.15.1
rasterio version: 1.1.0
pandas version: 1.0.3
numpy version: 1.18.1
scikit-learn version: 0.23.2
matplotlib version: 3.2.1
```

```
seaborn version: 0.11.0
rasterstats version: 0.14.0
```

To be able to also run this notebooks, some of the previously saved data needs to be loaded from a temporary location.

```
[3]: conflict_gdf = gpd.read_file(os.path.join('temp_files', 'conflicts.shp'))
     selected_polygons_gdf = gpd.read_file(os.path.join('temp_files', 'polygons.shp'))
```

```
[4]: global_arr = np.load(os.path.join('temp_files', 'global_df.npy'), allow_pickle=True)
     global_df = pd.DataFrame(data=global_arr, columns=['geometry', 'ID'])
     global_df.set_index(global_df.ID, inplace=True)
     global_df.drop(['ID'] , axis=1, inplace=True)
```

The configurations-file (cfg-file)

To be able to continue the simulation with the same settings as in the previous notebook, the cfg-file has to be read again and the model needs to be initialised subsequently. This is not needed if CoPro is run from command line. Please see the first notebook for additional information.

```
[5]: settings_file = 'example_settings.cfg'
```

```
[6]: main_dict, root_dir = utils.initiate_setup(settings_file, verbose=False)


     #### CoPro version 0.0.8 ####
     #### For information about the model, please visit https://copro.readthedocs.io/ ####
     #### Copyright (2020-2021): Jannis M. Hoch, Sophie de Bruin, Niko Wanders ####
     #### Contact via: j.m.hoch@uu.nl ####
     #### The model can be used and shared under the MIT license ####

     INFO: reading model properties from example_settings.cfg
     INFO: verbose mode on: False
     INFO: saving output to main folder C:\Users\hoch0001\Documents\_code\copro\example\./OUT
```

```
[7]: config_REF = main_dict['_REF'][0]
     out_dir_REF = main_dict['_REF'][1]
```

In addition to the config-object and output path for the reference period, `main_dict` also contains the equivalents for the projection run. In the cfg-file, an extra cfg-file can be provided per projection.

```
[8]: config_REF.items('PROJ_files')
```

```
[8]: [('proj_nr_1', './example_settings_proj.cfg')]
```

In this example, the files is called `example_settings_proj.cfg` and the name of the projection is `proj_nr_1`.

```
[9]: config_PROJ = main_dict['proj_nr_1'][0]
     print('the configuration of the projection run is {}'.format(config_PROJ))
     out_dir_PROJ = main_dict['proj_nr_1'][1]
     print('the output directory of the projection run is {}'.format(out_dir_PROJ))
```

```
the configuration of the projection run is [<configparser.RawConfigParser object at␣
→0x0000021E18A03508>]
the output directory of the projection run is C:\Users\hoch0001\Documents\_code\copro\
→example\./OUT\_PROJ\proj_nr_1
```

In the previous notebooks, conflict at the last year of the reference period as well as classifiers were stored temporarily
to another folder than the output folder. Now let's copy these files back to the folders where the belong.

```python
[10]: %%capture

      for root, dirs, files in os.walk('temp_files'):

          # conflicts at last time step
          files = glob.glob(os.path.abspath('./temp_files/conflicts_in*'))
          for file in files:
              fname = file.rsplit('\\')[-1]
              print(fname)
              copyfile(os.path.join('temp_files', fname),
                       os.path.join(out_dir_REF, 'files', str(fname)))

          # classifiers
          files = glob.glob(os.path.abspath('./temp_files/clf*'))
          for file in files:
              fname = file.rsplit('\\')[-1]
              print(fname)
              copyfile(os.path.join('temp_files', fname),
                       os.path.join(out_dir_REF, 'clfs', str(fname)))
```

Similarly, we need to load the sample data (X) for the reference run as we need to fit the scaler with this data before we
can make comparable and consistent projections.

```python
[11]: config_REF.set('pre_calc', 'XY', str(os.path.join(out_dir_REF, 'XY.npy')))
      X, Y = pipeline.create_XY(config_REF, out_dir_REF, root_dir, selected_polygons_gdf,␣
      →conflict_gdf)
```

```
INFO: loading XY data from file C:\Users\hoch0001\Documents\_code\copro\example\./OUT\_
→REF\XY.npy
```

Lastly, we need to get the scaler for the samples matrix again. The pre-computed and already fitted classifiers are
directly loaded from file (see above). The clf returned here will not be used.

```python
[12]: scaler, clf = pipeline.prepare_ML(config_REF)
```

Project!

With this all in place, we can now make projections. Under the hood, various steps are taken for each projectio run
specified:

1. Load the corresponding ConfigParser-object;

2. Determine the projection period defined as the period between last year of reference run and projection year
   specified in cfg-file of projection run;

3. Make a separate projection per classifier (the number of classifiers, or model runs, is specified in the cfg-file):

1. in the first year of the projection year, use conflict data from last year of reference run, i.e. still observed conflict data;

2. in all following year, use the conflict data projected for the previous year with this specific classifier;

3. all other variables are read from file for all years.

4. Per year, merge the conflict risk projected by all classifiers and derive a fractional conflict risk per polygon.

For detailed information, please see the documentatoin and code of `copro.pipeline.run_prediction()`. As this is one function doing all the work, it is not possible to split up the workflow in more detail here.

```
[13]: all_y_df = pipeline.run_prediction(scaler.fit(X[: , 2:]), main_dict, root_dir, selected_
      →polygons_gdf)
```

```
INFO: loading config-object for projection run: proj_nr_1
INFO: the projection period is 2013 to 2015
INFO: making projection for year 2013
INFO: making projection for year 2014
INFO: making projection for year 2015
```

Analysis of projection

All the previously used evaluation metrics are not applicable anymore, as there are no target values anymore. We can still look what the mean conflict probability is as computed by the model per polygon.

```
[15]: # link projection outcome to polygons via unique polygon-ID
      df_hit, gdf_hit = evaluation.polygon_model_accuracy(all_y_df, global_df, make_proj=True)

      # and plot
      fig, ax = plt.subplots(1, 1, figsize=(10, 10))
      gdf_hit.plot(ax=ax, column='probability_of_conflict', legend=True, figsize=(20, 10),
      →cmap='Blues', vmin=0, vmax=1,
              legend_kwds={'label': "mean conflict probability", 'orientation': "vertical",
      →'fraction': 0.045})
      selected_polygons_gdf.boundary.plot(ax=ax, color='0.5');
```
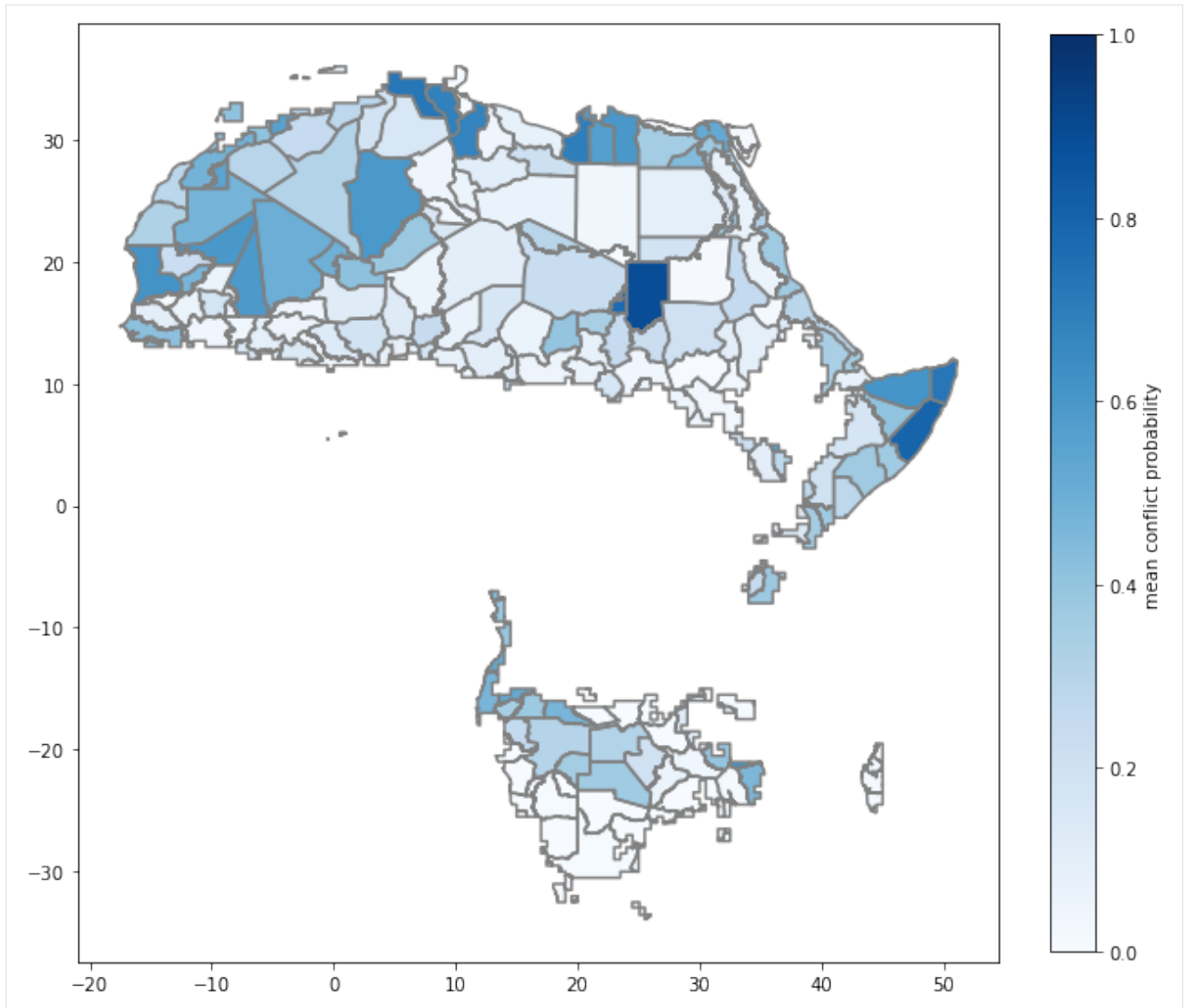
Projection output

The conflict projection per year is also stored in the output folder of the projection run as geoJSON files. These files can be used to post-process the data with the scripts provided with CoPro or to load them into bespoke scripts and functions written by the user.

2.5 Output

2.5.1 Output folder structure

All output is stored in the output folder as specified in the configurations-file (cfg-file) under [general].

```
[general]
output_dir=./path/to/store/output
```

By default, CoPro creates two sub-folders: _REF and _PROJ. In the latter, another sub-folder will be created per projection defined in the cfg-file. In the example below, this would be the folders /_PROJ/SSP1 and /_PROJ/SSP2.

```
[PROJ_files]
SSP1=/path/to/ssp1.cfg
SSP2=/path/to/ssp2.cfg
```

2.5.2 List of output files

Important: Not all model types provide the output mentioned below. If the 'leave-one-out' or 'single variable' model are selected, only the metrics are stored to a csv-file.

_REF

In addition to the output files listed below, the cfg-file is automatically copied to the _REF folder.

`selected_polygons.shp`: Shapefile containing all remaining polygons after selection procedure.

`selected_conflicts.shp`: Shapefile containing all remaining conflict points after selection procedure,

`XY.npy`: NumPy-array containing geometry, ID, and scaled data of sample (X) and target data (Y). Can be provided in cfg-file to safe time in next run; file can be loaded with numpy.load().

`raw_output_data.npy`: NumPy-array containing each single prediction made in the reference run. Will contain multiple predictions per polygon. File can be loaded with numpy.load().

`evaluation_metrics.csv`: Various evaluation metrics determined per repetition of the split-sample tests. File can e.g. be loaded with pandas.read_csv().

`feature_importance.csv`: Importance of each model variable in making projections. This is a property of RF Classifiers and thus only obtainable if RF Classifier is used.

`permutation_importance.csv`: Mean permutation importance per model variable. Computed with sklearn.inspection.permutation_importance.

`ROC_data_tprs.csv` and `ROC_data_aucs.csv`: False-positive rates respectively Area-under-curve values per repetition of the split-sample test. Files can e.g. be loaded with pandas.read_csv() and can be used to later plot ROC-curve.

`output_for_REF.geojson`: GeoJSON-file containing resulting conflict risk estimates per polygon based on out-of-sample projections of _REF run.

Conflict risk per polygon

At the end of all model repetitions, the resulting `raw_output_data.npy` file contains multiple out-of-sample predictions per polygon. By aggregating results per polygon, it is possible to assess model output spatially as stored in `output_for_REF.geojson`.

The main output metrics are calculated per polygon and saved to `output_per_polygon.shp`:

1. nr_predictions: the number of predictions made;

2. nr_correct_predictions: the number of correct predictions made;

3. nr_observed_conflicts: the number of observed conflict events;

4. nr_predicted_conflicts: the number of predicted conflicts;

5. min_prob_1: minimum probability of conflict in all repetitions;

6. probability_of_conflict (POC): probability of conflict averaged over all repetitions;

7. max_prob_1: maximum probability of conflict in all repetitions;

8. fraction_correct_predictions (FOP): ratio of the number of correct predictions over the total number of predictions made;

9. chance_of_conflict: ratio of the number of conflict predictions over the total number of predictions made.

_PROJ

Per projection, CoPro creates one output file per projection year.

`output_in_<YEAR>`: GeoJSON-file containing model output per polygon averaged over all classifier instances per YEAR of the projection. The number of instances is set with `n_runs` in `[machine_learning]` section.

Conflict risk per polygon

During the projection run, each classifier instances produces its own output per YEAR. CoPro merges these outputs into one `output_in_<YEAR>.geojson` file.

As there are no observations available for the projection period, the output metrics differ from the reference run:

1. nr_predictions: the number of predictions made, ie. number of classifier instances;

2. nr_predicted_conflicts: the number of predicted conflicts.

3. min_prob_1: minimum probability of conflict in all outputs of classifier instances.

4. probability_of_conflict (POC): probability of conflict averaged over all outputs of classifier instances.

5. max_prob_1: maximum probability of conflict in all outputs of classifier instances;

6. chance_of_conflict: ratio of the number of conflict predictions over the total number of predictions made.

2.6 Postprocessing

There are several command line scripts available for post-processing. In addition to quick plots to evaluate model output, they also produce files for use in bespoke plotting and analysis scripts.

The scripts are located under `/copro/scripts/postprocessing`.

The here shown help print-outs can always be accessed with

```
python <SCRIPT_FILE_NAME> --help
```

2.6.1 plot_value_over_time.py

```
Usage: python plot_value_over_time.py [OPTIONS] INPUT_DIR OUTPUT_DIR

    Quick and dirty function to plot the develoment of a column in the
    outputted geoJSON-files over time. The script uses all geoJSON-files
    located in input-dir and retrieves values from them. Possible to plot
    obtain development for multiple polygons (indicated via their ID) or
    entire study area. If the latter, then different statistics can be chosen
    (mean, max, min, std).

    Args:
        input-dir (str): path to input directory with geoJSON-files located per
→projection year.
        output-dir (str): path to directory where output will be stored.

    Output:
        a csv-file containing values per time step.
        a png-file showing development over time.

    Options:
        -id, --polygon-id TEXT
        -s, --statistics TEXT     which statistical method to use (mean, max, min,
                                    std). note: has only effect if with "-id all"!

        -c, --column TEXT         column name
        -t, --title TEXT          title for plot and file_object name
        --verbose / --no-verbose  verbose on/off
```

2.6.2 avg_over_time.py

```
Usage: python avg_over_time.py [OPTIONS] INPUT_DIR OUTPUT_DIR SELECTED_POLYGONS

    Post-processing script to calculate average model output over a user-
    specifeid period or all output geoJSON-files stored in input-dir.
    Computed average values can be outputted as geoJSON-file or png-file or both.

    Args:
        input_dir: path to input directory with geoJSON-files located per projection
→year.
        output_dir (str): path to directory where output will be stored.
        selected_polygons (str): path to a shp-file with all polygons used in a CoPro
→run.

    Output:
        geoJSON-file with average column value per polygon (if geojson is set).
        png-file with plot of average column value per polygon (if png is set)

    Options:
        -t0, --start-year INTEGER
        -t1, --end-year INTEGER
```

```
    -c, --column TEXT          column name
    --geojson / --no-geojson   save output to geojson or not
    --png / --no-png           save output to png or not
    --verbose / --no-verbose   verbose on/off
```

2.6.3 plot_polygon_vals.py

```
Usage: python plot_polygon_vals.py [OPTIONS] FILE_OBJECT OUTPUT_DIR

    Quick and dirty function to plot the column values of a geojson file with
    minimum user input, and save plot. Mainly used for quick inspection of
    model output in specific years.

    Args:
        file-object (str): path to geoJSON-file whose values are to be plotted.
        output-dir (str): path to directory where plot will be saved.

    Output:
        a png-file of values per polygon.

    Options:
        -c, --column TEXT          column name
        -t, --title TEXT           title for plot and file_object name
        -v0, --minimum-value FLOAT
        -v1, --maximum-value FLOAT
        -cmap, --color-map TEXT
```

2.6.4 geojson2gif.py

```
Usage: python geojson2gif.py [OPTIONS] INPUT_DIR OUTPUT_DIR

    Function to convert column values of all geoJSON-files in a directory into
    one GIF-file. The function provides several options to modify the design
    of the GIF-file. The GIF-file is based on png-files of column value per
    geoJSON-file.  It is possible to keep these png-file as simple plots of
    values per time step.

Args:
    input-dir (str): path to directory where geoJSON-files are stored.
    output_dir (str): path to directory where GIF-file will be stored.

Output:
    GIF-file with animated column values per input geoJSON-file.

Options:
    -c, --column TEXT          column name
    -cmap, --color-map TEXT
    -v0, --minimum-value FLOAT
```

```
    -v1, --maximum-value FLOAT
    --delete / --no-delete      whether or not to delete png-files
```

2.7 API docs

This section contains the Documentation of the Application Programming Interface (API) of 'copro'.

2.7.1 The model pipeline

pipeline.create_XY	Top-level function to create the X-array and Y-array.
pipeline.prepare_ML	Top-level function to instantiate the scaler and model as specified in model configurations.
pipeline.run_reference	Top-level function to run one of the four supported models.
pipeline.run_prediction	Top-level function to execute the projections.

copro.pipeline.create_XY

copro.pipeline.**create_XY**(*config*, *out_dir*, *root_dir*, *polygon_gdf*, *conflict_gdf*)

Top-level function to create the X-array and Y-array. If the XY-data was pre-computed and specified in cfg-file, the data is loaded. If not, variable values and conflict data are read from file and stored in array. The resulting array is by default saved as npy-format to file.

> **Parameters**
>
> - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> - **out_dir** (`str`) – path to output folder.
> - **root_dir** (`str`) – path to location of cfg-file.
> - **polygon_gdf** (`geo-dataframe`) – geo-dataframe containing the selected polygons.
> - **conflict_gdf** (`geo-dataframe`) – geo-dataframe containing the selected conflicts.
>
> **Returns** X-array containing variable values. array: Y-array containing conflict data.
>
> **Return type** array

copro.pipeline.prepare_ML

copro.pipeline.**prepare_ML**(*config*)

Top-level function to instantiate the scaler and model as specified in model configurations.

> **Parameters config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
>
> **Returns** the specified scaler instance. classifier: the specified model instance.
>
> **Return type** scaler

copro.pipeline.run_reference

copro.pipeline.**run_reference**(*X*, *Y*, *config*, *scaler*, *clf*, *out_dir*, *run_nr*)

Top-level function to run one of the four supported models.

> **Parameters**
> - **X** (`array`) – X-array containing variable values.
> - **Y** (`array`) – Y-array containing conflict data.
> - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> - **scaler** (`scaler`) – the specified scaler instance.
> - **clf** (`classifier`) – the specified model instance.
> - **out_dir** (`str`) – path to output folder.
>
> **Raises** `ValueError` – raised if unsupported model is specified.
>
> **Returns** containing the test-data X-array values. datatrame: containing model output on polygon-basis. dict: dictionary containing evaluation metrics per simulation.
>
> **Return type** dataframe

copro.pipeline.run_prediction

copro.pipeline.**run_prediction**(*scaler*, *main_dict*, *root_dir*, *selected_polygons_gdf*)

Top-level function to execute the projections. Per specified projection, conflict is projected forwards in time per time step until the projection year is reached. Pear time step, the sample data and conflict data are read individually since different conflict projections are made per classifier used. At the end of each time step, the projections of all classifiers are combined and output metrics determined.

> **Parameters**
> - **scaler** (`scaler`) – the specified scaler instance.
> - **main_dict** (`dict`) – dictionary containing config-objects and output directories for reference run and all projection runs.
> - **root_dir** (`str`) – path to location of cfg-file.
> - **selected_polygons_gdf** (`geo-dataframe`) –
>
> **Raises** `ValueError` – raised if another model type than the one using all data is specified in cfg-file.
>
> **Returns** containing model output on polygon-basis.
>
> **Return type** dataframe

2.7.2 The various models

`models.all_data`	Main model workflow when all XY-data is used.
`models.leave_one_out`	Model workflow when each variable is left out from analysis once.
`models.single_variables`	Model workflow when the model is based on only one single variable.
`models.dubbelsteen`	Model workflow when the relation between variables and conflict is based on randomness.
`models.predictive`	Predictive model to use the already fitted classifier to make annual projections for the projection period.

copro.models.all_data

copro.models.**all_data**(*X*, *Y*, *config*, *scaler*, *clf*, *out_dir*, *run_nr*)

> Main model workflow when all XY-data is used. The model workflow is executed for each classifier.
>
> **Parameters**
>
> - **X** (*array*) – array containing the variable values plus IDs and geometry information.
>
> - **Y** (*array*) – array containing merely the binary conflict classifier data.
>
> - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
>
> - **scaler** (*scaler*) – the specified scaling method instance.
>
> - **clf** (*classifier*) – the specified model instance.
>
> - **out_dir** (*str*) – path to output folder.
>
> **Returns** containing the test-data X-array values. datatrame: containing model output on polygon-basis. dict: dictionary containing evaluation metrics per simulation.
>
> **Return type** dataframe

copro.models.leave_one_out

copro.models.**leave_one_out**(*X*, *Y*, *config*, *scaler*, *clf*, *out_dir*)

> Model workflow when each variable is left out from analysis once. Output is limited to the metric scores. Output is stored to sub-folders of the output directory, with each sub-folder containing output for a n-1 variable combination. After computing metric scores per prediction (i.e. n-1 variables combinations), model exit is forced. Not tested yet for more than one simulation!
>
> **Parameters**
>
> - **X** (*array*) – array containing the variable values plus unique identifer and geometry information.
>
> - **Y** (*array*) – array containing merely the binary conflict classifier data.
>
> - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
>
> - **scaler** (*scaler*) – the specified scaling method instance.
>
> - **clf** (*classifier*) – the specified model instance.

> • **out_dir** (`str`) – path to output folder.

> **Raises DeprecationWarning** – this function will most likely be deprecated due to lack of added value and applicability.

copro.models.single_variables

copro.models.**single_variables**(*X*, *Y*, *config*, *scaler*, *clf*, *out_dir*)

> Model workflow when the model is based on only one single variable. Output is limited to the metric scores. Output is stored to sub-folders of the output directory, with each sub-folder containing output for a 1 variable combination. After computing metric scores per prediction (i.e. per variable), model exit is forced. Not tested yet for more than one simulation!

> **Parameters**

> > • **X** (`array`) – array containing the variable values plus unique identifer and geometry information.

> > • **Y** (`array`) – array containing merely the binary conflict classifier data.

> > • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.

> > • **scaler** (`scaler`) – the specified scaling method instance.

> > • **clf** (`classifier`) – the specified model instance.

> > • **out_dir** (`str`) – path to output folder.

> **Raises DeprecationWarning** – this function will most likely be deprecated due to lack of added value and applicability.

copro.models.dubbelsteen

copro.models.**dubbelsteen**(*X*, *Y*, *config*, *scaler*, *clf*, *out_dir*)

> Model workflow when the relation between variables and conflict is based on randomness. Thereby, the fraction of actual conflict is equal to observations, but the location in array is randomized by shuffling. The model workflow is executed for each model simulation.

> **Parameters**

> > • **X** (`array`) – array containing the variable values plus unique identifer and geometry information.

> > • **Y** (`array`) – array containing merely the binary conflict classifier data.

> > • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.

> > • **scaler** (`scaler`) – the specified scaling method instance.

> > • **clf** (`classifier`) – the specified model instance.

> > • **out_dir** (`str`) – path to output folder.

> **Returns** containing the test-data X-array values. datatrame: containing model output on polygon-basis. dict: dictionary containing evaluation metrics per simulation.

> **Return type** dataframe

copro.models.predictive

copro.models.**predictive**(*X*, *clf*, *scaler*, *config*)
> Predictive model to use the already fitted classifier to make annual projections for the projection period. As other models, it reads data which are then scaled and used in conjuction with the classifier to project conflict risk.

> **Parameters**
>> • **X** (`array`) – array containing the variable values plus unique identifer and geometry information.
>>
>> • **clf** (`classifier`) – the fitted specified classifier instance.
>>
>> • **scaler** (`scaler`) – the fitted specified scaling method instance.
>>
>> • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model for a projection run.

> **Returns** containing model output on polygon-basis.

> **Return type** datatrame

Note: The 'leave_one_out', 'single_variables', and 'dubbelsteen' models are only tested in beta-state. They will most likely be deprecated in near future.

2.7.3 Selecting polygons and conflicts

selection.select	Main function performing the selection procedure.
selection.filter_conflict_properties	Filters conflict database according to certain conflict properties such as number of casualties, type of violence or country.
selection.select_period	Reducing the geo-dataframe to those entries falling into a specified time period.
selection.clip_to_extent	As the original conflict data has global extent, this function clips the database to those entries which have occured on a specified continent.
selection.climate_zoning	This function allows for selecting only those conflicts and polygons falling in specified climate zones.

copro.selection.select

copro.selection.**select**(*config*, *out_dir*, *root_dir*)
> Main function performing the selection procedure. Also stores the selected conflicts and polygons to output directory.

> **Parameters**
>> • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
>>
>> • **out_dir** (`str`) – path to output folder.
>>
>> • **root_dir** (`str`) – path to location of cfg-file.

> **Returns** remaining conflict data after selection process. geo-dataframe: all polygons of the study

area. geo-dataframe: remaining polygons after selection process. dataframe: global look-up
dataframe linking polygon ID with geometry information.

> **Return type** geo-dataframe

copro.selection.filter_conflict_properties

copro.selection.**filter_conflict_properties**(*gdf*, *config*)
> Filters conflict database according to certain conflict properties such as number of casualties, type of violence or
> country.

> > **Parameters**
> >
> > - **gdf** (*geo-dataframe*) – geo-dataframe containing entries with conflicts.
> >
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of
> >   the model.
> >
> > **Returns** geo-dataframe containing filtered entries.
> >
> > **Return type** geo-dataframe

copro.selection.select_period

copro.selection.**select_period**(*gdf*, *config*)
> Reducing the geo-dataframe to those entries falling into a specified time period.

> > **Parameters**
> >
> > - **gdf** (*geo-dataframe*) – geo-dataframe containing entries with conflicts.
> >
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of
> >   the model.
> >
> > **Returns** geo-dataframe containing filtered entries.
> >
> > **Return type** geo-dataframe

copro.selection.clip_to_extent

copro.selection.**clip_to_extent**(*gdf*, *config*, *root_dir*)
> As the original conflict data has global extent, this function clips the database to those entries which have occured
> on a specified continent.

> > **Parameters**
> >
> > - **gdf** (*geo-dataframe*) – geo-dataframe containing entries with conflicts.
> >
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of
> >   the model.
> >
> > - **root_dir** (*str*) – path to location of cfg-file.
> >
> > **Returns** geo-dataframe containing filtered entries. geo-dataframe: geo-dataframe containing coun-
> > try polygons of selected continent.
> >
> > **Return type** geo-dataframe

copro.selection.climate_zoning

copro.selection.**climate_zoning**(*gdf*, *extent_gdf*, *config*, *root_dir*)

> This function allows for selecting only those conflicts and polygons falling in specified climate zones. Also, a global dataframe is returned containing the IDs and geometry of all polygons after selection procedure. This can be used to add geometry information to model output based on common ID.

> > **Parameters**
> >
> > - **gdf** (`geo-dataframe`) – geo-dataframe containing conflict data.
> >
> > - **extent_gdf** (`geo-dataframe`) – all polygons of study area.
> >
> > - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> >
> > - **root_dir** (`str`) – path to location of cfg-file.
> >
> > **Returns** conflict data clipped to climate zones. geo-dataframe: polygons of study area clipped to climate zones.
> >
> > **Return type** geo-dataframe

2.7.4 Machine learning

machine_learning.define_scaling	Defines scaling method based on model configurations.
machine_learning.define_model	Defines model based on model configurations.
machine_learning.split_scale_train_test_split	Splits and transforms the X-array (or sample data) and Y-array (or target data) in test-data and training-data.
machine_learning.fit_predict	Fits classifier based on training-data and makes predictions.
machine_learning.pickle_clf	(Re)fits a classifier with all available data and pickles it.
machine_learning.load_clfs	Loads the paths to all previously fitted classifiers to a list.

copro.machine_learning.define_scaling

copro.machine_learning.**define_scaling**(*config*)

> Defines scaling method based on model configurations.

> > **Parameters config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> >
> > **Raises ValueError** – raised if a non-supported scaling method is specified.
> >
> > **Returns** the specified scaling method instance.
> >
> > **Return type** scaler

copro.machine_learning.define_model

copro.machine_learning.**define_model**(*config*)

> Defines model based on model configurations. Model parameter were optimized beforehand using Grid-SearchCV.
>
> > **Parameters config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> >
> > **Raises ValueError** – raised if a non-supported model is specified.
> >
> > **Returns** the specified model instance.
> >
> > **Return type** classifier

copro.machine_learning.split_scale_train_test_split

copro.machine_learning.**split_scale_train_test_split**(*X*, *Y*, *config*, *scaler*)

> Splits and transforms the X-array (or sample data) and Y-array (or target data) in test-data and training-data. The fraction of data used to split the data is specified in the configuration file. Additionally, the unique identifier and geometry of each data point in both test-data and training-data is retrieved in separate arrays.
>
> > **Parameters**
> >
> > - **X** (*array*) – array containing the variable values plus unique identifer and geometry information.
> > - **Y** (*array*) – array containing merely the binary conflict classifier data.
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> > - **scaler** (*scaler*) – the specified scaling method instance.
> >
> > **Returns** arrays containing training-set and test-set for X-data and Y-data as well as IDs and geometry.
> >
> > **Return type** arrays

copro.machine_learning.fit_predict

copro.machine_learning.**fit_predict**(*X_train*, *y_train*, *X_test*, *clf*, *config*, *out_dir*, *run_nr*)

> Fits classifier based on training-data and makes predictions. The fitted classifier is dumped to file with pickle to be used again during projections. Makes prediction with test-data including probabilities of those predictions.
>
> > **Parameters**
> >
> > - **X_train** (*array*) – training-data of variable values.
> > - **y_train** (*array*) – training-data of conflict data.
> > - **X_test** (*array*) – test-data of variable values.
> > - **clf** (*classifier*) – the specified model instance.
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> > - **out_dir** (*path*) – path to output folder.
> > - **run_nr** (*int*) – number of fit/predict repetition and created classifier.
> >
> > **Returns** arrays including the predictions made and their probabilities

> **Return type** arrays

copro.machine_learning.pickle_clf

copro.machine_learning.**pickle_clf**(*scaler*, *clf*, *config*, *root_dir*)
> (Re)fits a classifier with all available data and pickles it.

> > **Parameters**

> > > - **scaler** (*scaler*) – the specified scaling method instance.
> > > - **clf** (*classifier*) – the specified model instance.
> > > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> > > - **root_dir** (*str*) – path to location of cfg-file.

> > **Returns** classifier fitted with all available data.

> > **Return type** classifier

copro.machine_learning.load_clfs

copro.machine_learning.**load_clfs**(*config*, *out_dir*)
> Loads the paths to all previously fitted classifiers to a list. Classifiers were saved to file in fit_predict(). With this list, the classifiers can be loaded again during projections.

> > **Parameters**

> > > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> > > - **out_dir** (*path*) – path to output folder.

> > **Returns** list with file names of classifiers.

> > **Return type** list

2.7.5 Variable values

variables.nc_with_float_timestamp	This function extracts a value from a netCDF-file (specified in the cfg-file) for each polygon specified in extent_gdf for a given year.
variables.nc_with_continous_datetime_timestamp	This function extracts a value from a netCDF-file (specified in the cfg-file) for each polygon specified in extent_gdf for a given year.

copro.variables.nc_with_float_timestamp

copro.variables.**nc_with_float_timestamp**(*extent_gdf*, *config*, *root_dir*, *var_name*, *sim_year*)

This function extracts a value from a netCDF-file (specified in the cfg-file) for each polygon specified in extent_gdf for a given year. In the cfg-file, it must also be specified whether the value is log-transformed or not, and which statistical method is applied.

NOTE: The key in the cfg-file must be identical to variable name in netCDF-file.

NOTE: This function is specifically written for netCDF-files where the time variable contains integer (year-)values, e.g. 1995, 1996, …

NOTE: Works only with nc-files with annual data.

> **Parameters**
>
> > - **extent_gdf** (`geodataframe`) – geo-dataframe containing one or more polygons with geometry information for which values are extracted.
> > - **config** (`config`) – parsed configuration settings of run.
> > - **root_dir** (`str`) – path to location of cfg-file.
> > - **var_name** (`str`) – name of variable in nc-file, must also be the same under which path to nc-file is specified in cfg-file.
> > - **sim_year** (`int`) – year for which data is extracted.
>
> **Raises**
>
> > - **ValueError** – raised if not everything is specified in cfg-file.
> > - **ValueError** – raised if the extracted variable at a time step does not contain data.
>
> **Returns** list containing statistical value per polygon, i.e. with same length as extent_gdf.
>
> **Return type** list

copro.variables.nc_with_continous_datetime_timestamp

copro.variables.**nc_with_continous_datetime_timestamp**(*extent_gdf*, *config*, *root_dir*, *var_name*, *sim_year*)

This function extracts a value from a netCDF-file (specified in the cfg-file) for each polygon specified in extent_gdf for a given year. In the cfg-file, it must also be specified whether the value is log-transformed or not, and which statistical method is applied.

NOTE: The key in the cfg-file must be identical to variable name in netCDF-file.

NOTE: Works only with nc-files with annual data.

> **Parameters**
>
> > - **extent_gdf** (`geodataframe`) – geo-dataframe containing one or more polygons with geometry information for which values are extracted
> > - **config** (`config`) – parsed configuration settings of run.
> > - **root_dir** (`str`) – path to location of cfg-file.
> > - **var_name** (`str`) – name of variable in nc-file, must also be the same under which path to nc-file is specified in cfg-file.
> > - **sim_year** (`int`) – year for which data is extracted.
>
> **Raises**

- **ValueError** – raised if not everything is specified in cfg-file.

- **ValueError** – raised if specfied year cannot be found in years in nc-file.

- **ValueError** – raised if the extracted variable at a time step does not contain data.

Returns list containing statistical value per polygon, i.e. with same length as extent_gdf.

Return type list

> **Warning:** Reading files with a float timestamp will most likely be deprecated in near future.

2.7.6 XY-Data

`data.initiate_XY_data`	Initiates an empty dictionary to contain the XY-data for each polygon, ie.
`data.initiate_X_data`	Initiates an empty dictionary to contain the X-data for each polygon, ie.
`data.fill_XY`	Fills the (XY-)dictionary with data for each variable and conflict for each polygon for each simulation year.
`data.fill_X_sample`	Fills the X-dictionary with the data sample data besides any conflict-related data for each polygon and each year.
`data.fill_X_conflict`	Fills the X-dictionary with the conflict data for each polygon and each year.
`data.split_XY_data`	Separates the XY-array into array containing information about variable values (X-array or sample data) and conflict data (Y-array or target data).
`data.neighboring_polys`	For each polygon, determines its neighboring polygons.
`data.find_neighbors`	Filters all polygons which are actually neighbors to given polygon.

copro.data.initiate_XY_data

copro.data.**initiate_XY_data**(*config*)

Initiates an empty dictionary to contain the XY-data for each polygon, ie. both sample data and target data. This is needed for the reference run. By default, the first column is for the polygon ID, the second for polygon geometry. The antepenultimate column is for boolean information about conflict at t-1 while the penultimate column is for boolean information about conflict at t-1 in neighboring polygons. The last column is for binary conflict data at t (i.e. the target data).

Every column in between corresponds to the variables provided in the cfg-file.

> **Parameters** **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
>
> **Returns** emtpy dictionary to be filled, containing keys for each variable (X), binary conflict data (Y) plus meta-data.
>
> **Return type** dict

copro.data.initiate_X_data

copro.data.**initiate_X_data**(*config*)

Initiates an empty dictionary to contain the X-data for each polygon, ie. only sample data. This is needed for each time step of each projection run. By default, the first column is for the polygon ID and the second for polygon geometry. The penultimate column is for boolean information about conflict at t-1 while the last column is for boolean information about conflict at t-1 in neighboring polygons. All remaining columns correspond to the variables provided in the cfg-file.

> **Parameters config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
>
> **Returns** emtpy dictionary to be filled, containing keys for each variable (X) plus meta-data.
>
> **Return type** dict

copro.data.fill_XY

copro.data.**fill_XY**(*XY*, *config*, *root_dir*, *conflict_data*, *polygon_gdf*, *out_dir*)

Fills the (XY-)dictionary with data for each variable and conflict for each polygon for each simulation year. The number of rows should therefore equal to number simulation years times number of polygons. At end of last simulation year, the dictionary is converted to a numpy-array.

> **Parameters**
>
> - **XY** (*dict*) – initiated, i.e. empty, XY-dictionary
> - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> - **root_dir** (*str*) – path to location of cfg-file.
> - **conflict_data** (*geo-dataframe*) – geo-dataframe containing the selected conflicts.
> - **polygon_gdf** (*geo-dataframe*) – geo-dataframe containing the selected polygons.
> - **out_dir** (*path*) – path to output folder.
>
> **Raises Warning** – raised if the datetime-format of the netCDF-file does not match conventions and/or supported formats.
>
> **Returns** filled array containing the variable values (X) and binary conflict data (Y) plus meta-data.
>
> **Return type** array

copro.data.fill_X_sample

copro.data.**fill_X_sample**(*X*, *config*, *root_dir*, *polygon_gdf*, *proj_year*)

Fills the X-dictionary with the data sample data besides any conflict-related data for each polygon and each year. Used during the projection runs as the sample and conflict data need to be treated separately there.

> **Parameters**
>
> - **X** (*dict*) – dictionary containing keys to be sampled.
> - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> - **root_dir** (*str*) – path to location of cfg-file of reference run.
> - **polygon_gdf** (*geo-dataframe*) – geo-dataframe containing the selected polygons.

> • **proj_year** (`int`) – year for which projection is made.

> **Raises** `Warning` – raised if the datetime-format of the netCDF-file does not match conventions and/or
> supported formats.

> **Returns** dictionary containing sample values.

> **Return type** dict

copro.data.fill_X_conflict

copro.data.**fill_X_conflict**(*X*, *config*, *conflict_data*, *polygon_gdf*)

> Fills the X-dictionary with the conflict data for each polygon and each year. Used during the projection runs as
> the sample and conflict data need to be treated separately there.

> **Parameters**

> > • **X** (`dict`) – dictionary containing keys to be sampled.
> >
> > • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of
> > the model.
> >
> > • **conflict_data** (`dataframe`) – dataframe containing all polygons with conflict.
> >
> > • **polygon_gdf** (`geo-dataframe`) – geo-dataframe containing the selected polygons.

> **Returns** dictionary containing sample and conflict values.

> **Return type** dict

copro.data.split_XY_data

copro.data.**split_XY_data**(*XY*, *config*)

> Separates the XY-array into array containing information about variable values (X-array or sample data) and
> conflict data (Y-array or target data). Thereby, the X-array also contains the information about unique identifier
> and polygon geometry.

> **Parameters**

> > • **XY** (`array`) – array containing variable values and conflict data.
> >
> > • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of
> > the model.

> **Returns** two separate arrays, the X-array and Y-array.

> **Return type** arrays

copro.data.neighboring_polys

copro.data.**neighboring_polys**(*config*, *extent_gdf*, *identifier='watprovID'*)

> For each polygon, determines its neighboring polygons. As result, a (n x n) look-up dataframe is obtained con-
> taining, where n is number of polygons in extent_gdf.

> **Parameters**

> > • **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of
> > the model.
> >
> > • **extent_gdf** (`geo-dataframe`) – geo-dataframe containing the selected polygons.

- **identifier** (*str, optional*) – column name in extent_gdf to be used to identify neighbors. Defaults to 'watprovID'.

Returns look-up dataframe containing True/False statement per polygon for all other polygons.

Return type dataframe

copro.data.find_neighbors

copro.data.**find_neighbors**(*ID*, *neighboring_matrix*)
Filters all polygons which are actually neighbors to given polygon.

Parameters

- **ID** (*int*) – ID of specific polygon under consideration.

- **neighboring_matrix** (*dataframe*) – output from neighboring_polys().

Returns dataframe containig IDs of all polygons that are actual neighbors.

Return type dataframe

2.7.7 Work with conflict data

conflict.conflict_in_year_bool	Creates a list for each timestep with boolean information whether a conflict took place in a polygon or not.
conflict.conflict_in_previous_year	Creates a list for each timestep with boolean information whether a conflict took place in a polygon at the previous timestep or not.
conflict.read_projected_conflict	Creates a list for each timestep with boolean information whether a conflict took place in a polygon or not.
conflict.calc_conflicts_nb	Determines whether in the neighbouring polygons of a polygon i_poly conflict took place.
conflict.get_poly_ID	Extracts and returns a list with unique identifiers for each polygon used in the model.
conflict.get_poly_geometry	Extracts geometry information for each polygon from geodataframe and saves to list.
conflict.split_conflict_geom_data	Separates the unique identifier, geometry information, and data from the variable-containing X-array.
conflict.get_pred_conflict_geometry	Stacks together the arrays with unique identifier, geometry, test data, and predicted data into a dataframe.

copro.conflict.conflict_in_year_bool

copro.conflict.**conflict_in_year_bool**(*config*, *conflict_gdf*, *extent_gdf*, *sim_year*, *out_dir*)
Creates a list for each timestep with boolean information whether a conflict took place in a polygon or not.

Parameters

- **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.

- **conflict_gdf** (*geodataframe*) – geo-dataframe containing georeferenced information of conflict (tested with PRIO/UCDP data).

- **extent_gdf** (`geodataframe`) – geo-dataframe containing one or more polygons with ge-
  ometry information for which values are extracted.

- **sim_year** (`int`) – year for which data is extracted.

- **out_dir** (`str`) – path to output folder. If 'None', no output is stored.

Raises `AssertionError` – raised if the length of output list does not match length of input geo-
dataframe.

Returns list containing 0/1 per polygon depending on conflict occurence.

Return type list

copro.conflict.conflict_in_previous_year

copro.conflict.**conflict_in_previous_year**(*config*, *conflict_gdf*, *extent_gdf*, *sim_year*,
                             check_neighbors=False, *neighboring_matrix=None*)
Creates a list for each timestep with boolean information whether a conflict took place in a polygon at the previous
timestep or not. If the current time step is the first (t=0), then this year is skipped and the model continues at the
next time step.

    Parameters

- **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of
  the model.

- **conflict_gdf** (`geodataframe`) – geo-dataframe containing georeferenced information of
  conflict (tested with PRIO/UCDP data).

- **extent_gdf** (`geodataframe`) – geo-dataframe containing one or more polygons with ge-
  ometry information for which values are extracted.

- **sim_year** (`int`) – year for which data is extracted.

- **check_neighbors** (`bool`) – whether to check conflict events in neighboring polygons. De-
  faults to False.

- **neighboring_matrix** (`dataframe`) – lookup-dataframe indicating which polygons are
  mutual neighbors. Defaults to None.

    Raises

- `ValueError` – raised if check_neighbors is True, but no matrix is provided.

- `AssertionError` – raised if the length of output list does not match length of input geo-
  dataframe.

Returns list containing 0/1 per polygon depending on conflict occurence if checkinf for conflict at t-1,
and containing log-transformed number of conflict events in neighboring polygons if specified.

Return type list

copro.conflict.read_projected_conflict

copro.conflict.**read_projected_conflict**(*extent_gdf*, *bool_conflict*, *check_neighbors=False*, *neighboring_matrix=None*)

> Creates a list for each timestep with boolean information whether a conflict took place in a polygon or not. Input conflict data (bool_conflict) must contain an index with IDs corresponding with the 'watprovID' values of extent_gdf. Optionally, the algorithm can be extended to the neighboring polygons.
>
> > **Parameters**
> >
> > - **extent_gdf** (*geodataframe*) – geo-dataframe containing one or more polygons with geometry information for which values are extracted.
> >
> > - **bool_conflict** (*dataframe*) – dataframe with boolean values (1) for each polygon with conflict.
> >
> > - **check_neighbors** (*bool, optional*) – whether or not to check for conflict in neighboring polygons. Defaults to False.
> >
> > - **neighboring_matrix** (*dataframe, optional*) – look-up dataframe listing all neighboring polygons. Defaults to None.
> >
> > **Returns** containing 1 and 0 values for each polygon with conflict respectively without conflict. If check_neighbors=True, then 1 if neighboring polygon contains conflict and 0 is not.
> >
> > **Return type** list

copro.conflict.calc_conflicts_nb

copro.conflict.**calc_conflicts_nb**(*i_poly*, *neighboring_matrix*, *conflicts_per_poly*)

> Determines whether in the neighbouring polygons of a polygon i_poly conflict took place. If so, a value 1 is returned, otherwise 0.
>
> > **Parameters**
> >
> > - **i_poly** (*int*) – ID number of polygon under consideration.
> >
> > - **neighboring_matrix** (*dataframe*) – look-up dataframe listing all neighboring polygons.
> >
> > - **conflicts_per_poly** (*dataframe*) – dataframe with conflict informatoin per polygon.
> >
> > **Returns** 1 is conflict took place in neighboring polygon, 0 if not.
> >
> > **Return type** int

copro.conflict.get_poly_ID

copro.conflict.**get_poly_ID**(*extent_gdf*)

> Extracts and returns a list with unique identifiers for each polygon used in the model. The identifiers are currently limited to 'watprovID'.
>
> > **Parameters** **extent_gdf** (*geo-dataframe*) – geo-dataframe containing one or more polygons.
> >
> > **Raises** `AssertionError` – error raised if length of output list does not match length of input geodataframe.
> >
> > **Returns** list containing a unique identifier extracted from geo-dataframe for each polygon used in the model.
> >
> > **Return type** list

copro.conflict.get_poly_geometry

copro.conflict.**get_poly_geometry**(*extent_gdf*, *config*)

> Extracts geometry information for each polygon from geodataframe and saves to list. The geometry column in geodataframe must be named 'geometry'.
>
> > **Parameters**
> >
> > - **extent_gdf** (*geo-dataframe*) – geo-dataframe containing one or more polygons with geometry information.
> >
> > - **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.
> >
> > **Raises** **AssertionError** – error raised if length of output list does not match length of input geodataframe.
> >
> > **Returns** list containing the geometry information extracted from geo-dataframe for each polygon used in the model.
> >
> > **Return type** list

copro.conflict.split_conflict_geom_data

copro.conflict.**split_conflict_geom_data**(*X*)

> Separates the unique identifier, geometry information, and data from the variable-containing X-array.
>
> > **Parameters** **X** (*array*) – variable-containing X-array.
> >
> > **Returns** seperate arrays with ID, geometry, and actual data
> >
> > **Return type** arrays

copro.conflict.get_pred_conflict_geometry

copro.conflict.**get_pred_conflict_geometry**(*X_test_ID*, *X_test_geom*, *y_test*, *y_pred*, *y_prob_0*, *y_prob_1*)

> Stacks together the arrays with unique identifier, geometry, test data, and predicted data into a dataframe. Contains therefore only the data points used in the test-sample, not in the training-sample. Additionally computes whether a correct prediction was made.
>
> > **Parameters**
> >
> > - **X_test_ID** (*list*) – list containing the unique identifier per data point.
> >
> > - **X_test_geom** (*list*) – list containing the geometry per data point.
> >
> > - **y_test** (*list*) – list containing test-data.
> >
> > - **y_pred** (*list*) – list containing predictions.
> >
> > **Returns** dataframe with each input list as column plus computed 'correct_pred'.
> >
> > **Return type** dataframe

2.7.8 Model evaluation

`evaluation.init_out_dict`	Initiates the main model evaluatoin dictionary for a range of model metric scores.
`evaluation.fill_out_dict`	Appends the computed metric score per run to the main output dictionary.
`evaluation.init_out_df`	Initiates and empty main output dataframe.
`evaluation.fill_out_df`	Appends output dataframe of each simulation to main output dataframe.
`evaluation.evaluate_prediction`	Computes a range of model evaluation metrics and appends the resulting scores to a dictionary.
`evaluation.polygon_model_accuracy`	Determines a range of model accuracy values for each polygon.
`evaluation.init_out_ROC_curve`	Initiates empty lists for range of variables needed to plot ROC-curve per simulation.
`evaluation.save_out_ROC_curve`	Saves data needed to plot mean ROC and standard deviation to csv-files.
`evaluation.calc_correlation_matrix`	Computes the correlation matrix for a dataframe.
`evaluation.get_feature_importance`	Determines relative importance of each feature (i.e.
`evaluation.get_permutation_importance`	Returns a dataframe with the mean permutation importance of the features used to train a RF tree model.

copro.evaluation.init_out_dict

copro.evaluation.**init_out_dict**()

> Initiates the main model evaluatoin dictionary for a range of model metric scores. The scores should match the scores used in the dictioary created in 'evaluation.evaluate_prediction()'.
>
> > **Returns** empty dictionary with metrics as keys.
> >
> > **Return type** dict

copro.evaluation.fill_out_dict

copro.evaluation.**fill_out_dict**(*out_dict*, *eval_dict*)

> Appends the computed metric score per run to the main output dictionary. All metrics are initialized in init_out_dict().
>
> > **Parameters**
> >
> > - **out_dict** (*dict*) – main output dictionary.
> >
> > - **eval_dict** (*dict*) – dictionary containing scores per simulation.
> >
> > **Returns** dictionary with collected scores for each simulation
> >
> > **Return type** dict

copro.evaluation.init_out_df

copro.evaluation.**init_out_df**()
> Initiates and empty main output dataframe.
>
> > **Returns** empty dataframe.
> >
> > **Return type** dataframe

copro.evaluation.fill_out_df

copro.evaluation.**fill_out_df**(*out_df*, *y_df*)
> Appends output dataframe of each simulation to main output dataframe.
>
> > **Parameters**
> >
> > - **out_df** (`dataframe`) – main output dataframe.
> >
> > - **y_df** (`dataframe`) – output dataframe of each simulation.
> >
> > **Returns** main output dataframe containing results of all simulations.
> >
> > **Return type** dataframe

copro.evaluation.evaluate_prediction

copro.evaluation.**evaluate_prediction**(*y_test*, *y_pred*, *y_prob*, *X_test*, *clf*, *config*)
> Computes a range of model evaluation metrics and appends the resulting scores to a dictionary. This is done for each model execution separately. Output will be stored to stderr if possible.
>
> > **Parameters**
> >
> > - **y_test** (`list`) – list containing test-sample conflict data.
> >
> > - **y_pred** (`list`) – list containing predictions.
> >
> > - **y_prob** (`array`) – array resulting probabilties of predictions.
> >
> > - **X_test** (`array`) – array containing test-sample variable values.
> >
> > - **clf** (`classifier`) – sklearn-classifier used in the simulation.
> >
> > - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> >
> > **Returns** dictionary with scores for each simulation
> >
> > **Return type** dict

copro.evaluation.polygon_model_accuracy

copro.evaluation.**polygon_model_accuracy**(*df*, *global_df*, *make_proj=False*)
> Determines a range of model accuracy values for each polygon. Reduces dataframe with results from each simulation to values per unique polygon identifier. Determines the total number of predictions made per polygon as well as fraction of correct predictions made for overall and conflict-only data.
>
> > **Parameters**
> >
> > - **df** (`dataframe`) – output dataframe containing results of all simulations.

- **global_df** (`dataframe`) – global look-up dataframe to associate unique identifier with geometry.

- **make_proj** (`bool, optional`) – whether or not this function is used to make a projection. If True, a couple of calculations are skipped as no observed data is available for projections. Defaults to 'False'.

Returns dataframe and geo-dataframe with data per polygon.

Return type (geo-)dataframe

copro.evaluation.init_out_ROC_curve

copro.evaluation.**init_out_ROC_curve**()
Initiates empty lists for range of variables needed to plot ROC-curve per simulation.

Returns empty lists for variables.

Return type lists

copro.evaluation.save_out_ROC_curve

copro.evaluation.**save_out_ROC_curve**(*tprs*, *aucs*, *out_dir*)
Saves data needed to plot mean ROC and standard deviation to csv-files. They can be loaded again with pandas in a post-processing step.

Parameters

- **tprs** (`list`) – list with false positive rates.

- **aucs** (`list`) – list with area-under-curve values.

- **out_dir** (`str`) – path to output folder. If 'None', no output is stored.

copro.evaluation.calc_correlation_matrix

copro.evaluation.**calc_correlation_matrix**(*df*, *out_dir=None*)
Computes the correlation matrix for a dataframe. The dataframe should only contain numeric values.

Parameters

- **df** (`dataframe`) – dataframe with analysed output per polygon.

- **out_dir** (`str`) – path to output folder. If 'None', no output is stored. Default to 'None'.

Returns dataframe containig correlation matrix.

Return type dataframe

copro.evaluation.get_feature_importance

copro.evaluation.**get_feature_importance**(*clf*, *config*, *out_dir*)

Determines relative importance of each feature (i.e. variable) used. Must be used after model/classifier is fit. Returns dataframe and saves it to csv too.

> **Parameters**
>
> - **clf** (`classifier`) – sklearn-classifier used in the simulation.
> - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.
> - **out_dir** (`str`) – path to output folder. If 'None', no output is stored.
>
> **Raises** `Warning` – raised if the chosen ML model has not built-in feature importances.
>
> **Returns** dataframe containing feature importance.
>
> **Return type** dataframe

copro.evaluation.get_permutation_importance

copro.evaluation.**get_permutation_importance**(*clf*, *X_ft*, *Y*, *df_feat_imp*, *out_dir*)

Returns a dataframe with the mean permutation importance of the features used to train a RF tree model. Dataframe is stored to output directory as csv-file.

> **Parameters**
>
> - **clf** (`classifier`) – sklearn-classifier used in the simulation.
> - **X_ft** (`array`) – X-array containing variable values after scaling.
> - **Y** (`array`) – Y-array containing conflict data.
> - **df_feat_imp** (`dataframe`) – dataframe containing feature importances to align names across outputs.
> - **out_dir** (`str`) – path to output folder. If 'None', no output is stored.
>
> **Returns** contains mean permutation importance for each feature.
>
> **Return type** dataframe

2.7.9 Plotting

plots.selected_polygons	Creates a plotting instance of the boundaries of all selected polygons.
plots.selected_conflicts	Creates a plotting instance of the best casualties estimates of the selected conflicts.
plots.metrics_distribution	Plots the value distribution of a range of evaluation metrics based on all model simulations.
plots.correlation_matrix	Plots the correlation matrix of a dataframe.
plots.plot_ROC_curve_n_times	Plots the ROC-curve per model simulation to a pre-initiated matplotlib-instance.
plots.plot_ROC_curve_n_mean	Plots the mean ROC-curve to a pre-initiated matplotlib-instance.

copro.plots.selected_polygons

copro.plots.**selected_polygons**(*polygon_gdf*, ***kwargs*)

Creates a plotting instance of the boundaries of all selected polygons.

> **Parameters** **polygon_gdf** (`geo-dataframe`) – geo-dataframe containing the selected polygons.

Kwargs: Geopandas-supported keyword arguments.

> **Returns** Matplotlib axis object.
>
> **Return type** ax

copro.plots.selected_conflicts

copro.plots.**selected_conflicts**(*conflict_gdf*, ***kwargs*)

Creates a plotting instance of the best casualties estimates of the selected conflicts.

> **Parameters** **conflict_gdf** (`geo-dataframe`) – geo-dataframe containing the selected conflicts.

Kwargs: Geopandas-supported keyword arguments.

> **Returns** Matplotlib axis object.
>
> **Return type** ax

copro.plots.metrics_distribution

copro.plots.**metrics_distribution**(*out_dict*, *metrics*, ***kwargs*)

Plots the value distribution of a range of evaluation metrics based on all model simulations.

> **Parameters** **out_dict** (`dict`) – dictionary containing metrics score for various metrics and all simulation.

Kwargs: Matplotlib-supported keyword arguments.

> **Returns** Matplotlib axis object.
>
> **Return type** ax

copro.plots.correlation_matrix

copro.plots.**correlation_matrix**(*df*, ***kwargs*)

Plots the correlation matrix of a dataframe.

> **Parameters** **df** (`dataframe`) – dataframe containing columns to be correlated.

Kwargs: Seaborn-supported keyword arguments.

> **Returns** Matplotlib axis object.
>
> **Return type** ax

copro.plots.plot_ROC_curve_n_times

copro.plots.**plot_ROC_curve_n_times**(*ax*, *clf*, *X_test*, *y_test*, *tprs*, *aucs*, *mean_fpr*, ***kwargs*)
Plots the ROC-curve per model simulation to a pre-initiated matplotlib-instance.

> **Parameters**
>> • **ax** (*axis*) – axis of pre-initaited matplotlib-instance
>>
>> • **clf** (*classifier*) – sklearn-classifier used in the simulation.
>>
>> • **X_test** (*array*) – array containing test-sample variable values.
>>
>> • **y_test** (*list*) – list containing test-sample conflict data.
>>
>> • **tprs** (*list*) – list with false positive rates.
>>
>> • **aucs** (*list*) – list with area-under-curve values.
>>
>> • **mean_fpr** (*array*) – array with mean false positive rate.
>
> **Returns** lists with true positive rates and area-under-curve values per plot.
>
> **Return type** list

copro.plots.plot_ROC_curve_n_mean

copro.plots.**plot_ROC_curve_n_mean**(*ax*, *tprs*, *aucs*, *mean_fpr*, ***kwargs*)
Plots the mean ROC-curve to a pre-initiated matplotlib-instance.

> **Parameters**
>> • **ax** (*axis*) – axis of pre-initaited matplotlib-instance
>>
>> • **tprs** (*list*) – list with false positive rates.
>>
>> • **aucs** (*list*) – list with area-under-curve values.
>>
>> • **mean_fpr** (*array*) – array with mean false positive rate.

2.7.10 Auxiliary functions

utils.print_model_info	click.echos a header with main model information.
utils.get_geodataframe	Georeferences a pandas dataframe using longitude and latitude columns of that dataframe.
utils.show_versions	click.echos the version numbers by the main python-packages used.
utils.parse_settings	Reads the model configuration file.
utils.parse_projection_settings	This function parses the (various) cfg-files for projections.
utils.determine_projection_period	Determines the period for which projections need to be made.
utils.make_output_dir	Creates the output folder at location specfied in cfg-file, and returns dictionary with config-objects and out-dir per run.
utils.download_UCDP	If specfied in cfg-file, the PRIO/UCDP data is directly downloaded and used as model input.
utils.initiate_setup	Initiates the model set-up.

Table 10 – continued from previous page

utils.create_artificial_Y	Creates an array with identical percentage of conflict points as input array.
utils.global_ID_geom_info	Retrieves unique ID and geometry information from geo-dataframe for a global look-up dataframe.
utils.get_conflict_datapoints_only	Filters out only those polygons where conflict was actually observed in the test-sample.
utils.save_to_csv	Saves an dictionary to csv-file.
utils.save_to_npy	Saves an argument (either dictionary or dataframe) to npy-file.

copro.utils.print_model_info

copro.utils.**print_model_info**()
>   click.echos a header with main model information.

copro.utils.get_geodataframe

copro.utils.**get_geodataframe**(*config*, *root_dir*, *longitude='longitude'*, *latitude='latitude'*, *crs='EPSG:4326'*)
>   Georeferences a pandas dataframe using longitude and latitude columns of that dataframe.

>   **Parameters**

>   - **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.

>   - **root_dir** (`str`) – path to location of cfg-file.

>   - **longitude** (`str, optional`) – column name with longitude coordinates. Defaults to 'longitude'.

>   - **latitude** (`str, optional`) – column name with latitude coordinates. Defaults to 'latitude'.

>   - **crs** (`str, optional`) – coordinate system to be used for georeferencing. Defaults to 'EPSG:4326'.

>   **Returns** geo-referenced conflict data.

>   **Return type** geo-dataframe

copro.utils.show_versions

copro.utils.**show_versions**()
>   click.echos the version numbers by the main python-packages used.

copro.utils.parse_settings

copro.utils.**parse_settings**(*settings_file*)
> Reads the model configuration file.

>> **Parameters** **settings_file** (*str*) – path to settings-file (cfg-file).

>> **Returns** parsed model configuration.

>> **Return type** ConfigParser-object

copro.utils.parse_projection_settings

copro.utils.**parse_projection_settings**(*config*, *root_dir*)
> This function parses the (various) cfg-files for projections. These cfg-files need to be specified one by one in the PROJ_files section of the cfg-file for the reference run. The function returns then a dictionary with the name of the run and the associated config-object.

>> **Parameters** **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model for the reference run.

>> **Returns** dictionary with name and config-object per specified projection run.

>> **Return type** dict

copro.utils.determine_projection_period

copro.utils.**determine_projection_period**(*config_REF*, *config_PROJ*)
> Determines the period for which projections need to be made. This is defined as the period between the end year of the reference run and the specified projection year for each projection.

>> **Parameters**

>>> • **config_REF** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model for the reference run.

>>> • **config_PROJ** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model for a projection run..

>> **Returns** list containing all years of the projection period.

>> **Return type** list

copro.utils.make_output_dir

copro.utils.**make_output_dir**(*config*, *root_dir*, *config_dict*)
> Creates the output folder at location specfied in cfg-file, and returns dictionary with config-objects and out-dir per run.

>> **Parameters**

>>> • **config** (*ConfigParser-object*) – object containing the parsed configuration-settings of the model.

>>> • **root_dir** (*str*) – absolute path to location of configurations-file

>>> • **config_dict** (*dict*) – dictionary containing config-objects per projection.

>> **Returns** dictionary containing config-objects and output directories for reference run and all projection runs.

Return type dict

copro.utils.download_UCDP

copro.utils.**download_UCDP**(*config*, *root_dir*)

    If specfied in cfg-file, the PRIO/UCDP data is directly downloaded and used as model input.

        Parameters

- **config** (`ConfigParser-object`) – object containing the parsed configuration-settings of the model.

- **root_dir** (`str`) – absolute path to location of configurations-file

copro.utils.initiate_setup

copro.utils.**initiate_setup**(*settings_file*, *verbose=None*)

    Initiates the model set-up. It parses the cfg-file, creates an output folder, copies the cfg-file to the output folder, and, if specified, downloads conflict data.

        Parameters

- **settings_file** (`str`) – path to settings-file (cfg-file).

- **verbose** (`bool, optional`) – whether model is verbose or not, e.g. click.echos DEBUG output or not. If None, then the setting in cfg-file counts. Otherwise verbose can be set directly to function which superseded the cfg-file. Defaults to None.

        Returns parsed model configuration. out_dir_list: list with paths to output folders; first main output folder, then reference run folder, then (multiple) folders for projection runs. root_dir: path to location of cfg-file.

        Return type ConfigParser-object

copro.utils.create_artificial_Y

copro.utils.**create_artificial_Y**(*Y*)

    Creates an array with identical percentage of conflict points as input array.

        Parameters **Y** (`array`) – original array containing binary conflict classifier data.

        Returns array with reshuffled conflict classifier data.

        Return type array

copro.utils.global_ID_geom_info

copro.utils.**global_ID_geom_info**(*gdf*)

    Retrieves unique ID and geometry information from geo-dataframe for a global look-up dataframe. The IDs currently supported are 'name' or 'watprovID'.

        Parameters **gdf** (`geo-dataframe`) – containing all polygons used in the model.

        Returns look-up dataframe associated ID with geometry

        Return type dataframe

copro.utils.get_conflict_datapoints_only

copro.utils.**get_conflict_datapoints_only**(*X_df*, *y_df*)

> Filters out only those polygons where conflict was actually observed in the test-sample.

>> **Parameters**

>>> - **X_df** (*dataframe*) – variable values per polygon.

>>> - **y_df** (*dataframe*) – conflict data per polygon.

>> **Returns** variable values for polyons where conflict was observed. dataframe: conflict data for polyons where conflict was observed.

>> **Return type** dataframe

copro.utils.save_to_csv

copro.utils.**save_to_csv**(*arg*, *out_dir*, *fname*)

> Saves an dictionary to csv-file.

>> **Parameters**

>>> - **arg** (*dict*) – dictionary or dataframe to be saved.

>>> - **out_dir** (*str*) – path to output folder.

>>> - **fname** (*str*) – name of stored item.

copro.utils.save_to_npy

copro.utils.**save_to_npy**(*arg*, *out_dir*, *fname*)

> Saves an argument (either dictionary or dataframe) to npy-file.

>> **Parameters**

>>> - **arg** (*dict or dataframe*) – dictionary or dataframe to be saved.

>>> - **out_dir** (*str*) – path to output folder.

>>> - **fname** (*str*) – name of stored item.

AUTHORS

- Jannis M. Hoch (Utrecht University)

- Sophie de Bruin (Utrecht University, PBL)

- Niko Wanders (Utrecht University)

Corresponding author: Jannis M. Hoch (j.m.hoch@uu.nl)

FOUR

INDICES AND TABLES

- search
- genindex

R

S