# Xcessiv Documentation

*Release 0.1.0*

**Reiichiro Nakano**

**Aug 21, 2017**

# Contents

Xcessiv is a web-based application for quick and scalable hyperparameter tuning and stacked ensembling in Python.

CHAPTER 1

# Features

- Fully define your data source, cross-validation process, relevant metrics, and base learners with Python code
- Any model following the Scikit-learn API can be used as a base learner
- Task queue based architecture lets you take full advantage of multiple cores and embarrassingly parallel hyperparameter searches
- Direct integration with TPOT for automated pipeline construction
- Automated hyperparameter search through Bayesian optimization
- Easy management and comparison of hundreds of different model-hyperparameter combinations
- Automatic saving of generated secondary meta-features
- Stacked ensemble creation in a few clicks
- Automated ensemble construction through greedy forward model selection
- Export your stacked ensemble as a standalone Python file to support multiple levels of stacking

## Define your base learners and performance metrics

## Keep track of hundreds of different model-hyperparameter combinations

## Effortlessly choose your base learners and create stacked ensembles

CHAPTER 2

Contents

# Installation and Configuration

Xcessiv is currently tested on Python `2.7` and Python `3.5`.

**A note about Windows**

Because of its dependency on RQ, Xcessiv does not natively support Windows. At the moment, the only alternative for Windows users to run Xcessiv is to use Docker. See *Installation through Docker* for details.

## Installing and running Redis

For Xcessiv to work properly, it must be able to access a running Redis server.

Instructions for installing and running Redis are OS dependent and can be found at https://redis.io/topics/quickstart.

Make sure to take note of the port at which Redis is running, especially if it is not running at the default Redis port 6379.

## Installing Xcessiv

### Installing with Pip

The easiest and recommended way to install Xcessiv is to use pip:

```
pip install xcessiv
```

### Installing from source

If you want to install the latest version of Xcessiv from the master branch, you need some extra JavaScript tools to build the ReactJS application.

First, you need to install Node>=6 and Create React App.

Then, run the following commands to clone the Xcessiv master branch and build and install Xcessiv.:

```
git clone https://github.com/reiinakano/xcessiv.git
cd xcessiv/xcessiv/ui
npm run build
cd ..
cd ..
python setup.py install
```

### Installation through Docker

An alternative way to run Xcessiv is to run the server inside a Docker container. At the moment, this is the only alternative for Windows users to run Xcessiv.

There is a full guide for using Docker to run Xcessiv here.

## Configuration

To configure Xcessiv outside the default settings, create a Python file at `{HOME_FOLDER}/.xcessiv/config.py`. Here are the parameters (at their default values) you can copy / paste in that configuration module.:

```
#-------------------------------------------------
# Xcessiv config
#-------------------------------------------------
REDIS_HOST = 'localhost'  # Host address of Redis server
REDIS_PORT = 6379  # Port of Redis Server
REDIS_DB = 8  # Redis database number to use

XCESSIV_PORT = 1994  # Port at which to start the Xcessiv server
NUM_WORKERS = 1  # Number of RQ workers to start
```

Please note that aside from this configuration file, another way to configure Xcessiv is to directly pass the parameters when starting Xcessiv from the command line. In this case, the configuration variables passed through the command line overrides the the configuration found in `config.py`. See *Starting Xcessiv* for details.

# Walkthrough of Typical Xcessiv Workflow

This guide aims to demonstrate the power and flexibility of Xcessiv by walking you through a typical Xcessiv workflow. We'll optimize our performance on the breast cancer sample dataset that comes with the scikit-learn library.

## Starting Xcessiv

First, make sure your Redis server is up and running. In most cases, Redis will be running at its default port of 6379.

Open up your terminal and move to your working directory. Let's make a directory called XcessivProjects and move inside it:

```
mkdir XcessivProjects
cd XcessivProjects
```

XcessivProjects will contain all projects we create with Xcessiv.

To run Xcessiv in the current directory, we simply run:

```
xcessiv
```

This will run the Xcessiv server and a single worker process with the default configuration. You can view the Xcessiv application by pointing your browser at localhost:1994 by default.

To view the full range of configuration variables you can configure using the command line, type:

```
xcessiv -h
```

For example, to run the Xcessiv server along with 3 separate worker processes, run:

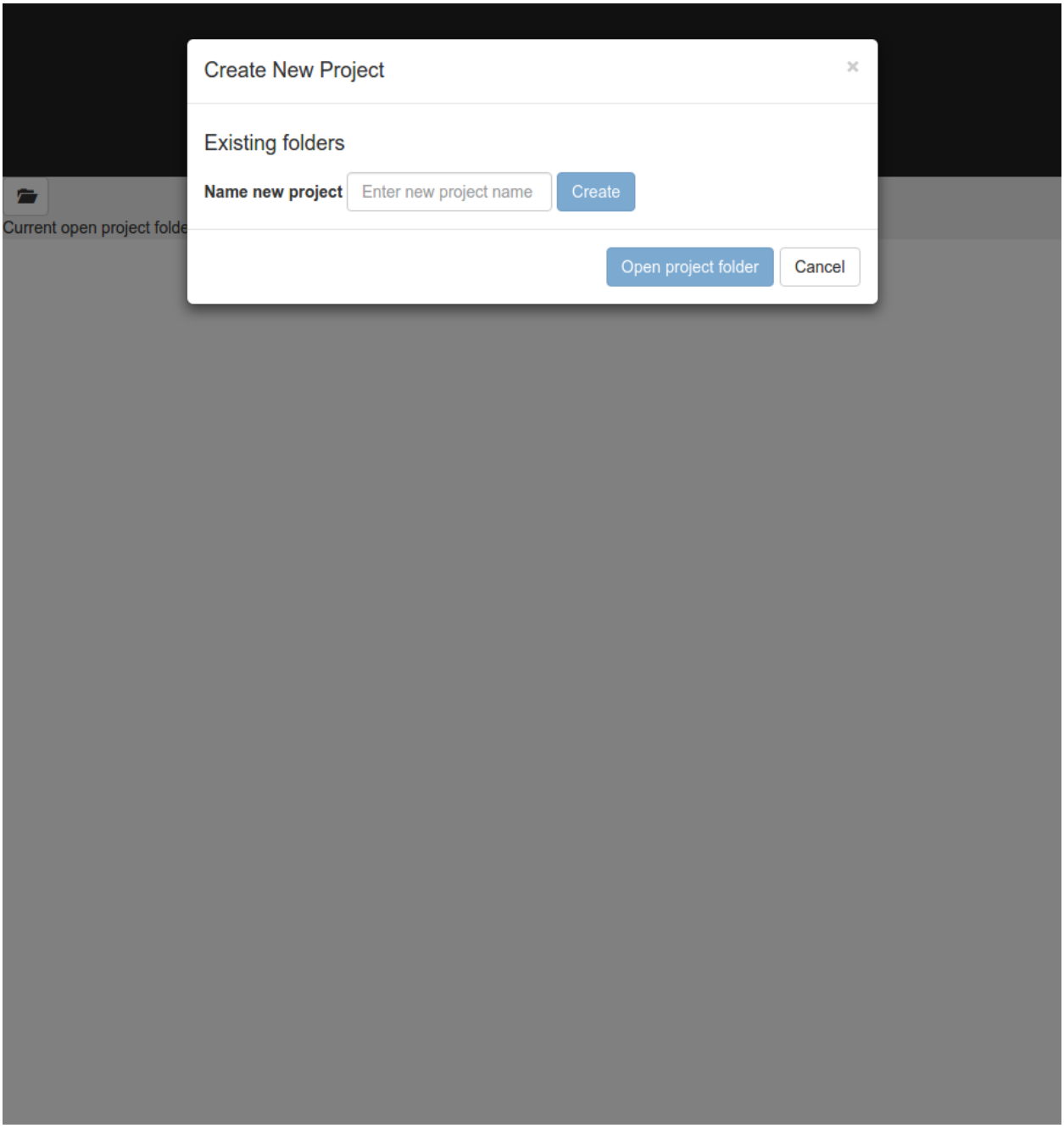
```
xcessiv -w 3
```

**A note about worker processes**

Xcessiv doesn't do the heavy processing in its application server. Instead, Xcessiv hands the jobs off to separate RQ worker processes. If you have more than one worker process running, then you will be able to process jobs in parallel without any additional configuration. However, keep in mind that each worker will consume its own CPU and memory. The optimal number of workers will then depend on your dataset size, number of cores, and available system memory.

## Creating/Opening a Project

When you open Xcessiv for the first time, you'll see a plain screen and a single button. Click on the button to open the `Create New Project` modal. This modal provides all functionality needed to create and open an Xcessiv project.




Since XcessivProjects is an empty folder, we won't see any existing projects yet. Create a new project then open it.

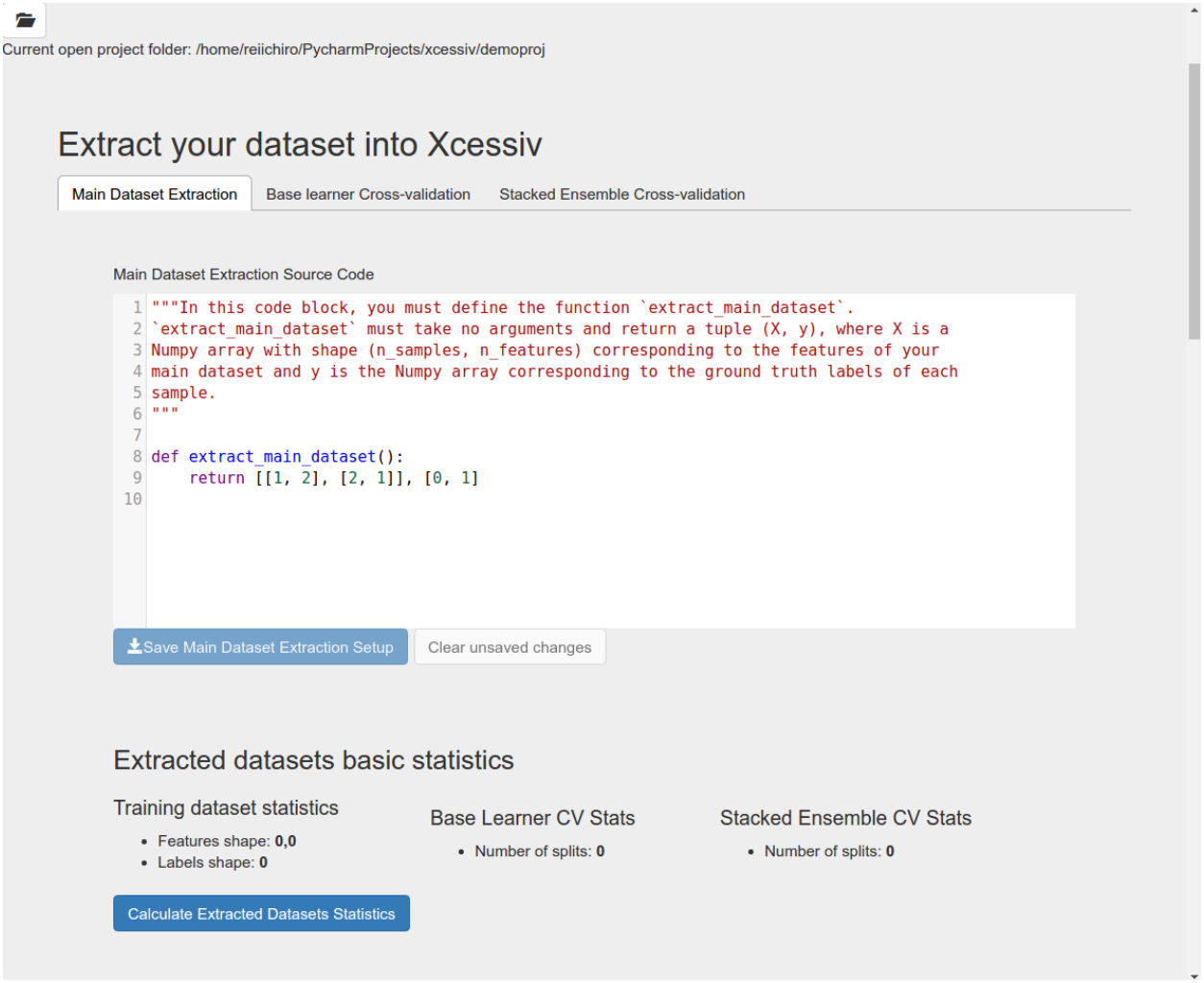
Now would be a good time to explain the structure of an Xcessiv project. An Xcessiv project is essentially a folder with a SQLite database and a sub-folder for storing saved meta-features. When you want to share your project with other people, all you need to do is give them a copy of this folder and they will be able to open it using their own Xcessiv installation. Keep in mind that this folder might get very big for large projects with a large number of saved meta-features.

## Importing your dataset into Xcessiv

After opening your new project, the first thing to do is to define your dataset.

### Define the main dataset

First, we must define the main dataset.




In the code block shown, you must define a function `extract_main_dataset()` that takes no arguments and returns a tuple `(X, y)`, where `X` is a Numpy array with shape `(n_samples, n_features)` corresponding to the features of your main dataset and `y` is the Numpy array corresponding to the ground truth labels of each sample.

Experienced **scikit-learn** users will recognize this format as the one accepted by **scikit-learn** estimators. As a project heavily influenced by the wonderful **scikit-learn** API, this is a theme that will come up repeatedly when using Xcessiv.

Since we're going with the breast cancer sample dataset that comes with scikit-learn, copy the following code into the Main Dataset Extraction code block.:

```
from sklearn.datasets import load_breast_cancer

def extract_main_dataset():
    X, y = load_breast_cancer(return_X_y=True)
    return X, y
```

Xcessiv gives you the flexibility to extract your dataset any way you want with whatever packages are included in your Python installation. You can open up the quintessential csv file with **pandas**. Or directly download the data from Amazon S3 with **boto**. As long as `extract_main_dataset()` returns the proper format of your data, any way

convenient for you will do. One important thing to keep in mind here is that for every process that needs your data, Xcessiv will call `extract_main_dataset()`. Keeping this function as light as possible is recommended.

Save your dataset extraction code and click the **Calculate Extracted Datasets Statistics** button. This will look for the `extract_main_dataset()` function in your provided code block and display the shape of `X` and `y`. This is a good way to confirm if your code works properly.

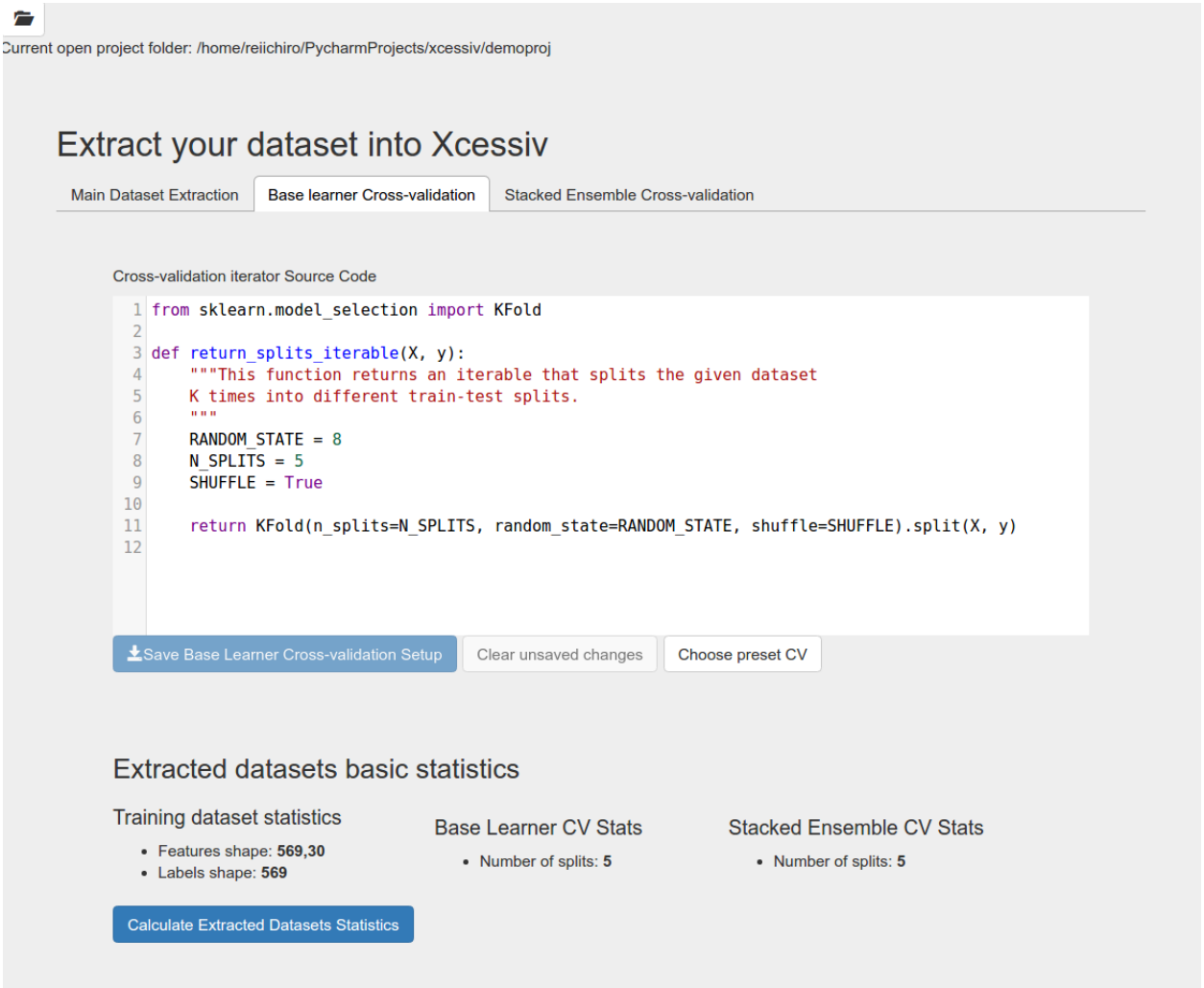
Confirm that `X` (Features array) has a shape of `(569, 30)` and `y` (Labels array) has a shape of `(569,)`.

### Define the base learner cross-validation method

After defining the main dataset, we must define how Xcessiv does cross-validation for its base learners. In Xcessiv, the base learner cross-validation's purpose is two-fold.

First, the cross-validation method is used to calculate the model-hyperparameter combination's relevant evaluation metrics on the data. Experienced users will recognize this as the usual purpose of cross-validation in machine learning.

Second, the cross-validation method is used to generate the meta-features. Meta-features are the term used for new features generated by the base learners that are used by the second-level learner in stacked ensembling. In most cases, this can be the actual predictions or the output probabilities of each class.




For stacked ensembles, there are two main ways to extract meta-features: cross-validation to get out-of-fold predictions for every sample in the dataset, or having a single train-test split to generate the meta-features (blending). The

difference between the two can be found at the Kaggle ensembling guide

For now, all you need to know is that using full cross-validation will allow you to use your whole training set for training the secondary learner at the expense of added computational complexity while using a single train-test split will only train the secondary learner on the meta-features generated from that test split.

Therefore, for smaller datasets, cross-validation is preferred while for larger datasets where computational cost is a real factor, you should use a single train-test split.

Since the breast cancer dataset has only 569 samples, we will use cross-validation. In the code block shown, copy the following code.:

```
from sklearn.model_selection import StratifiedKFold

def return_splits_iterable(X, y):
    """This function returns an iterable that splits the given dataset
    K times into different stratified train-test splits.
    """
    RANDOM_STATE = 8
    N_SPLITS = 5
    SHUFFLE = True

    return StratifiedKFold(n_splits=N_SPLITS, random_state=RANDOM_STATE,␣
↪shuffle=SHUFFLE).split(X, y)
```

Xcessiv gives you the flexibility to generate cross-validation folds however method you want to. To define a cross-validation method, you must define a function `return_splits_iterable()` that takes two arguments `X` and `y`. These arguments will be passed the `X` and `y` variables returned from the previously defined `extract_main_dataset()` function. `return_splits_iterable()` will then return an iterator that yields a pair of indices for each train-test split it generates. Again, this is a concept taken straight out of the **scikit-learn** API and as such, most built-in cross-validation iterators from **scikit-learn** will work. See http://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-iterators for the details.

---

**A note about random seeds**

It is extremely important that the folds generated by `return_splits_iterable()` are deterministic. Otherwise, ensembling will not work correctly. Therefore, for any cross-validation iterator that depends on a random state, make sure to set it in the function as well.

---

The given code does stratified K-Fold validation with 5 train-test splits and a random seed set at 8.

So far, we have given code for defining cross-validation. What if we wanted to do a simple train-test split for generating meta-features (blending)? In that case, it is interesting to note that a single train-test split can be defined by a cross-validation iterator that yields only one pair of indices for a train-test split. You can use either `sklearn.model_selection.ShuffleSplit` or `sklearn.model_selection.StratifiedShuffleSplit` with `n_splits` set to 1 for this functionality. Or, roll your own implementation.

If you click again on **Calculate Extracted Datasets Statistics**, you will notice that the base learner cross-validation statistics will show you the number of splits generated.

Since most problems will rely on very common cross-validation methods, Xcessiv provides several preset `return_splits_iterable()` implementations based on existing **scikit-learn** cross-validation iterators.

### Define the stacked ensemble cross-validation method

Since the secondary learner of a stacked ensemble is trained on a different set of features (the meta-features), it is natural to define a separate cross-validation method for it. Under the **Stacked Ensemble Cross-validation** tab, we see

a field extremely similar to the one we found in the previous step.

In fact, to define your cross-validation method for the secondary learner, you also need to define a function `return_splits_iterable()` with the exact same function signature as before. Keep in mind though, that the `X` and `y` arrays passed to this function will be from the meta-features.

In most use cases and for valid comparison with the base learner metrics, you can just use the exact same cross-validation method you used for the base learners.

Go ahead and copy the exact same code we used previously into this code block.:

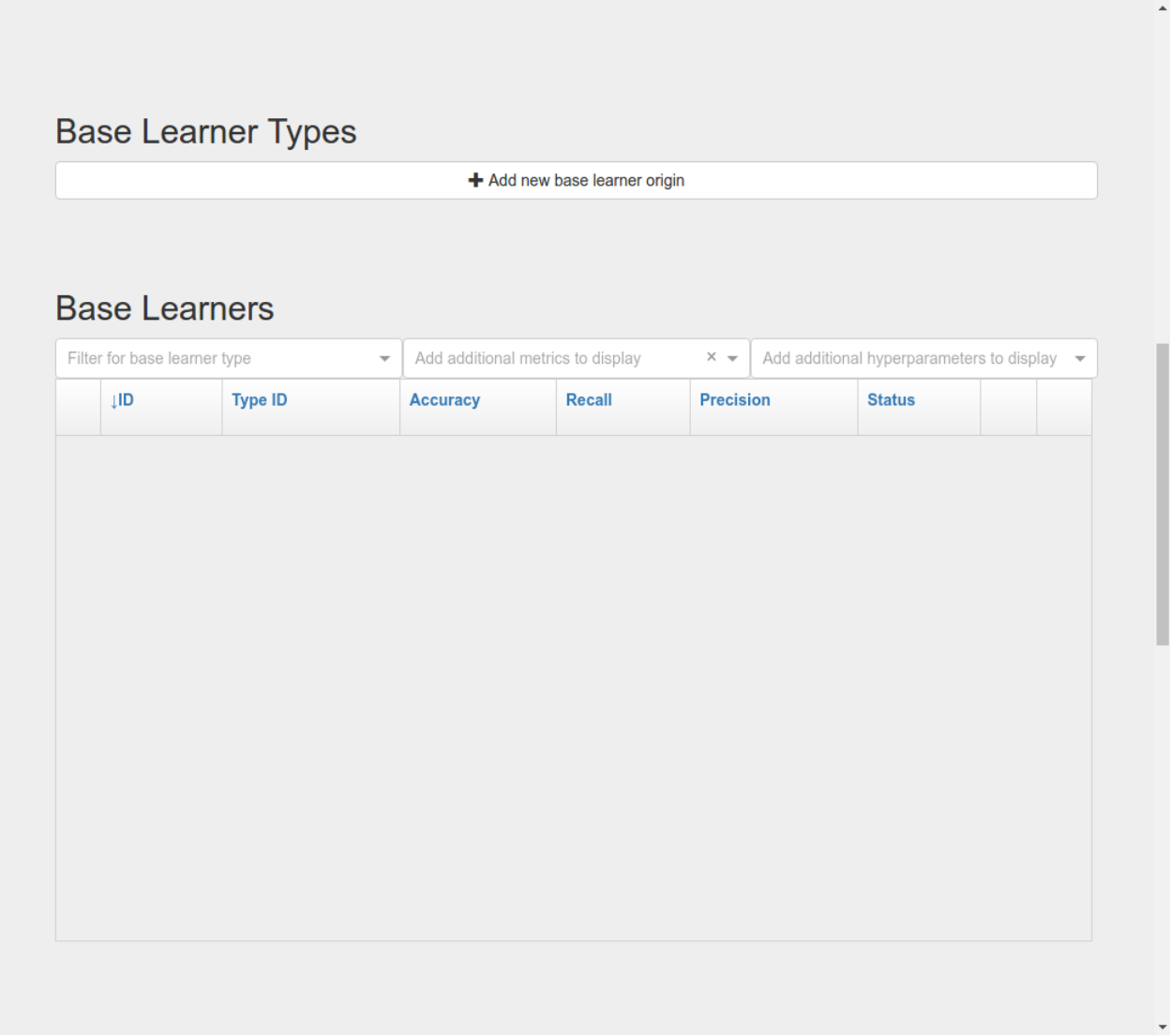
```
from sklearn.model_selection import StratifiedKFold

def return_splits_iterable(X, y):
    """This function returns an iterable that splits the given dataset
    K times into different stratified train-test splits.
    """
    RANDOM_STATE = 8
    N_SPLITS = 5
    SHUFFLE = True

    return StratifiedKFold(n_splits=N_SPLITS, random_state=RANDOM_STATE,␣
↪shuffle=SHUFFLE).split(X, y)
```

Click on **Calculate Extracted Dataset Statistics** and you should see that the stacked ensemble cross-validation statistics shows the number of splits at 5.

## Defining your base learners and metrics




When you're satisfied with your dataset extraction and base learner cross-validation setup, the next step is to define your base learners and the metrics by which you will judge the performance of each base learner.

In Xcessiv, a base learner is an *instance of a class* with the methods `fit`, `get_params`, and `set_params`.

Again, **scikit-learn** users will recognize that these are methods common across all **scikit-learn** estimators. In Xcessiv, all **scikit-learn** estimators can be used straight out of the box with no extra configuration. This is a good thing as well even if you wish to use algorithms from external libraries such as **XGBoost** or **Keras**, as these libraries often have **scikit-learn** compatible wrappers around their core estimators e.g. `XGBoostClassifier`, `KerasClassifier`.

### Use a basic scikit-learn estimator

Let's begin by defining a classic **scikit–learn** estimator, the `sklearn.ensemble.RandomForestClassifier`.

Click the **Add new base learner origin** button to define a new base learner.

Rename the default name **Base Learner Setup** to **Scikit-learn Random Forest**. Then, copy the following code into the code block then save.:

```
from sklearn.ensemble import RandomForestClassifier

base_learner = RandomForestClassifier(random_state=8)
```

All it takes to define the base learner is to assign an *instance of your estimator class* to the variable `base_learner`.

You will notice that we initialized the Random Forest's `random_state` parameter with a value of 8. We want `base_learner` initialized with the default parameters we want it to have.

Why `random_state`? Since we will be storing the performance of our base learners, we want any estimators with a randomized element to run the same way every time. Estimators with the same hyperparameters except for the random seed should still be considered different estimators. It is good practice to set any random seeds in `base_learner` with a deterministic value

### Use the scikit-learn pipeline object for more advanced estimators

An incredibly useful tool for chaining together different transformers and estimators is the **scikit-learn** `sklearn.pipeline.Pipeline` object. If you want an in-depth guide to pipelines, see http://scikit-learn.org/stable/modules/pipeline.html.

Create another base learner origin, rename it to **PCA + Random Forest**, and copy the following code into the code block then save.:

```
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.decomposition import PCA

estimators = [('pca', PCA(random_state=8)), ('rf', RandomForestClassifier(random_
↪state=8))]
base_learner = Pipeline(estimators)
```

Here we've defined a pipeline of PCA followed by Random Forest and assigned it to `base_learner`. This is now considered a single base learner type whose hyperparameters are a combination of PCA hyperparameters and Random Forest hyperparameters.

Again, notice how we've initialized all random seeds to a fixed value.

### Predefined base learners

Xcessiv contains predefined base learners for the some of the more common base learners such as Random Forest and Logistic Regression.

You can click the **Choose preset learner setting** button to view and use predefined base learners.

### Define the meta-feature generator method for a base learner

Up to now we've defined estimators that have `fit` methods for training on a train data set, and `get_params` and `set_params` for getting and setting hyperparameters, respectively.

But we haven't yet defined what method base learners use to generate the meta-features. For classifiers, the most common way to generate meta-features is either `predict` or `predict_proba`. By default, Xcessiv sets the meta-feature generator method to `predict_proba`.

For estimators that don't have the `predict_proba` method, you can change the meta-feature generator to whatever you want. For example, for SVM classifiers, it is recommended to use `decision_function` instead of `predict_proba` because of the additional computational complexity in when probabilities are generated.

Whatever you choose to be the meta-feature generator method, it must take a single variable `X`, where `X` is an array-like object of shape `(n_samples, n_features)`, and return a Numpy array of shape `(n_samples,)` or `(n_samples, num_meta_features)`, where `num_meta_features` is a positive integer referring to the number of meta-features generated per sample e.g. 5 for `predict_proba` in a dataset with 5 unique classes. In other words, the estimator must take every sample and decompose it into a single meta-feature e.g. `predict`, or a set of meta-features e.g. `predict_proba`.

This flexibility allows you to do things like using regressors as base learners for classifier ensembles, or even PCA-transformed features as meta-features.

### Define your metrics

To quantify the "goodness" of a base learner, we'll need to define metrics to evaluate the quality of its generated meta-features.

For classifiers, very common metrics include Accuracy, Recall, and Precision. For regression, a useful metric is Mean Squared Error.

Other important metrics include the Area Under Curve of the Receiver Operating Characteristic (AUC-ROC) or the Brier Score, both of which can be calculated through the class probabilities output of a classifier.

Let's define an Accuracy metric for our Random Forest base learner.

Click the **Add new metric generator** button. Name it Accuracy. In the resulting code block, add in the following code and save:

```
from sklearn.metrics import accuracy_score
import numpy as np

def metric_generator(y_true, y_probas):
    """This function computes the accuracy given the true labels array (y_true)
    and the scores/probabilities array (y_probas) with shape (num_samples, num_
↪classes).
    For the function to work correctly, the columns of the probabilities array must
    correspond to a sorted set of the unique values present in y_true.
    """
    classes_ = np.unique(y_true)
    if len(classes_) != y_probas.shape[1]:
        raise ValueError('The shape of y_probas does not correspond to the number of␣
↪unique values in y_true')
    argmax = np.argmax(y_probas, axis=1)
    y_preds = classes_[argmax]
    return accuracy_score(y_true, y_preds)
```

To define a metric, you must define a function `metric_generator` that takes two arguments. The first argument should take an array-like object referring to the set of true labels, in this case, `y_true`, with shape `(num_samples, )`. The second argument should take an array-like object with shape `(num_samples, num_meta_features)` corresponding to the generated meta-features per sample, `y_probas`. The value returned should be the calculated value of the particular metric.

The function above calculates the Accuracy metric from the ground truth labels and the corresponding set of class probabilities returned by a classifier.

In the case that our meta-feature generator method is set to `predict`, this would be the correct code for calculating Accuracy:

```
from sklearn.metrics import accuracy_score

metric_generator = accuracy_score
```

Like predefined base learners, Xcessiv comes with a bunch of preset metric generators for some commonly-used metrics. You can use and reuse these for the most common use cases instead of writing your own function every time you define a base learner.

You can add as many valid metrics as you want. These will be calculated every time the base learner is processed. Let's go ahead and add preset metric generators "Recall from Scores/Probabilities", "Precision from Scores/Probabilities", and "F1 Score from Scores/Probabilities" with the **Add preset metric generator** button.

Save your changes.

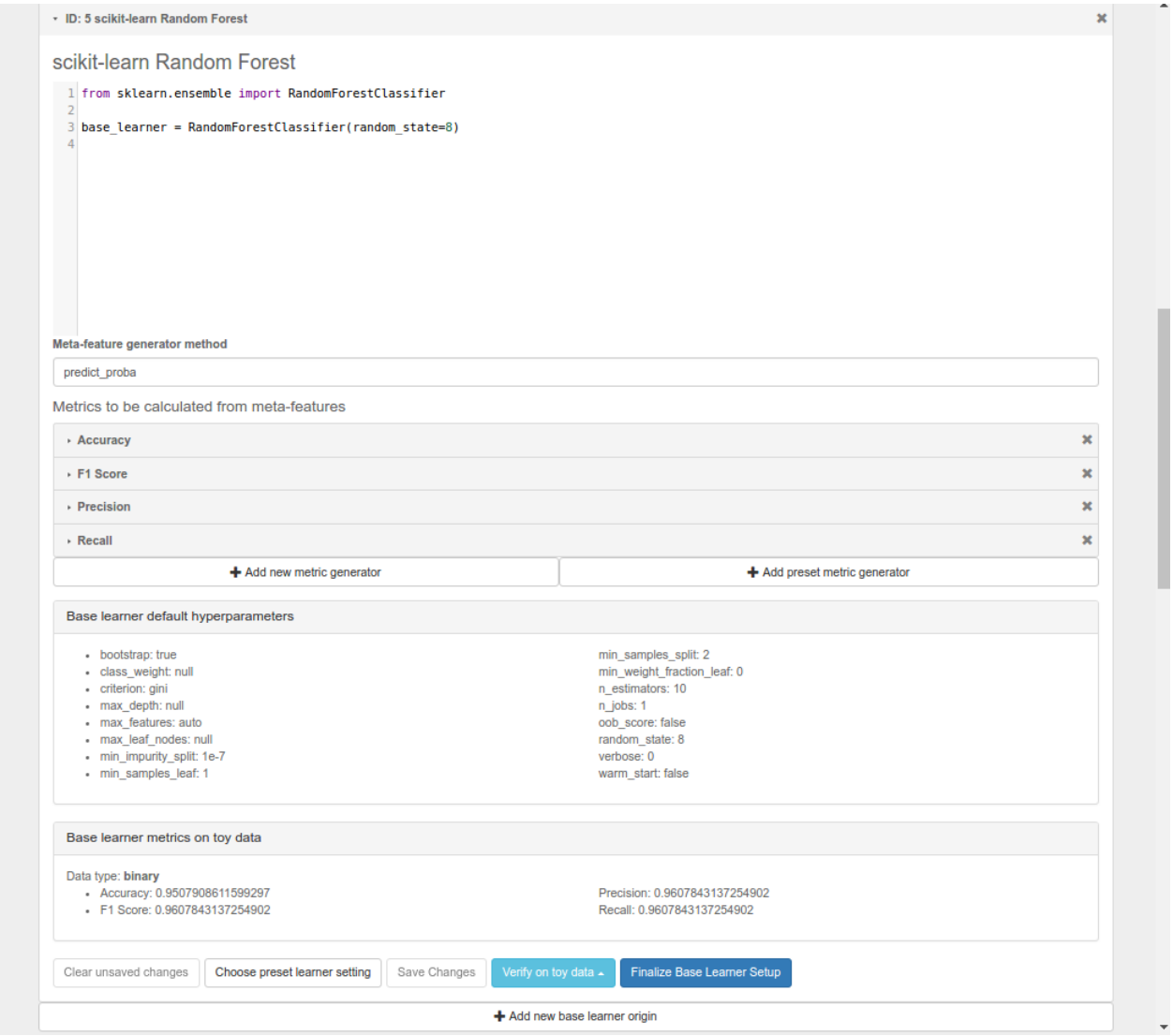
### Verify your base learner definitions and metrics

After defining your base learners and evaluation metrics, we'll want to ensure they work as expected.

Xcessiv provides verification functionality that takes your base learner and calculates its metrics on a small sample dataset.

You can choose from toy datasets such as MNIST (multiclass classification), the Wisconsin breast cancer dataset (binary classification), Boston housing prices (regression), and many more. Xcessiv also gives you an option to verify your learner against a custom dataset. You should select a sample dataset with properties that most closely resembles your actual dataset.

Since for this example, we'll be using our estimator on the breast cancer dataset, we'll want to verify it on, well, the breast cancer dataset. Click the **Verify on toy data** button and select **Breast cancer data (Binary)**. If nothing went wrong with your setup, you'll be able to see your base learner's hyperparameters with their default values, and the base learner's metrics on the sample data.



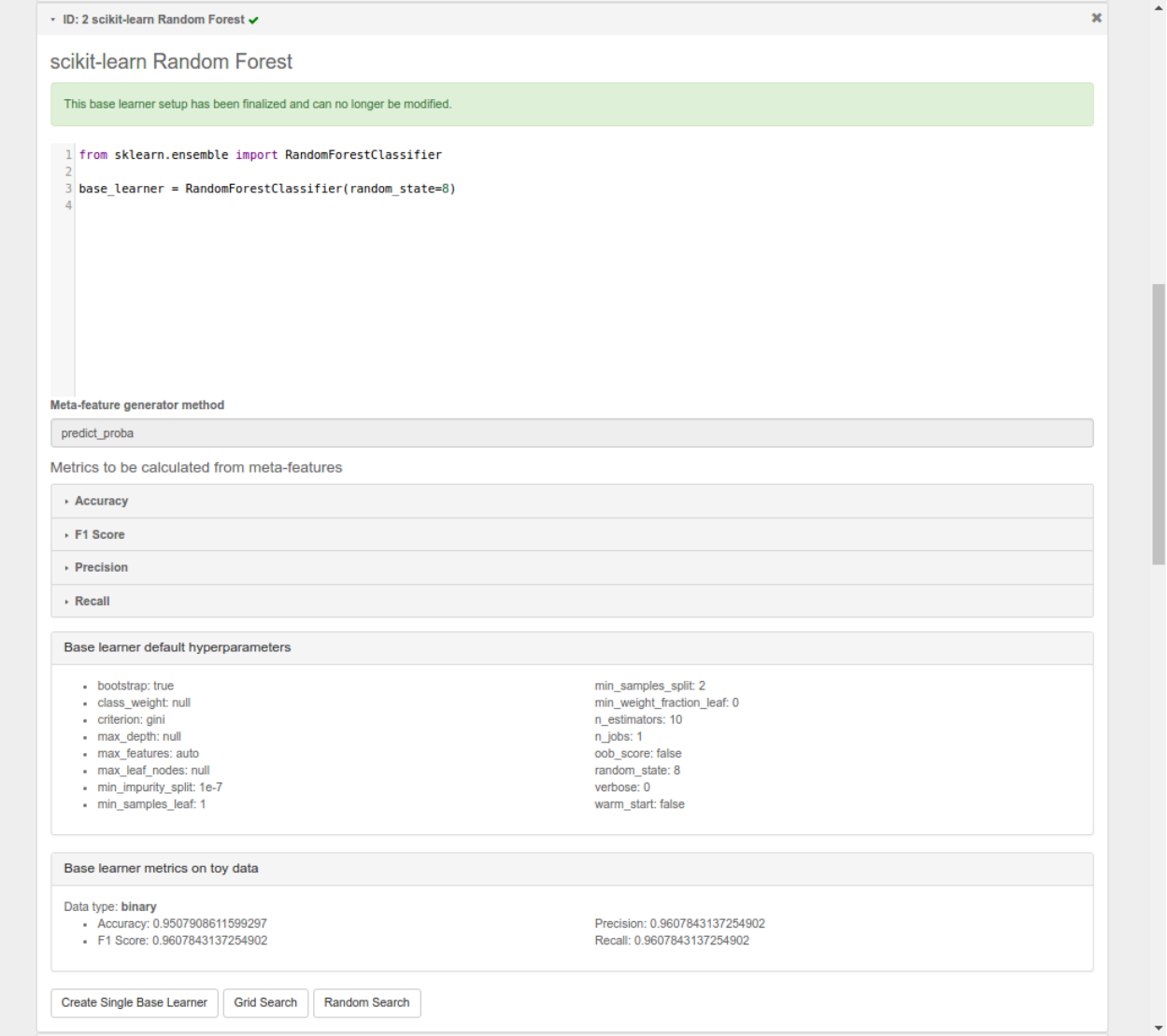

When doing an actual project, you'll want to verify your base learner on a sample dataset with the closest possible characteristics to your actual data.

### Finalize your base learner

Once you're happy with your base learner and metrics, there is one last step before you can start testing it on actual data: finalization.

Finalizing locks your base learner setup, after which you will no longer be allowed to make any changes to it. This ensures consistency during the generation of meta-features and metrics while optimizing hyperparameters and creating stacked ensembles.

After finalization, your base learner setup should look like this.




At this point, feel free to create and play around with different learners and metrics. Make sure to verify and finalize all your base learners so you can use them in the next step. For the rest of this guide, I'll assume you've created and finalized a Logistic Regression base learner and an Extra Trees Classifier base learner. Both are available as preset learners.

## Optimizing your base learners' hyperparameters

Once you've finalized a base learner, three new buttons appear in the base learner setup window: **Create Single Base Learner**, **Grid Search**, and **Random Search**.

These buttons let you generate meta-features and metrics for your data while giving you different ways to set or search through the space of hyperparameters.

Again, **scikit-learn** forms the basis for these search methods. Experienced users should have no problem figuring out how they work. For more details on grid search and random search, see http://scikit-learn.org/stable/modules/grid_search.html.

### Single base learner

Let's begin with evaluating a single base learner on the the data. Open up our Random Forest classifier, and click on the **Create Single Base Learner** button.

In the code block shown, enter the following code.:

```
params = {'n_estimators': 10}
```

For creating a single base learner, the code block only has to define a single variable `params` containing a Python dictionary. The dictionary should contain the base learner hyperparameters and corresponding values as key-value pairs. Any hyperparameter not included in the dictionary will be left at the default value. In fact, if you pass an empty dictionary to `params`, a base learner with the default hyperparameters will be run on the dataset.

After clicking **Create single base learner**, you should immediately be able to see your newly generated base learner in the "Base Learners" list. After about 5 seconds, the spinner should disappear and get replaced with a check symbol, signifying that the processing has finished.

Xcessiv does the following after creation and during processing of the base learner.

1. Xcessiv creates a new job and stores it in the Redis queue.
2. An available RQ worker reads the job and starts processing.
3. The worker loads both the dataset and base learner, and sets the base learner with the desired hyperparameters using `set_params`.
4. The worker generates meta-features using the method defined during dataset extraction (cross-validation or through a separate holdout set).
5. Using the newly generated meta-features and ground truth labels, the worker calculates the provided metrics for the given base learner.
6. The worker updates the database directly with the newly calculated metrics.
7. The worker saves a copy of the meta-features to the Xcessiv project folder. These are used during the ensembling phase.
8. The browser polls the Xcessiv server from time to time to see if the job has finished and updates the user interface accordingly.

One significant advantage provided by this architecture is that you don't need to keep the browser open to see the results later on. As long as the worker itself is not stopped while processing, the corresponding database entry will be updated upon success, and you will be able to view the result when you reopen the Xcessiv web application later.

### Grid Search

Doing a grid search is a common way of quickly exploring hyperparameter spaces.

Let's open up our Logistic Regression classifier.

Click **Grid Search**, and enter the following code.:

```
param_grid = [{'C': [0.01, 0.1, 1, 10, 100]}]
```

Five new base learners should be created, with `C` values of 0.01, 0.1, 1, 10, and 100 respectively.

The format of `param_grid` should be exactly as that described in http://scikit-learn.org/stable/modules/grid_search.html#exhaustive-grid-search.

Now, reopen the Grid Search modal and re-enter the parameter grid you ran previously. You'll see that your request is successful but no new base learners are actually created. Xcessiv automatically detects whether a previous model-hyperparameter combination has already been processed and skips it. You don't need to worry about overlapping grid search spaces.

Remember that this is Python code, so if you're feeling creative, you can also enter things like:

```
param_grid = [{'C': range(10)}]
```

### Random Search

Randomized parameter optimization is also a popular method of searching hyperparameters.

On our Extra Trees Classifier, click **Random Search**, and enter the following:

```
from scipy.stats import randint
from scipy.stats import expon

import numpy as np

np.random.seed(8)

param_distributions = {'max_depth': randint(10, 100),
                       'min_weight_fraction_leaf': expon(scale=.1)}
```

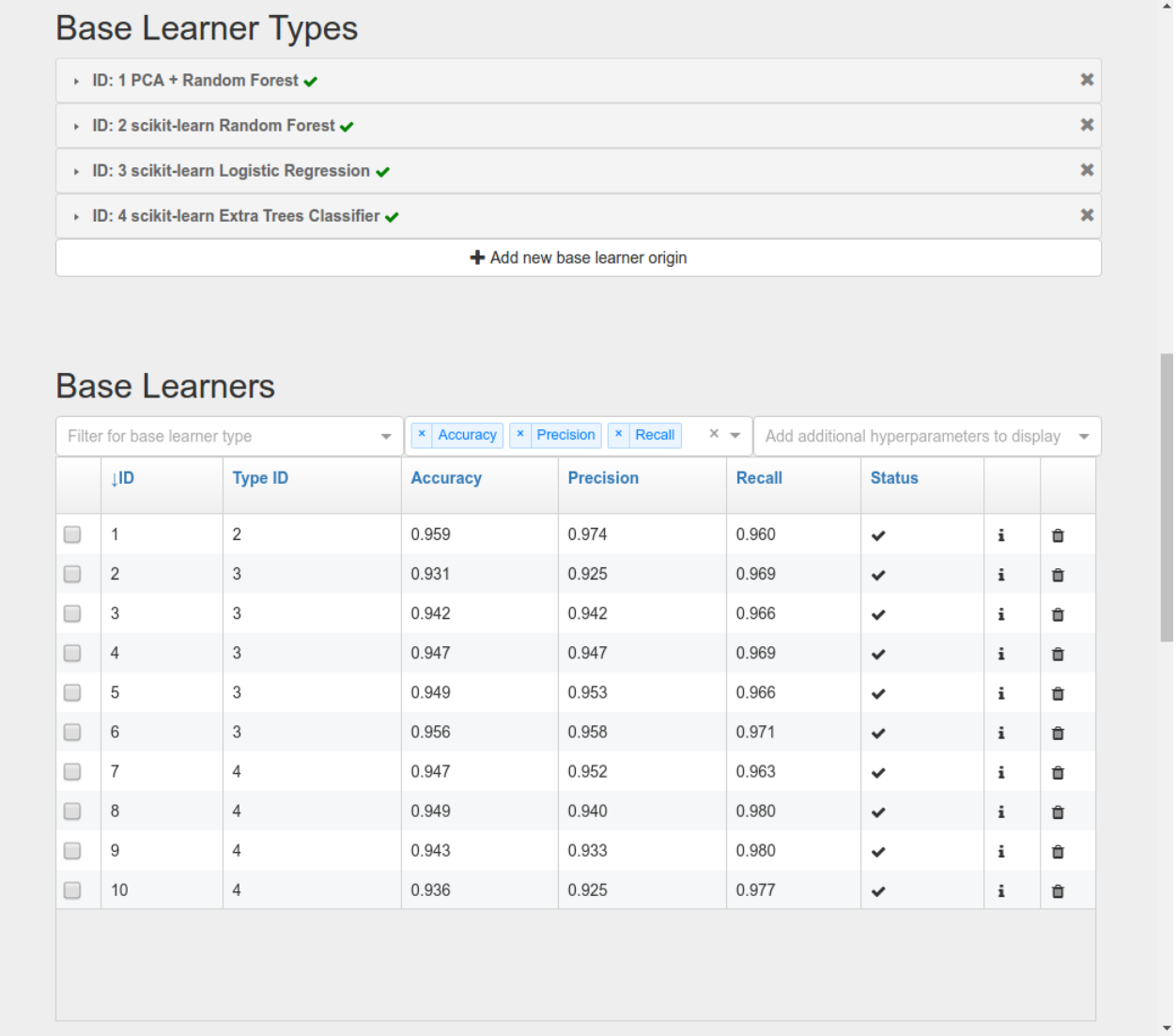
Enter `4` in the **Number of base learners to create** field.

Four new base learners should be created, with random values for `max_depth` and `min_weight_fraction_leaf`, sampled from the given **scipy** distributions.

`param_distributions` should be a dictionary whose format is described in detail in http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization.

By default, the **scipy** distributions will return different values every time you run the random search because it is, well, *random*. However, if you set the Numpy global random seed using `np.random.seed()`, you'll be able to exactly reproduce random searches.

At this point your list of base learners should look like this.




### Bayesian Search

As of `v0.3.0`, Xcessiv includes an experimental automated hyperparameter tuning functionality based on Bayesian search. For the purposes of this initial walkthrough, we will skip this and move on to the next section. A detailed tutorial for using Bayesian optimization can be found in *Bayesian Hyperparameter Search*.

## Creating a stacked ensemble

If you followed all steps up to now, you'd have 10 base learners. In practice, you'd probably try a lot more than ten but for now, let's go ahead and stack them together using a second-level classifier.

You can add base learners to your ensemble through their checkbox, or by manually selecting their IDs.

Let's select the highest performing base learner from each base learner type. For stacked ensembles, it's good to have as much variance as possible in your meta-features. One way to ensure that is to use as many different types of base learners as you can.

In the **Select secondary base learner to use** dropdown list, choose Logistic Regression as your secondary classifier. You can use anything you want here of course, but let's keep things simple for now.

To set the hyperparameters of the secondary learner, enter the following into the code block.:

```
params = {}
```

This should keep the Logistic Regression at its default values. If you'll notice, the format required for this code block is exactly the same as that required when creating a single base learner.

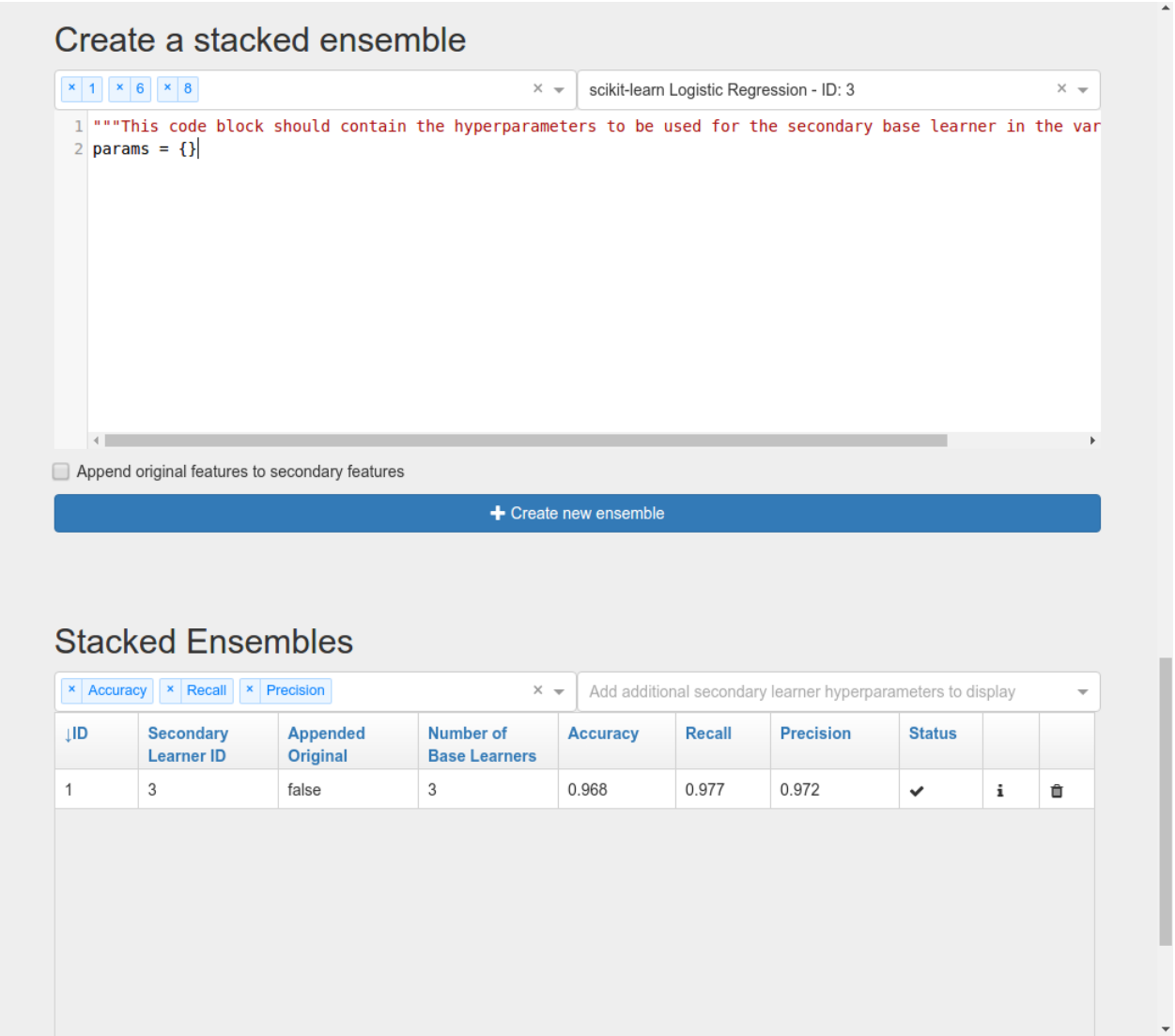
---

**Appending original features to the meta-features**

As of 0.3.8, Xcessiv no longer provides the checkbox option to append the original features to the meta-features. However, this functionality can be achieved by using the preset "Identity Transformer" and including this as a base learner in the ensemble. This approach reinforces the fact that a base learner is simply something that generates meta-features from features, and one that lets all features pass through unchanged fits this description.

---

After a short time, your ensemble should finish processing, and you'll be able to see its performance. Here we get an accuracy of 0.968, which is higher than any individual base learner.



Here's a complete list of what happens when Xcessiv creates a new ensemble. Note that it is very similar to what Xcessiv does when processing a base learner.

1. Xcessiv creates a new job and stores it in the Redis queue.
2. An available RQ worker reads the job and starts processing.
3. The worker loads the secondary learner class and selected base learners' saved meta-features from the project folder, and sets the secondary learner with the desired hyperparameters using `set_params`.
4. The worker concatenates the meta-features, and if selected, the original features, together to create the new feature set.
5. Using the cross-validation method you set for Stacked Ensemble Cross-validation, the secondary base learner's metrics on the new feature set are calculated.
6. The worker updates the database directly with the newly calculated metrics.
7. The browser polls the Xcessiv server from time to time to see if the job has finished and updates the user interface accordingly.

And that's it! Try experimenting with more base learners, appending the original features to the meta-features, and even changing the type of your secondary learner. Push that accuracy up as high as you possibly can!

Normally, it would take a lot of extraneous code just to set things up and keep track of everything you try, but Xcessiv takes care of all the dirty work so you can focus solely on the important thing, constructing your ultimate ensemble.

## Exporting your stacked ensemble

### As a Python file

Let's say that after trying out different stacked ensemble combinations, you think you've found the one. It wouldn't be very useful if you didn't have a way to use it on other data to generate predictions. Xcessiv offers a way to convert any stacked ensemble into an importable Python file. Click on the export icon of your chosen ensemble, and enter a unique name to save your file as.

In this walkthrough, we'll save our ensemble as "myensemble.py".

On successful export, Xcessiv will automatically save your Python file inside your project folder.

Your ensemble can then be imported from `myensemble.py` like this.:

```
# Make sure myensemble.py is importable
from myensemble import base_learner
```

`base_learner` will then contain a stacked ensemble instance with the methods `get_params`, `set_params`, `fit`, and the ensemble's secondary learner's meta-feature generator method. For example, if your secondary learner's meta-feature generator method is `predict`, you'll be able to call `base_learner.predict()` after fitting.

Here's an example of how you'd normally use an imported ensemble.:

```
from myensemble import base_learner

# Fit all base learners and secondary learner on training data
base_learner.fit(X_train, y_train)

# Generate some predictions on test/unseen data
predictions = base_learner.predict(X_test)
```

Most common use cases for `base_learner` will involve using a method other than the configured meta-feature generator. Take the case of using `sklearn.linear_model.LogisticRegression` as our secondary learner. `sklearn.linear_model.LogisticRegression` has both methods `predict()` and `predict_proba()`, but if our meta-feature generator is set to `predict()`, Xcessiv doesn't know `predict_proba()` actually exists and only `base_learner.predict()` will be a valid method. For these cases, `base_learner` exposes a method `_process_using_meta_feature_generator()` you can use in the following way.:

```
from myensemble import base_learner

# Fit all base learners and secondary learner on training data
base_learner.fit(X_train, y_train)

# Generate some prediction probabilities on test/unseen data
probas = base_learner._process_using_meta_feature_generator(X_test, 'predict_proba')
```

### As a standalone base learner setup

You'll notice that `base_learner` follows the **scikit-learn** interface for estimators. That means you'll be able to use it as its own standalone base learner. If you're crazy enough, you can even try *stacking together already stacked ensembles*.

In fact, Xcessiv has built in functionality to directly export your stacked ensemble as a standalone base learner setup.

In the **Export ensemble** modal, simply click on **Export as separate base learner setup**. A new base learner setup will be created containing source code for the selected stacked ensemble. At this point, you'll be able to use it just like any other base learner. Rename it, add any relevant metrics, tune it, and stack it!

**Warning:** Xcessiv's export functionality works by simply concatenating the source code for the different base learners and your cross-validation scheme. While this is not a problem in most cases, things *can* break. For example, if a base learner's source code starts with `from __future__ import`, it will *not* end up on the first line and this will need to be manually edited out in the exported file.

## Automated Runs

Xcessiv includes support for various algorithms that aim to provide automation for things such as hyperparameter optimization and base learner/pipeline construction.

Once you begin an automated run, Xcessiv will take care of updating your base learner setups/base learners for you while you go do something else.

As of v0.4.0, Xcessiv supports two types of automated runs: Bayesian Hyperparameter Search and TPOT base learner construction.

### TPOT base learner construction

Xcessiv is great for tuning different pipelines/base learners and stacking them together, but with all possible combinations of pipelines, it is a boon to use something that can build that pipeline for you automatically.

This is exactly what TPOT promises to do for you.

As of v0.4, Xcessiv has built-in support for directly exporting the pipeline code generated by TPOT as a base learner setup in Xcessiv.

Right next to the **Add new base learner origin** button, click on the **Automated base learner generation with TPOT** button. In the modal that pops up, enter the following code.:

```
from tpot import TPOTClassifier

tpot_learner = TPOTClassifier(generations=5, population_size=50, verbosity=2)
```

To use TPOT, simply define a `TPOTClassifer` or `TPOTRegressor` and assign it to the variable `tpot_learner`. The arguments for `TPOTClassifer` or `TPOTRegressor` can be found in the TPOT API documentation.

When you click **Go**, a new automated run will be created that runs `tpot_learner` on your training data then creates a new base learner setup containing the code for the best pipeline found by TPOT.

Once TPOT is finished, you'll likely end up with something like this in your newly generated base learner.:

```
import numpy as np

from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer

# NOTE: Make sure that the class is labeled 'class' in the data file
tpot_data = np.recfromcsv('PATH/TO/DATA/FILE', delimiter='COLUMN_SEPARATOR', dtype=np.
↪float64)
features = np.delete(tpot_data.view(np.float64).reshape(tpot_data.size, -1), tpot_
↪data.dtype.names.index('class'), axis=1)
training_features, testing_features, training_classes, testing_classes = \
    train_test_split(features, tpot_data['class'], random_state=42)

exported_pipeline = make_pipeline(
    Normalizer(norm="max"),
    ExtraTreesClassifier(bootstrap=False, criterion="entropy", max_features=0.15, min_
↪samples_leaf=7, min_samples_split=13, n_estimators=100)
)

exported_pipeline.fit(training_features, training_classes)
results = exported_pipeline.predict(testing_features)
```

To convert it to an Xcessiv-compatible base learner, remove all the unneeded parts and modify the code to this.:

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer

base_learner = make_pipeline(
    Normalizer(norm="max"),
    ExtraTreesClassifier(bootstrap=False, criterion="entropy", max_features=0.15, min_
↪samples_leaf=7, min_samples_split=13, n_estimators=100, random_state=8)
)
```

Notice two changes: we renamed `exported_pipeline` to `base_learner` to follow the Xcessiv format, and set the `random_state` parameter in the `sklearn.ensemble.ExtraTreesClassifier` object to 8 for determinism.

Set the name, meta-feature generator, and metrics for your base learner setup as usual, then verify and confirm. You will now be able to use your curated pipeline as any other base learner in your Xcessiv workflow.

## Bayesian Hyperparameter Search

Aside from grid search and random search that were covered in the previous chapter, Xcessiv offers another popular hyperparameter optimization method - Bayesian optimization.

Unlike grid search and random search, where hyperparameters are explored independent of each other, Bayesian optimization records the results of previously explored hyperparameter combinations and uses them to figure out which hyperparameters to try next. Theoretically, this should allow for faster convergence to a local maximum and less time wasted on exploring hyperparameters that are not likely to produce good results.

Keep in mind that there are a few limitations to this method. First, since the hyperparameter combinations to explore are based on previously explored hyperparameters, the Bayesian hyperparameter search cannot take advantage of multiple Xcessiv workers in the same way as Grid Search and Random Search. All hyperparameter combinations are explored by a single worker.

Second, Bayesian optimization can only explore numerical hyperparameters. A hyperparameter that takes only strings (e.g. `criterion` in `sklearn.ensemble.RandomForestClassifier`), cannot be tuned with Bayesian optimization. Instead, you must set the value or leave it at default before the search begins.

The Bayesian optimization method used by Xcessiv is implemented through the open-source BayesianOptimization Python package.

Let's begin.

Suppose you're exploring the hyperparameter space of a scikit-learn Random Forest classifier on some classification data. Your base learner setup will have this code.:

```
from sklearn.ensemble import RandomForestClassifier

base_learner = RandomForestClassifier(random_state=8)
```

Make sure you also use "Accuracy" as a metric.

You want to use Bayesian optimization to tune the hyperparameters `max_depth`, `min_samples_split`, and `min_samples_leaf`. After verifying and finalizing the base learner, click the **Bayesian Optimization** button and enter the following configuration into the code block and hit Go.:

```
random_state = 8  # Random seed

# Default parameters of base learner
default_params = {
  'n_estimators': 200,
  'criterion': 'entropy'
}

# Min-max bounds of parameters to be searched
pbounds = {
  'max_depth': (10, 300),
  'min_samples_split': (0.001, 0.5),
  'min_samples_leaf': (0.001, 0.5)
}

# List of hyperparameters that should be rounded off to integers
integers = [
  'max_depth'
]

metric_to_optimize = 'Accuracy'  # metric to optimize

invert_metric = False  # Whether or not to invert metric e.g. optimizing a loss
```

```
# Configuration to pass to maximize()
maximize_config = {
  'init_points': 2,
  'n_iter': 10,
  'acq': 'ucb',
  'kappa': 5
}
```

If everything goes well, you should see that an "Automated Run" has started. From here, you can just watch as the Base Learners list updates with a new entry every time the Bayesian search explores a new hyperparameter combination.

Let's review the code we used to configure the Bayesian search.

All variables shown need to be defined for Bayesian search to work properly.

First, the `random_state` parameter is used to seed the Numpy random generator that is used internally by the Bayesian search. You can set this to any integer you like.:

```
random_state = 8
```

Next, define the default parameters of your base learner in the `default_params` dictionary. In our case, we don't really want to search `n_estimators` or `criterion` but we don't want to leave them at their default values either. This dictionary will set `n_estimators` to 200 and `criterion` to "entropy" for base learners produced by the Bayesian search. If `default_params` is an empty dictionary, the default values for all non-searchable hyperparameters will be used.:

```
default_params = {
  'n_estimators': 200,
  'criterion': 'entropy'
}
```

The `pbounds` variable is a dictionary that maps the hyperparameters to tune with their minimum and maximum values. In our example, `max_depth` will be searched but kept between 10 and 300, while `min_samples_split` will be searched but kept between 0.001 and 0.5.:

```
# Min-max bounds of parameters to be searched
pbounds = {
  'max_depth': (10, 300),
  'min_samples_split': (0.001, 0.5),
  'min_samples_leaf': (0.001, 0.5)
}
```

`integers` is an array containing the list of hyperparameters that should be converted to an integer before using it to configure the base learner. In our example `max_depth` only accepts integer values, so we add it to the list.:

```
# List of hyperparameters that should be rounded off to integers
integers = [
  'max_depth'
]
```

`metric_to_optimize` defines the metric that the Bayesian search will use to determine the effectiveness of a single base learner. In our case, the search optimizes for higher accuracy.

`invert_metric` must be set to `True` when the metric you are optimizing is "better" at a lower value. For example, metrics such as the Brier Score Loss and Mean Squared Error are better when they are smaller.:

```
metric_to_optimize = 'Accuracy'  # metric to optimize

invert_metric = False  # Whether or not to invert metric e.g. optimizing a loss
```

`maximize_config` is a dictionary of parameters used by the actual Bayesian search to dictate behavior such as the number of points to explore and the algorithm's acquisition function. `init_points` sets the number of initial points to randomly explore before the actual Bayesian search takes over. `n_iter` sets the number of hyperparameter combinations the Bayesian search will explore. `acq` and `kappa` refer to the parameters of the acquisition function and determine the search's balance between exploration and exploitation. Keys included in `maximize_config` that are not directly used by the Bayesian search process are passed on to the underlying `sklearn.gaussian_process.GaussianProcessRegressor` object.:

```
# Configuration to pass to maximize()
maximize_config = {
  'init_points': 2,
  'n_iter': 10,
  'acq': 'ucb',
  'kappa': 5
}
```

For more info on setting `maximize_config`, please see the `maximize()` method of the `bayes_opt.BayesianOptimization` class in the BayesianOptimization source code. Seeing this notebook example will also give you some intuition on how the different acquisition function parameters `acq`, `kappa`, and `xi` affect the Bayesian search.

## Greedy Forward Model Selection

Stacking is usually reserved as the last step of the Xcessiv process, after you've squeezed out all you can from pipeline and hyperparameter optimization. When creating stacked ensembles, you can usually expect its performance to be better than any single base learner in the ensemble.

The problem here lies in figuring out which base learners to include in your ensemble. Stacking together the top N base learners is a good first strategy, but not always optimal. Even if a base learner doesn't perform that well on its own, it could still provide brand new information to the secondary learner, thereby boosting the entire ensemble's performance even further. One way to look at it is that it provides the secondary learner a *new angle* to look at the problem and make better judgments moving forward.

Figuring out which base learners to add to a stacked ensemble is much like hyperparameter optimization. You can't really be sure if something will work until you try it. Unfortunately, trying out every possible combination of base learners is unfeasible when you have hundreds of base learners to choose from.

Xcessiv provides an automated ensemble construction method based on a heuristic process called **greedy forward model selection**. This method is adapted from Ensemble Selection from Libraries of Models by Caruana et al.

In a nutshell, the algorithm is as follows:

1. Start with the empty ensemble
2. Add to the ensemble the model in the library that maximizes the ensemmble's performance on the error metric.
3. Repeat step 2 for a fixed number of iterations or until all models have been used.

That's it!

To perform greedy forward model selection in Xcessiv, simply click on the **Automated ensemble search** button in the Stacked Ensemble section.

Select your secondary base learner in the configuration modal (Logistic Regression is a good first choice for classification tasks) and copy the following code into the code box and click Go to start your automated run.:

```
secondary_learner_hyperparameters = {}  # hyperparameters of secondary learner

metric_to_optimize = 'Accuracy'  # metric to optimize

invert_metric = False  # Whether or not to invert metric e.g. optimizing a loss

max_num_base_learners = 6  # Maximum size of ensemble to consider (the higher this is,
↪ the longer the run will take)
```

`secondary_learner_hyperparameters` is a dictionary containing the hyperparameters for your chosen secondary learner. Again, an empty dictionary signifies default parameters.

`metric_to_optimize` and `invert_metric` mean the same things they do as in *Bayesian Hyperparameter Search*.

`max_num_base_learners` refers to the total number of iterations of the algorithm. As such, this also signifies the maximum number of base learners that a stacked ensemble found through this automated run can contain. Please note that the higher this number is, the longer the search will run.

Unlike TPOT pipeline construction and Bayesian optimization, which both have an element of randomness, greedy forward model selection will always explore the same ensembles if the pool of base learners remains unchanged.

## Xcessiv and Third Party Libraries

Xcessiv provides an extremely flexible framework for experimentation with your own algorithms. Anything you can dream of can be used, as long as they conform to the **scikit-learn** interface.

Here are a few example workflows using third party libraries that work well with Xcessiv.

---

### Xcessiv with TPOT

---

**Note**

As of v0.4, Xcessiv now provides direct integration with TPOT. View *TPOT base learner construction* for details. This section is kept here to demonstrate the power of stacking together different TPOT pipelines.

---

Xcessiv is a great tool for tuning different models and pipelines and stacking them into one big ensemble, but with all the possible combinations of pipelines, where would you even begin?

Enter TPOT.

TPOT is a Python tool that automatically creates and optimizes machine learning pipelines using genetic programming.

TPOT will automatically try out hundreds of different machine learning pipelines and pick out the best one it finds. You can then export it as source code that contains a `sklearn.pipeline.Pipeline` object. From there, you can just add your curated and tuned pipeline as a new base learner type, ready for even further improvement from stacking.

In this example, we'll be using the Hill-Valley dataset with noise from the UCI Machine Learning Repository. To load it into Xcessiv, we'll use a neat little Python wrapper called pmlb. Start by installing pmlb:

```
pip install pmlb
```

Now, start a new Xcessiv project and let's dive in.

### Set up data entry and cross-validation into Xcessiv

Copy the following code into the **Main Dataset Extraction** code block.:

```
from pmlb import fetch_data

def extract_main_dataset():
    return fetch_data('Hill_Valley_with_noise', local_cache_dir='./', return_X_y=True)
```

pmlb has a nice interface that lets you extract datasets in a **scikit-learn** format very easily. You can change the argument to `local_cache_dir` above to any directory where you want to store the dataset. This way, the dataset is only downloaded the first time `extract_main_dataset()` is run.

Since the dataset is rather small, we'll use cross-validation. For both **Base learner Cross-validation** and **Stacked Ensemble Cross-validation**, copy the following code.:

```
from sklearn.model_selection import KFold

def return_splits_iterable(X, y):
    """This function returns an iterable that splits the given dataset
    K times into different train-test splits.
    """
    RANDOM_STATE = 8
    N_SPLITS = 5
    SHUFFLE = True

    return KFold(n_splits=N_SPLITS, random_state=RANDOM_STATE, shuffle=SHUFFLE).
↪split(X, y)
```

### Run TPOT to get an optimized pipeline

Open up your favorite place to run Python code (I used Jupyter notebook) and run the following TPOT code.:

```
from pmlb import fetch_data
from tpot import TPOTClassifier

X, y = fetch_data('Hill_Valley_with_noise', local_cache_dir='./', return_X_y=True)
tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, n_jobs=-1)
tpot.fit(X, y)

tpot.export('tpot_1.py')
```

This snippet will run the TPOT algorithm on the Hill valley with noise dataset and automatically find an optimal pipeline. Then, it will export the found pipeline as Python code in `tpot_1.py`.

Note that this could take a while. On my computer, it took around 30 minutes to an hour to run. If you want, you can just skip this part since the pipelines I found will be available in this documentation anyway.

---

**Note**

Note that the TPOT algorithm is stochastic, so different runs will probably result in different pipelines found. It might be best to set the `random_state` parameter in `TPOTClassifier` for reproducibility. This randomness is a good thing, because stacking works best when very different base learners are used.

---

Once the algorithm is finished running, open up `tpot_1.py` and you should see something like the following code.:

```
import numpy as np

from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer

# NOTE: Make sure that the class is labeled 'class' in the data file
tpot_data = np.recfromcsv('PATH/TO/DATA/FILE', delimiter='COLUMN_SEPARATOR', dtype=np.
↪float64)
features = np.delete(tpot_data.view(np.float64).reshape(tpot_data.size, -1), tpot_
↪data.dtype.names.index('class'), axis=1)
training_features, testing_features, training_classes, testing_classes = \
    train_test_split(features, tpot_data['class'], random_state=42)

exported_pipeline = make_pipeline(
    Normalizer(norm="max"),
    ExtraTreesClassifier(bootstrap=False, criterion="entropy", max_features=0.15, min_
↪samples_leaf=7, min_samples_split=13, n_estimators=100)
)

exported_pipeline.fit(training_features, training_classes)
results = exported_pipeline.predict(testing_features)
```

You can see that our exported pipeline is in the variable `exported_pipeline`. This is actually the only part of the code we need to add into Xcessiv.

### Adding TPOT Pipelines to Xcessiv

Create a new base learner setup and copy the following code into Xcessiv.:

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer

base_learner = make_pipeline(
    Normalizer(norm="max"),
    ExtraTreesClassifier(bootstrap=False, criterion="entropy", max_features=0.15, min_
↪samples_leaf=7, min_samples_split=13, n_estimators=100, random_state=8)
)
```

This is a stripped down version of the code in `tpot_1.py`, with only the part we need. Notice two changes: we renamed `exported_pipeline` to `base_learner` to follow the Xcessiv format, and set the `random_state` parameter in the `sklearn.ensemble.ExtraTreesClassifier` object to 8 for determinism.

Name your base learner "TPOT 1", set `predict_proba` as the meta-feature generator, and add the following preset metrics: **Accuracy from Scores/Probabilities**, **Recall from Scores/Probabilities**, **Precision from Scores/Probabilities**, **F1 Score from Scores/Probabilities**, and **AUC from Scores/Probabilities**.

Since the hill-valley dataset is binary, verify and finalize your base learner on the breast cancer dataset.

Keep in mind that the pipeline returned by TPOT has already been tuned, so there isn't much need to tune it now. Feel free to do so, though. It's very easy to do this in Xcessiv. For this case, let's just create a single new base learner with default hyperparameters. You should get a pretty good accuracy of about 0.9868.

As mentioned earlier, different runs of TPOT will probably produce different results. I ran the script two more times, this time with different random seeds set. For a random state of 10, TPOT produced the following pipeline (stripped

down to Xcessiv format).:

```
from copy import copy
from sklearn.ensemble import VotingClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline, make_union
from sklearn.preprocessing import FunctionTransformer
from sklearn.svm import LinearSVC

base_learner = make_pipeline(
    make_union(VotingClassifier([("est", LinearSVC(C=5.0, loss="hinge", tol=0.0001,␣
↪random_state=8))]), FunctionTransformer(copy)),
    LinearSVC(C=0.0001, random_state=8, loss="squared_hinge")
)
```

This combination of Linear SVCs and a VotingClassifier gets an accuracy of about 0.9612.

For a random state of 242, the following stripped down pipeline is produced.:

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer

base_learner = make_pipeline(
    Normalizer(norm="l1"),
    KNeighborsClassifier(n_neighbors=22, p=1)
)
```

This pipeline gets an accuracy of 0.9876, our highest so far.

### Stacking TPOT Pipelines together

Once they're in Xcessiv, TPOT pipelines are just regular base learners you can tune or stack. For now, we've got three high-performing base learners with rather different decision models i.e. a tree-based model, a linear SVM, and a nearest neighbors classifier. These should be ideal to stack together.

Create and finalize a preset Logistic Regression base learner. We'll use this to stack the base learners together.

Let's begin by stacking together the two highest performers, the ExtraTreesClassifier and the KNeighborsClassifier without the original features. Right off the bat, cross-validating on the secondary meta-features yields an accuracy of 0.9975.

Going further, let's see if adding the less effective (on its own) Linear SVM will prove useful to our small ensemble. Running it, we get an even better 0.9992 accuracy.

It seems that seeing how the Linear SVM looks at the problem lets our Logistic Regression meta-learner further improve its own understanding of the data.

Quoting top Kaggler Marios Michailidis:

> Sometimes it is useful to allow XGBoost to see what a KNN-classifier sees.

And that's it for our guide to using TPOT in Xcessiv. There's loads more you can try if you want to push up model performance even more. For instance, why not see if a TPOT pipeline as your secondary learner will work better? Or try experimenting with adding the original features appended to the meta-features. Xcessiv is built for this kind of crazy exploration. Go get those accuracies up as high as you can!

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search