Foreshadow Documentation

Release 0.2.1

Adithya Balaji, Alexander Allen

Oct 11, 2019

Contents

1 Installing Foreshadow **3**

2 Getting Started **5**

3 Key Features **7**

4 Documentation **9**

5 The User Guide **11**
5.1 User Guide 11
5.2 Frequently Asked Questions 21

6 The Developer Guide **23**
6.1 Developers Guide 23
6.2 Contributing 27

7 API **31**
7.1 API Reference 31
7.2 Project Architecture 42

8 Changelog **47**
8.1 Foreshadow 0.2.1 (2019-09-26) 47
8.2 Foreshadow 0.2.0 (2019-09-24) 47
8.3 Foreshadow 0.1.0 (2019-06-28) 48

9 Indices and tables **49**

Python Module Index **51**

Index **53**

Foreshadow is an automatic pipeline generation tool that makes creating, iterating, and evaluating machine learning pipelines a fast and intuitive experience allowing data scientists to spend more time on data science and less time on code.

CHAPTER 1

Installing Foreshadow

```
$ pip install foreshadow
```

Read the documentation to set up the project from source.

CHAPTER 2

Getting Started

To get started with foreshadow, install the package using pip install. This will also install the dependencies. Now create a simple python script that uses all the defaults with Foreshadow.

First import foreshadow

```
import foreshadow as fs
```

Also import sklearn, pandas, and numpy for the demo

```
import pandas as pd

from sklearn.datasets import boston_housing
from sklearn.model_selection import train_test_split
```

Now load in the boston housing dataset from sklearn into pandas dataframes. This is a common dataset for testing machine learning models and comes built in to scikit-learn.

```
boston = load_boston()
bostonX_df = pd.DataFrame(boston.data, columns=boston.feature_names)
bostony_df = pd.DataFrame(boston.target, columns=['target'])
```

Next, exactly as if working with an sklearn estimator, perform a train test split on the data and pass the train data into the fit function of a new Foreshadow object

```
X_train, X_test, y_train, y_test = train_test_split(bostonX_df,
   bostony_df, test_size=0.2)
shadow = fs.Foreshadow()
shadow.fit(X_train, y_train)
```

Now *fs* is a fit Foreshadow object for which all feature engineering has been performed and the estimator has been trained and optimized. It is now possible to utilize this exactly as a fit sklearn estimator to make predictions.

```
shadow.score(X_test, y_test)
```

Great, you now have a working Foreshaow installation! Keep reading to learn how to export, modify and construct pipelines of your own.

CHAPTER 3

Key Features

- Automatic Feature Engineering
- Automatic Model Selection
- Rapid Pipeline Development / Iteration
- Automatic Parameter Optimization
- Ease of Extensibility
- Scikit-Learn Compatible

Foreshadow supports python 3.6+

CHAPTER 4

Documentation

Read the docs!

CHAPTER 5

The User Guide

5.1 User Guide

5.1.1 Getting Started

To get started with foreshadow, install the package using pip install. This will also install the dependencies. Now create a simple python script that uses all the defaults with Foreshadow.

First import foreshadow

```
import foreshadow as fs
```

Also import sklearn, pandas, and numpy for the demo

```
import pandas as pd

from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
```

Now load in the Boston housing dataset from sklearn into pandas dataframes. This is a common dataset for testing machine learning models and comes built-in to scikit-learn.

```
boston = load_boston()
bostonX_df = pd.DataFrame(boston.data, columns=boston.feature_names)
bostony_df = pd.DataFrame(boston.target, columns=['target'])
```

Next, exactly as if working with an sklearn estimator, perform a train test split on the data and pass the train data into the fit function of a new Foreshadow object

```
X_train, X_test, y_train, y_test = train_test_split(bostonX_df,
   bostony_df, test_size=0.2)
shadow = fs.Foreshadow()
shadow.fit(X_train, y_train)
```

Now *fs* is a fit Foreshadow object for which all feature engineering has been performed and the estimator has been trained and optimized. It is now possible to utilize this exactly as a fit sklearn estimator to make predictions.

```
print(shadow.score(X_test, y_test))
```

Great, you now have a working Foreshaow installation! Keep reading to learn how to export, modify and construct pipelines of your own.

Here it is all together.

```
import foreshadow as fs
import pandas as pd
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

boston = load_boston()
bostonX_df = pd.DataFrame(boston.data, columns=boston.feature_names)
bostony_df = pd.DataFrame(boston.target, columns=['target'])

X_train, X_test, y_train, y_test = train_test_split(bostonX_df,
   bostony_df, test_size=0.2)
shadow = fs.Foreshadow()
shadow.fit(X_train, y_train)

print(shadow.score(X_test, y_test))
```

Recommended Workflow

There are many ways to use Foreshadow, but we recommend using this workflow initially as it is the quickest and easiest way to generate a high-performing model with minimum effort.

First, prep your data into X_train, X_test, y_train and y_test pandas dataframes. For example:

```
boston = load_boston()
bostonX_df = pd.DataFrame(boston.data, columns=boston.feature_names)
bostony_df = pd.DataFrame(boston.target, columns=['target'])

X_train, X_test, y_train, y_test = train_test_split(bostonX_df,
   bostony_df, test_size=0.2)
```

Then initialize a default Foreshadow object with a sklearn estimator such as XGBoost. We want this process to be fast so we can iterate, so for the time being we will override the default TPOT model selection, ensembling and hyperparameter optimization for regression problems with a simple default XGBoost regressor.

```
shadow = fs.Foreshadow(estimator=XGBRegressor())
```

Then fit the train data on that object

```
shadow.fit(X_train, Y_train)
```

You now have an initial pipeline. Lets see how it did and serialize it to a JSON file so we can look at it.

```
# Score the pipeline
shadow.score(X_test, y_test)

# Serialize the pipeline
```

(continues on next page)

(continued from previous page)

```
x_proc = shadow.X_preparer.serialize()
y_proc = shadow.y_preparer.serialize()

# Write the serialized pipelines to file
json.dump(x_proc, open("x_proc.json", "w"), indent=2)
json.dump(y_proc, open("y_proc.json", "w"), indent=2)
```

Now we have two pipeline configurations, one for our X data and one for our Y data. We also have an initial idea of how well the initial pipeline performed.

Let's suppose that you want to experiment with a different scaler. Open the configuration JSON and make this change. (Look to the *Configuration* section for more details on this)

For example, add the following snippet to the bottom or x_proc.json

```
"combinations": [
    {
      "columns.CHAS.0": "['NumericIntent', 'CategoricalIntent']"
    }
 ]
```

Now let's re-create the Foreshadow object with your changes.

```
import json

# Load in the configs from file
x_proc = json.load(open("x_proc.json", "r"))
y_proc = json.load(open("y_proc.json", "r"))

# Create the preprocessors
x_processor = fs.Preprocessor(from_json=x_proc)
y_processor = fs.Preprocessor(from_json=y_proc)

# Create the foreshadow object
shadow = fs.Foreshadow(X_preparer=x_processor, y_preparer=y_processor,␣
↪estimator=XGBRegressor())

# Fit the foreshadow object
shadow.fit(X_train, y_train)

# Score the foreshadow object
shadow.score(X_test, y_test)
```

Now we can see the performance difference as a result of the changes. This process of swapping in and out different scalers is slow and tedious though. Let's add a combinations section to the configuration file and let an optimizer do the heavy lifting of evaluating the framework.

First, read the *Hyperparameter Tuning* section about how hyperparameter optimization works in Foreshadow. Then add a combinations section to the exported JSON file(s) you have from the preprocessor. Remember that the more parameters you add, the longer it will take. We recommend focusing on a set of related parameters one by one and optimizing them individually. e.g. Optimize thresholds for Scaling, then thresholds for Encoding, then feature reduction (PCA / LDA) etc.

Once you add a combinations section to figure out the best parameters, create the Foreshadow object again, except this time with an optimizer such as GridSearchCV or RandomSearchCV from sklearn.

```
# Load in the configs from file
x_proc_combo = json.load(open("x_proc_combo.json", "r"))
y_proc_combo = json.load(open("y_proc_combo.json", "r"))

# Create the preprocessors
x_processor = Preprocessor(from_json=x_proc_combo)
y_processor = Preprocessor(from_json=y_proc_combo)

# Create the foreshadow object
shadow = fs.Foreshadow(X_preparer=x_processor, y_preparer=y_processor,␣
↪estimator=XGBRegressor(), optimizer=GridSearchCV)

# Fit the foreshadow object
shadow.fit(X_train, y_train)

# Score the foreshadow object
shadow.score(X_test, y_test)

# Extract the optimized pipeline
pipeline = shadow.pipeline

# Save it to file
pickle.dump(pipeline, open("final_pipeline.pkl", "wb"))

# Export the best pipelines

# Serialize the pipeline
x_proc_best = shadow.X_preparer.serialize()
y_proc_best = shadow.y_preparer.serialize()

# Write the serialized pipelines to file
json.dump(x_proc_best, open("x_proc_best.json", "w"), indent=2)
json.dump(y_proc_best, open("y_proc_best.json", "w"), indent=2)
```

Once you have a preprocessor pipeline that you are happy with, you should attempt to optimize the model. The AutoEstimator will be good for this as it will automatically do model selection and hyperparameter optimization. To do this, construct the Foreshadow object in the same way as above, using the optimized JSON configuration, but instead of passing in an sklearn estimator and optimizer, leave those fields as default. This will force Foreshadow to use the defaults which automatically chooses either TPOT (regression) or AutoSklearn (classification) to fit the preprocessed data without any of their in-built feature engineering. When serializing the pipeline, Foreshadow will automatically choose the pipeline with the highest cross-validation score.

This will take a long time to execute... get yourself a cup of coffee or tea, sit back, and relax

Great! Now you have an optimized sklearn pipeline that you can share, load, manipulate, and inspect!

5.1.2 Foreshadow

Foreshadow is the primary object and interface for the Foreshadow framework. By default, Foreshadow creates a `Preprocessor` object for both the input data and the target vector.

It also automatically determines whether the target data is categorical or numerical and determines whether to use a Classification estimator or a Regressor. By default Foreshadow will either pick TPOT for regression or auto-sklearn for classification.

This pipeline is then fit and exposed via the `pipeline` object attribute.

Foreshadow can optionally take in a `Preprocessor` object for the input data, a `Preprocessor` object for the target vector, a `sklearn.base.BaseEstimator` object to fit the preprocessed data, and a `sklearn.grid_search.BaseSearchCV` class to optimize the available hyperparameters.

Here is an example of a fully defined *`Foreshadow`* object

```
shadow = fs.Foreshadow(X_preparer=Preprocessor(), y_preparer=Preprocessor(),␣
↪estimator=AutoEstimator(), optimizer=None)
```

This code is equivalent to the `fs.Foreshadow()` definition but explicitly defines each component. In order to disable one or more of these components simply pass `False` to the named parameter (Note that the default `None` automatically initializes the above).

`AutoEstimator` is automatically defined as the estimator for Foreshadow. This estimator detects the problem type (classification or regression) and then either uses TPOT or Auto-Sklearn to serve as the estimator. The preprocessing methods are stripped from TPOT and Auto-Sklearn when they are used in this manner as we favor our own Preprocessor over their methods. As such these two frameworks will only perform model selection and estimator hyperparameter optimization by default.

NOTE: Future work includes implementing TPOT and AutoSklean's optimizers into this platform such that they can be used for both model selection and optimizing hyperparameters for the feature engineering aspects. Until then, however, they will only optimize the model as they are blind to the earlier parts of the pipeline.

Foreshadow, acting as an estimator is also capable of being used in a `sklearn.pipeline.Pipeline` object. For example:

```
pipeline = Pipeline([("estimator", Foreshadow())])
pipeline.fit(X_train, y_train)
pipeline.score(X_test, y_test)
```

By passing an optimizer into Foreshadow, it will attempt to optimize the pipeline it creates by extracting all the hyperparameters from the preprocessors and the estimator and passing them into the optimizer object along with the partially fit pipeline. This is a potentially long-running process and is not reccomended to be used with estimators such as TPOT or AutoSklearn which also do their own optimization.

5.1.3 Preprocessor

The Preprocessor object provides the feature engineering capabilities for the Foreshadow framework. Like the *`Foreshadow`* object, the `Preprocessor` is capable of being used as a standalone object to perform feature engineering, or it can be used in a `Pipeline` as a Transformer to perform preprocessing for an estimator.

In its most basic form, a Preprocessor can be initialized with no parameters as `fs.Preprocessor()` in which all defaults will be applied. Ideally, a default preprocessor will be able to produce an acceptable pipeline for feature engineering.

The preprocessor performs the following tasks in order

1. Load configuration (if present)
2. Iterate columns and match Intents
3. Execute single-pipelines on columns in parallel
4. Execute multi-pipelines on columns in series

Intents

Preprocessor works by using *Intents*. These classes describe a type of feature that a dataset could possibly contain. For example, we have a `NumericIntent` and a `CategoricalIntent`.

Depending on the characterization of the data performed by the `is_intent()` class method, *each Intent individually determines if it applies to a particular feature in the dataset.* However, it is possible for multiple intents to match to a feature. In order to resolve this, Preprocessor uses a hierarchical structure defined by the superclass (parent) and `children` attributes of and intent. There is also a priority order defined in each intent to break ties at the same level.

This tree-like structure which has `GenericIntent` as its root node is used to prioritize Intents. Intents further down the tree more precisely define a feature and intents further to the right hold a higher priority than those to the left, thus the Intent represented by the right-most node of the tree that matches will be selected.

Each Intent contains a `multi-pipeline` and a `single-pipeline`. These objects are lists of tuples of the form `[('name', TransformerObject()),...]` and are used by Preprocessor to construct sklearn Pipeline objects.

Single Pipeline

The single pipeline defines operations (transformations of data) on a single column of the dataset matched to a specific intent. For example, in the Boston Housing dataset, the `'CRIM'` column could match to the `NumericIntent` in which the single pipeline within that Intent would be executed on that feature.

This process is highly parallelized interally.

Multi Pipeline

Intents also contain a `multi-pipeline` which operates on all columns of data of a given intent simultaneously. For example, in the Boston Housing dataset, the `'CRIM'` feature (per capita crime rate), the `'RM'` feature (average rooms per house), and the `'TAX'` feature (property tax rate) could be matched to `NumericIntent` in which the corresponding multi-pipeline would apply transformers across the columns such as feature reduction methods like PCA or methods of inference such as Multiple Imputation.

Additionally, while single pipelines are applied on an exclusive basis, multiple pipelines are applied on an inclusive basis. All multiple pipelines in the Intent hierarchy are executed on matching columns in the order from lowest (most-specific) intent, to the highest (most-general) intent.

NOTE: All transformers within a single or multi pipeline can access the entire current dataframe as it stands via `fit_params['full_df']` in fit or fit_transform

Smart Transformers

Smart Transformers are a special subclass of sklearn Transformers derived from the `SmartTransformer` base class. These transformers do not perform operations on data themselves but instead return a Transformer object at the time of pipeline execution. This allows pipelines to make logical decisions about actions to perform on features in real-time.

Smart Transformers make up the essence of single and multi pipelines in Intents as they allow conditional operations to be performed on data depending on any statistical analysis or hypothesis testing. Smart transformers can be overriden using the `override` attribute which takes in a string which is capable of being resolved as an internal transformer in the Foreshadow library, an external transfomer from sklearn or another smart transformer. The attributes of this override can be set via the `set_params()` methods for which all parameters other than the `override` parameter itself will be passed to the override object.

To use a smart transformer outside of the Intent / Foreshadow environment simply use it exactly as a sklearn transformer. When you call `fit()` or `fit_transform()` it automatically resolves which transformer to use by interally calling the `_get_transformer()` overriden method.

Note: Arguments passed into the constructor of a smart transformer will be passed into the fit function of the transformer it resolves to. This is meant to primarily be used alongside the override argument.

5.1.4 Configuration

The configurability is by far the most powerful aspect of this framework. Through configuration, data scientists can quickly iterate on pipelines generated by Foreshadow and Preprocessor. Preprocessors take a python dictionary configuration in the `from_json` named parameter in the constructor. This dictionary can be used to override all decision -making processes used by Preprocessor.

An example configuration for processing the Boston Housing dataset is below. We will step through this one by one and demonstrate all the capabilities.

```
{
    "columns": {
        "crim": {
            "intent": "GenericIntent",
            "pipeline": [{
                "transformer": "StandardScaler",
                "name": "Scaler",
                "parameters": {
                    "with_mean": false
                }
            }]
        },
        "indus": {
            "intent": "GenericIntent"
        }
    },
    "postprocess": [{
        "name": "pca",
        "columns": ["age"],
        "pipeline": [{
            "transformer": "PCA",
            "name": "PCA",
            "parameters": {
                "n_components": 2
            }
        }]
    }],
    "intents": {
        "NumericIntent": {
            "single": [{
                "transformer": "Imputer",
                "name": "impute",
                "parameters": {
                    "strategy": "mean"
                }
            }],
            "multi": []
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The configuration file is composed of a root dictionary containing three hard-coded keys: `columns`, `postprocess`, and `intents`. First, we will examine the `columns` section.

Column Override

```
"columns": {
    "crim": {
        "intent": "GenericIntent",
        "pipeline": [{
            "transformer": "StandardScaler",
            "name": "Scaler",
            "parameters": {
                "with_mean": false
            }
        }]
    },
    "indus": {
        "intent": "GenericIntent"
    }
}
```

This section is a dictionary containing two keys, each of which are columns in the Boston Housing set. First we will look at the value of the `"crim"` key which is a dict.

```
{
    "intent": "GenericIntent",
    "pipeline": [{
        "transformer": "StandardScaler",
        "name": "Scaler",
        "parameters": {
            "with_mean": false
        }
    }]
}
```

Here we can see that this column has been assigned the intent `"GenericIntent` and the pipeline `[{"transformer": "StandardScaler", "name": "Scaler", "parameters": {"with_mean":false}}]`

This means that regardless of how Preprocessor automatically assigns Intents, the intent GenericIntent will always be assigned to the crim column. It also means that regardless of what intent is assigned to the column (this value is still important for multi-pipelines), the Preprocessor will always use this hard-coded pipeline to process that column. The column would still be processed by its initially identifited multi-pipeline unless explicitly overridden.

The pipeline itself is defined by the following standard `[{"transformer":class, "name":name, "parameters":{param_key: param_value, ...}], ...]` When preprocessor parses this configuration it will create a Pipeline object with the given transformers of the given class, name, and parameters. For example, the preprocessor above will look something like `sklearn.pipeline.Preprocessor([('Scaler', StandardScaler(with_mean=False))])` Any class implementing the sklearn Transformer standard (including SmartTransformer) can be used here.

That pipeline object will be fit on the column crim and will be used to transform it.

Moving on to the `"indus"` column defined by the configuration. We can see that it has an intent override but not a pipeline override. This means that the default `single_pipeline` for the given intent will be used to process that column. By default the serialized pipeline will have a list of partially matching intents under the "all_matched_intents" dict key. These can likely be substituted into the Intent name with little or no compatibility issues.

Intent Override

```
"intents": {
    "NumericIntent": {
        "single": [{
            "transformer": "Imputer",
            "name": "impute",
            "parameters": {
                "strategy": "mean"
            }
        }],
        "multi": []
    }
}
```

Next, we will examine the `intents` section. This section is used to override intents globally, unlike the columns section which overrode intents on a per-column basis. Any changes to intents defined in this section will apply across the entire Preprocessor pipeline. However, individual pipelines defined in the columns section will override pipelines defined here.

The keys in this section each represent the name of an intent. In this example, `NumericIntent` is being overridden. The value is a dictionary with the keys `"single"` and `"multi"` respresent the single and multi pipeline overrides. The value of these pipelines is parsed through the same mechanism as the pipelines in the columns section.

If a pipeline is empty such as the multi pipeline is above, it will be removed from the final pipeline. However, if the multi key is ommitted from the configuration file, then the default multi pipeline for that intent will be used.

In this case, for all NumericIntent columns, by default, the pipeline `Pipeline([('impute', Imputer(strategy=mean))])` will be executed on the column. No multi-pipeline will be executed on columns of NumericIntent.

Postprocessor Override

```
{
    "postprocess": [{
        "name": "pca",
        "columns": ["age"],
        "pipeline": [{
            "class": "PCA",
            "name": "PCA",
            "parameters": {
                "n_components": 2
            }
        }]
    }]
}
```

Finally, in the `postprocess` section of the configuration, you can manually define pipelines to execute on columns of your choosing. The content of this section is a list of dictionaries of the form `[{"name":name, "columns":[cols, ...], "pipeline":pipeline}, ...]`. Each list defines a pipeline that will execute on certain columns. These processes execute after the intent pipelines!

IMPORTANT There are two ways of selecting columns through the cols list. By default, specifying a column, or a list of columns, will automatically select the columns in the data frame that are computed columns deriving from that column. For example, in the list above, all columns derived from the `age` column will be passed into the PCA transformer and reduced to 2 components. To override this behavior and select columns by their name at the current stage in the process, prepend a dollar sign to the column name. For example `["$age_scale_0", "$indus_encode_0", "$indus_encode_1"]`

Through overriding these various components, any combination of feature engineering can be achieved. To generate this configuration dictionary after fitting a Preprocessor or a Foreshadow object, run the `serialize()` method on the Preprocessor object or on `Foreshadow.X_preparer` or `y_preparer`. That dictionary can be programmatically modified in python or can be serialized to JSON where it can be modified by hand. By default the output of `serialize()` will fix all feature engineering to be constant. To only enforce sections of the configuration output from `serialize()` simply copy and paste the relevant sections into a new JSON file.

5.1.5 Hyperparameter Tuning

Foreshadow also supports hyperparameter tuning through two mechanisms. By default, Foreshadow will use *`AutoEstimator`* as an estimator in the pipeline. This estimator will automatically choose either TPOT, for regression problems or AutoSklearn for classification problems. It also strips all feature engineering and preprocessing from these two frameworks. This, in effect, uses TPOT and AutoSklearn only for model selection and model hyperparameter optimization. These estimators are not passed hyperparameters from the Preprocessor and thus will not optimize them.

The second method of hyperparameter tuning is to use a vanilla sklearn estimator when declaring foreshadow (such as XGBoost or LogisticRegression) and also pass in a `BaseSearchCV` class into the `optimizer` parameter. This will use the provided optimizer to perform a parameter search on both the preprocessing and the model at the same time. The parameter search space for this configuration is defined in two locations.

Default Dictionary

The first is in `foreshadow/optimizers/param_mapping.py` which contains a dictionary like:

```
config_dict = {
    "StandardScaler.with_std": [True, False]
    "StandardScaler.with_mean": [True, False]
    }
```

This dictionary contains keys and values of the form `ClassName.attribute: iterator(test_values)` If any items in the pipeline match the classname.attribute selector then that attribute will be added as a hyperparameter with the values of the iterator (list, generator, etc.) as the search space.

NOTE: In the future, this dictionary will be able to be passed in to Foreshadow, for now it must be modified manually if changes wish to be made.

JSON Combinations Config

If you wish to manually define spaces to search for the Preprocessor those can be defined in the configuration dictionary of the preprocessor in the `combinations` section. This is what a combinations section looks like.

```
{
    "columns": {
        "crim": {
            "intent": "GenericIntent",
```

(continues on next page)

(continued from previous page)

```
            "pipeline": [{
                "transformer": "StandardScaler",
                "name": "Scaler",
                "parameters": {
                    "with_mean": false
                }
            }]
        },
        "indus": {
            "intent": "GenericIntent"
        }
    },

    "postprocess": [],

    "intents": {},

    "combinations": [{
        "columns.crim.pipeline.0.parameters.with_mean": "[True, False]",
        "columns.crim.pipeline.0.name": "['Scaler', 'SuperScaler']"
    }]

}
```

This section of the configuration file is a list of dictionaries. Each dictionary represents a single parameter space definition that should be searched. Within these dictionaries each key is an identifier for a value in another part of the configuration file. For example `columns.crim.1.0.2.with_mean` will identify the *columns* key and then the *crim* key, then the 1th index of that list, the 0th index of the next list, the 2nd index of the next list, and finally the *with_mean* key of that dictionary. Each value is a string of **python code** that will be evaluated to create an **iterator** object that will be used to generate the parameter space.

In this example 4 combinations will be searched:

- `StandardScaler(with_mean=False, name="Scaler")`
- `StandardScaler(with_mean=True, name="Scaler")`
- `StandardScaler(with_mean=False, name="SuperScaler")`
- `StandardScaler(with_mean=True, name="SuperScaler")`

In addition to any search parameters defined in the default search space dictionary above

5.2 Frequently Asked Questions

Test page

CHAPTER 6

The Developer Guide

6.1 Developers Guide

Thank you for taking the time to contribute and reading this page, any and all help is appreciated!

6.1.1 Setting up the Project From Source

General Setup

1. Clone the project down to your computer

```
$ git clone https://github.com/georgianpartners/foreshadow.git
$ cd foreshadow
$ git checkout development
```

2. Install and setup pyenv and pyenv-virtualenv

   Follow the instructions on their pages or use homebrew if you have a Mac

```
$ brew install pyenv
$ brew install pyenv-virtualenv
```

   Make sure to add the following lines to your `.bash_profile`

```
export PYENV_ROOT="$HOME/.pyenv"
export PATH="$PYENV_ROOT/bin:$PATH"
if command -v pyenv 1>/dev/null 2>&1; then
  eval "$(pyenv init -)"
fi
eval "$(pyenv virtualenv-init -)"
```

   Restart your shell session for the changes to take effect and perform the following setup **in the root directory of the project**. This sets up a convenient virtualenv that automatically activates in the root

of your project. (Note: there is a known error with pyenv. Also, you may need to change the file path depending on your version or you may not even need to do that step.

```
$ open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_
↪macOS_10.14.pkg
$ pyenv install 3.6.8
$ pyenv global 3.6.8
$ pyenv virtualenv -p python3.6 3.6.8 venv
$ pyenv local venv 3.6.8
```

3. Install poetry package manager

```
(venv) $ pyenv shell system
$ curl -sSL https://raw.githubusercontent.com/sdispater/poetry/master/get-
↪poetry.py | python
$ pyenv shell --unset
```

Prepare for Autosklearn install Autosklearn was setup as an optional dependency as it can be sometimes difficult to install because of its requirement of xgboost. In order to have a development environment that passes all tests, autosklearn is required.

1. Install swig

   Use your package manager to install swig

```
(venv) $ brew install swig # (or apt-get)
```

2. Install gcc (MacOS only)

   Use your package manager to install gcc (necessary for xgboost)

```
(venv) $ brew install gcc@5 # (or apt-get)
```

Install all the packages and commit hooks When the project is installed through poetry both project requirements and development requirements are installed. Install commit-hooks using the pre-commit utility.

```
(venv) $ poetry install -v
(venv) $ export CC=gcc-5; export CXX=g++-5;
(venv) $ poetry install -E dev
(venv) $ poetry run pre-commit install
```

Configure PlantUML

```

```

(venv) $ brew install plantuml # MacOS (requires brew cask install adoptopenjdk) (venv) $ sudo apt install plantuml # Linux

Making sure everything works

1. Run pytest to make sure you're good to go

```
(venv) $ poetry run pytest
```

2. Run tox to run in supported python versions (optional)

```
(venv)$ poetry run tox -r # supply the -r flag if you changed the dependencies
```

3. Run make html in foreshadow/doc to build the documentation (optional)

```
(venv) $ poetry run make html
```

If all the tests pass you're all set up!

Note: Our platform also includes integration tests that asses the overall performance of our framework using the default settings on a few standard ML datasets. By default these tests are not executed, to run them, set an environmental variable called *FORESHADOW_TESTS* to *ALL*

Suggested development work flow

1. Create a branch off of development to contain your change

```
(venv) $ git checkout development
(venv) $ git checkout -b {your_feature}
```

2. Run pytest and pre-commit while developing This will help ensure something hasn't broken while adding a feature. Pre-commit will lint the code before each commit.

```
$ poetry run pytest
$ poetry run pre-commit run --all-files
```

3. Run tox to test your changes across versions Make sure to add test cases for your change in the appropriate folder in foreshadow/tests and run tox to test your project across python 3.5 and 3.6

```
$ poetry run tox
```

4. Submit a pull request This can be tricky if you have cloned the project instead of forking it but no worries the fix is simple. First go to the project page and **fork it there**. Then do the following.

```
(venv) $ git remote add upstream https://github.com/georgianpartners/
↪foreshadow.git
(venv) $ git remote set-url origin https://github.com/{YOUR_USERNAME}/
↪foreshadow.git
(venv) $ git push origin {your_feature}
```

Now you can go to the project on your github page and submit a pull request to the main project.

Note: Make sure to submit the pull request against the development branch.

6.1.2 Adding Transformers

Adding transformers is quite simple. Simply write a class with the *fit transform* and *inverse_transform* methods that extends `scikit_learn.base.BaseEstimator` and `sklearn.base.TransformerMixin`. Take a look at the structure below and modify it to suit your needs. We would recommend taking a look at the sklearn.preprocessing.RobustScaler source code for a good example.

```
from foreshadow.base import TransformerMixin, BaseEstimator
from sklearn.utils import check_array

class CustomTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        X = check_array(X)
```

(continues on next page)

(continued from previous page)

```
        return self

    def transform(self, X, y=None):
        X = check_array(X, copy=True)
        # modify input based on fit here
        return X

    def inverse_transform(self, X):
        X = check_array(X, copy=True)
        # if applicable, write inverse transform here
        return X
```

After writing your transformer make sure place it in the internals folder in its own file with the associated tests for the transformer in the mirrored test directory and you are all set. If you want to add an external transformer that is not already supported by foreshadow submit a pull request with the appropriate modification to the *externals.py* file in transformers.

6.1.3 Adding Smart Transformers

Building smart transformers is even easier than build transformers. Simply extend `SmartTransformer` and implement the `_get_transformer()`. Modify the example below to suit your needs.

```
class CustomTransformerSelector(SmartTransformer):
    def _get_transformer(self, X, y=None, **fit_params):
        data = X.iloc[:, 0] # get single column to decide upon
        # perform some computation to determin the best transformer to choose
        return BestTransformer() # return an instance of the selected transformer
```

Add the smart transformer implementation to the bottom of the *smart.py* file and add the appropriate tests to the mirrored tests folder as well.

6.1.4 Adding Intents

Intents are where the magic of Foreshadow all comes together. You need to be thoughtful when adding an intent especially with respect to where your intent will slot into the intent tree. This positioning will determine the priority with which the intent is mapped to a column. You will need to subclass your intent off of the parent intent that you determine is the best fit. Intents should be constructed in the form matching *BaseIntent*.

You will need to set the `dtype`, `children`, `single_pipeline`, and `multi_pipeline` class attributes. You will also need to implement the `is_intent` classmethod. In most cases when adding an intent you can initialize `children` to an empty list. Set the `dtype` to the most appropriate initial form of that entering your intent.

Use the `single_pipeline` field to determine the transformers that will be applied to a **single** column that is mapped to your intent. Add a **unique** name describing each step that you choose to include in your pipeline. This field is represented as a list of PipelineTemplateEntry objects which are constructed using the following format *PipelineTemplateEntry([unique_name], [class], [can_operate_on_y])* The class name is either a singular transformer class, or a tuple of the form *([cls], {**args})* where args will be passed into the constructor of the transformer. The final boolean determines whether that transformer should be applied when operating on y-variables.

It is important to note the utility of smart transformers here as you can now include branched logic in your pipelines deciding between different individual transformers based on the input data at runtime. The `multi_pipeline` pipeline should be used to apply transformations to all columns of a specific intent after the single pipelines have been evaluated. The same rules for defining the pipelines themselves apply here as well.

The `is_intent` classmethod determines whether a specific column maps to an intent. Use this method to apply any heuristics, logic, or methods of determine whether a raw column maps to the intent that you are defining. Below is an example intent definition that you can modify to suit your needs.

The *`column_summary`* classmethod is used to generate statistical reports each time an intent operates on a columns allowing a user to examine how effective the intent will be in processing the data. These reports can be accessed by calling the `summarize` method after fitting the Foreshadow object.

Make **sure** to go to the parent intent and add your intent class name to the ordered `children` field in the order of priority among the previously defined intents. The last intent in this list will be the most preferred intent upon evaluation in the case of multiple intents being able to process a column.

Take a look at the `NumericIntent` implementation for an example of how to implement an intent.

6.1.5 Future Architecture Roadmap

In progress

6.2 Contributing

6.2.1 Project Setup

Foreshadow has one main master branch and feature branches if an when collaborative development is required. Each pypi version and their associated commit will be tagged in GitHub. Before each release a new branch will be created freezing that specific version. Pull requests are merged directly into master. This project follows the semantic versioning standard.

6.2.2 Issues

Please feel free to submit issues to github with any bugs you encounter, ideas you may have, or questions about useage. We only ask that you tag them appropriately as *bug fix*, *feature request*, *usage*. Please also follow the following format

````
#### Description
<!--
Example: DropFeatures fails on categorical features when assessing a string column
-->

#### Steps/Code to Reproduce
<!--
Please add the minimum code required to reproduce the issue if possible.
Example:
```python
import uuid
import numpy as np
import pandas as pd
from foreshadow.preprocessor import Preprocessor

cat1 = [str(uuid.uuid4()) for _ in range(40)]
cat2 = [str(uuid.uuid4()) for _ in range(40)]

input = pd.DataFrame({
    'col1': np.random.choice(cat1, 1000),
    'col2': np.random.choice(cat2, 1000)
````

(continues on next page)
````

(continued from previous page)

```
})

processor = Preprocessor()
output = processor.fit_transform(input)
```
If the code is too long, feel free to put it in a public gist and link it in the␣
↪issue: https://gist.github.com
-->

Expected Results
<!--
Please add the results that you would expect here.
Example: Error should not be thrown
-->

Actual Results
<!--
Please place the full traceback here, again use a gist if you feel that it is too␣
↪long.
-->

Versions
<!--
Please run the following snippet in your environment and paste the results here.

```python
import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import sklearn; print("Scikit-Learn", sklearn.__version__)
import pandas; print("Pandas", pandas.__version__)
import foreshadow; print("Foreshadow", foreshadow.__version__)
from foreshadow.utils import check_transformer_imports; check_transformer_imports()
```
-->

<!--Thank you for contributing!-->
```

### 6.2.3 How to Contribute: Pull Requests

We accept pull requests! Thank you for taking the time to read this. There are only a few guidelines before you get started. Make sure you have read the *Developers Guide* and have appropriately setup your project. Please make sure to do the following to appropriately create a pull request for this project.

1. Fork the project on GitHub
2. Setup the project following the instructions in the *Developers Guide* **using your fork**
3. Create a branch to hold your change

```
$ git checkout development
$ git checkout -b contribution_branch_name
```

4. Start making changes to this branch and remember to never work on the master branch.
```

5. Make sure to add tests for your changes to *foreshadow/tests/* and make sure to run those changes. You need to run these commands from the root of the project repository.

```
$ black foreshadow # required formatter
$ pytest
$ coverage html
$ open htmlcov/index.html
$ tox -r
```

6. If everything is green and looks good, you're ready to commit

```
$ git add changed_files
$ git commit # make sure use descriptive commit messages
$ git push -u origin contribution_branch_name
```

7. Go to the github fork page and submit your pull request against the **development** branch. Please use the following template for pull requests

```
<!--
Thanks you for taking the time to submit a pull request! Please take a look at some␣
↪guidelines before submitting a pull request: https://github.com/georgianpartners/
↪foreshadow/blob/development/doc/contrib.rst
-->

### Related Issue
<!--
Example: Fixes #7. See also #35.
Please use keywords (e.g., Fixes) to create link to the issues or pull requests
you resolved, so that they will automatically be closed when your pull request
is merged. See https://github.com/blog/1506-closing-issues-via-pull-requests
-->

### Description
<!--
Please add a narrative description of your the changes made and the rationale behind␣
↪them. If making an enhancement include the motivation and use cases addressed.
-->
```

CHAPTER 7

API

7.1 API Reference

7.1.1 Foreshadow

Core end-to-end pipeline, foreshadow.

class Foreshadow(*X_preparer=None*, *y_preparer=None*, *estimator=None*, *optimizer=None*, *optimizer_kwargs=None*)

An end-to-end pipeline to preprocess and tune a machine learning model.

Example

```
>>> shadow = Foreshadow()
```

Parameters

- **X_preparer** (`Preprocessor`, optional) – Preprocessor instance that will apply to X data. Passing False prevents the automatic generation of an instance.
- **y_preparer** (`Preprocessor`, optional) – Preprocessor instance that will apply to y data. Passing False prevents the automatic generation of an instance.
- **estimator** (`sklearn.base.BaseEstimator`, optional) – Estimator instance to fit on processed data
- **optimizer** (`sklearn.grid_search.BaseSeachCV`, optional) – Optimizer class to optimize feature engineering and model hyperparameters

X_preparer

Preprocessor object for performing feature engineering on X data.

Getter Returns Preprocessor object

Setter Verifies Preprocessor object, if None, creates a default Preprocessor

Type Preprocessor

y_preparer
Preprocessor object for performing scaling and encoding on Y data.

Getter Returns Preprocessor object

Setter Verifies Preprocessor object, if None, creates a default Preprocessor

Type Preprocessor

estimator
Estimator object for fitting preprocessed data.

Getter Returns Estimator object

Setter Verifies Estimator object. If None, an *AutoEstimator* object is created in place.

Type sklearn.base.BaseEstimator

optimizer
Optimizer class that will fit the model.

Performs a grid or random search algorithm on the parameter space from the preprocessors and estimators in the pipeline

Getter Returns optimizer class

Setter Verifies Optimizer class, defaults to None

fit (*data_df*, *y_df*)
Fit the Foreshadow instance using the provided input data.

Parameters

- **data_df** (DataFrame) – The input feature(s)
- **y_df** (DataFrame) – The response feature(s)

Returns The fitted instance.

Return type *Foreshadow*

predict (*data_df*)
Use the trained estimator to predict the response variable.

Parameters **data_df** (DataFrame) – The input feature(s)

Returns The response feature(s) (transformed if necessary)

Return type DataFrame

predict_proba (*data_df*)
Use the trained estimator to predict the response variable.

Uses the predicted confidences instead of binary predictions.

Parameters **data_df** (DataFrame) – The input feature(s)

Returns The probability associated with each response feature

Return type DataFrame

score (*data_df*, *y_df=None*, *sample_weight=None*)
Use the trained estimator to compute the evaluation score.

The scoding method is defined by the selected estimator.

Parameters

- **data_df** (DataFrame) – The input feature(s)
- **y_df** (DataFrame, optional) – The response feature(s)
- **sample_weight** (numpy.ndarray, optional) – The weights to be used when scoring each sample

Returns A computed prediction fitness score

Return type float

dict_serialize(*deep=False*)
Serialize the init parameters of the foreshadow object.

Parameters **deep** (*bool*) – If True, will return the parameters for this estimator recursively

Returns The initialization parameters of the foreshadow object.

Return type dict

classmethod dict_deserialize(*data*)
Deserialize the dictionary form of a foreshadow object.

Parameters **data** – The dictionary to parse as foreshadow object is constructed.

Returns A re-constructed foreshadow object.

Return type object

get_params(*deep=True*)
Get params for this object. See super.

Parameters **deep** – True to recursively call get_params, False to not.

Returns params for this object.

set_params(***params*)
Set params for this object. See super.

Parameters ****params** – params to set.

Returns See super.

7.1.2 dp

7.1.3 Intents

Intents package used by IntentMapper PreparerStep.

class Categoric
Defines a categoric column type.

confidence_computation = {<class 'foreshadow.metrics.MetricWrapper' with function 'num

fit(*X*, *y=None*, ***fit_params*)
Empty fit.

Parameters

- **X** – The input data
- **y** – The response variable
- ****fit_params** – Additional parameters for the fit

Returns self

transform(*X*, *y=None*)
Pass-through transform.

Parameters

- **X** – The input data
- **y** – The response variable

Returns The input column

classmethod column_summary(*df*)

class Numeric
Defines a numeric column type.

confidence_computation = {<class 'foreshadow.metrics.MetricWrapper' with function 'num

fit(*X*, *y=None*, ***fit_params*)
Empty fit.

Parameters

- **X** – The input data
- **y** – The response variable
- ****fit_params** – Additional parameters for the fit

Returns self

transform(*X*, *y=None*)
Convert a column to a numeric form.

Parameters

- **X** – The input data
- **y** – The response variable

Returns A column with all rows converted to numbers.

classmethod column_summary(*df*)

class Text
Defines a text column type.

confidence_computation = {<class 'foreshadow.metrics.MetricWrapper' with function 'num

fit(*X*, *y=None*, ***fit_params*)
Empty fit.

Parameters

- **X** – The input data
- **y** – The response variable
- ****fit_params** – Additional parameters for the fit

Returns self

transform(*X*, *y=None*)
Convert a column to a text form.

Parameters

- **X** – The input data
- **y** – The response variable

Returns A column with all rows converted to text.

classmethod column_summary(*df*)

class BaseIntent

Base for all intent definitions.

For each intent subclass a class attribute called *confidence_computation* must be defined which is of the form:

```
{
     metric_def: weight
}
```

classmethod get_confidence(*X*, *y=None*)

Determine the confidence for an intent match.

Parameters

- **X** – input DataFrame.
- **y** – response variable

Returns A confidence value bounded between 0.0 and 1.0

Return type float

classmethod column_summary(*df*)

7.1.4 Transformers

Internal Transformers

Smart Transformers

Transformer Bases

7.1.5 Estimators

Estimators provided by foreshadow.

class AutoEstimator(*problem_type=None*, *auto=None*, *include_preprocessors=False*, *estimator_kwargs=None*)

A wrapped estimator that selects the solution for a given problem.

By default each automatic machine learning solution runs for 1 minute but that can be changed through passed kwargs. Autosklearn is not required for this to work but if installed it can be used alongside TPOT.

Parameters

- **problem_type** (*str*) – The problem type, 'regression' or 'classification'
- **auto** (*str*) – The automatic estimator, 'tpot' or 'autosklearn'
- **include_preprocessors** (*bool*) – Whether include preprocessors in AutoML pipelines
- **estimator_kwargs** (*dict*) – A dictionary of args to pass to the specified auto estimator (both problem_type and auto must be specified)

problem_type
Type of machine learning problem.

Either *regression* or *classification*.

Returns self._problem_type

auto
Type of automl package.

Either *tpot* or *autosklearn*.

Returns self._auto, the type of automl package

estimator_kwargs
Get dictionary of kwargs to pass to AutoML package.

Returns estimator kwargs

configure_estimator(*y*)
Construct and return the auto estimator instance.

Parameters **y** – input labels

Returns autoestimator instance

fit(*X*, *y*)
Fit the AutoEstimator instance.

Uses the selected AutoML estimator.

Parameters

- **X** (*pandas.DataFrame or numpy.ndarray or list*) – The input feature(s)
- **y** (*pandas.DataFrame or numpy.ndarray or list*) – The response feature(s)

Returns The selected estimator

predict(*X*)
Use the trained estimator to predict the response.

Parameters **X** (*pandas.DataFrame or numpy.ndarray or list*) – The input feature(s)

Returns The response feature(s)

Return type pandas.DataFrame

predict_proba(*X*)
Use the trained estimator to predict the responses probabilities.

Parameters **X** (*pandas.DataFrame or numpy.ndarray or list*) – The input feature(s)

Returns The probability associated with each response feature

Return type pandas.DataFrame

score(*X*, *y*, *sample_weight=None*)
Use the trained estimator to compute the evaluation score.

Note: sample weights are not supported

Parameters

- **X** (*pandas.DataFrame or numpy.ndarray or list*) – The input feature(s)

- **y** (*pandas.DataFrame or numpy.ndarray or list*) – The response feature(s)
- **sample_weight** – sample weighting. Not implemented.

Returns A computed prediction fitness score

Return type float

class MetaEstimator(*estimator*, *preprocessor*)

Wrapper that allows data preprocessing on the response variable(s).

Parameters

- **estimator** – An instance of a subclass of `sklearn.base.BaseEstimator`
- **preprocessor** – An instance of `foreshadow.preprocessor.Preprocessor`

dict_serialize(*deep=False*)

Serialize the init parameters (dictionary form) of a transformer.

Parameters **deep** (*bool*) – If True, will return the parameters for this estimator recursively

Returns The initialization parameters of the transformer.

Return type dict

fit(*X*, *y=None*)

Fit the AutoEstimator instance using a selected AutoML estimator.

Parameters

- **X** (`pandas.DataFrame` or `numpy.ndarray` or list) – The input feature(s)
- **y** (`pandas.DataFrame` or `numpy.ndarray` or list) – The response feature(s)

Returns self

predict(*X*)

Use the trained estimator to predict the response.

Parameters **X** (pandas.DataFrame or `numpy.ndarray` or list) – The input feature(s)

Returns The response feature(s) (transformed)

Return type `pandas.DataFrame`

predict_proba(*X*)

Use the trained estimator to predict the response probabilities.

Parameters **X** (`pandas.DataFrame` or `numpy.ndarray` or list) – The input feature(s)

Returns The probability associated with each feature

Return type `pandas.DataFrame`

score(*X*, *y*)

Use the trained estimator to compute the evaluation score.

Note: sample weights are not supported

Parameters

- **X** (`pandas.DataFrame` or `numpy.ndarray` or list) – The input feature(s)
- **y** (`pandas.DataFrame` or `numpy.ndarray` or list) – The response feature(s)

Returns A computed prediction fitness score

Return type float

7.1.6 Optimizers

Foreshadow optimizers.

class ParamSpec (*fs_pipeline=None*, *X_df=None*, *y_df=None*)
Holds the specification of the parameter search space.

A search space is a dict or list of dicts. This search space should be viewed as one run of optimization on the foreshadow object. The algorithm for optimization is determined by the optimizer that is chosen. Hence, this specification is agnostic of the optimizer chosen.

A dict represents the set of parameters to be applied in a single run.

A list represents a set of choices that the algorithm (again, agnostic at this point) can pick from.

For example, imagine s as our top level object, of structure:

s (object)

.transformer (object) .attr

s has an attribute that may be optimized and in turn, that object has parameters that may be optimized. Below, we try two different transformers and try 2 different parameter specifications for each. Note that these parameters are specific to the type of transformer (StandardScaler does not have the parameter feature_range and vice versa).

[

{ “s__transformer”: “StandardScaler”, “s__transformer__with_mean”: [False, True],

}, {

“s__transformer”: “MinMaxScaler”, “s__transformer__feature_range”: [(0, 1), (0, 0.5)]
),

},

],

Here, the dicts are used to tell the optimizer where to values to set are. The lists showcase the different values that are possible.

convert (*key*, *replace_val=<function hp_choice>*)
Convert internal self.param_distributions to valid distribution.

Uses _replace_list to replace all lists with replace_val

Parameters

- **key** – key to use for top level hp.choice name
- **replace_val** – value to replace lists with.

get_params (*deep=True*)
Get the params for this object. Used for serialization.

Parameters **deep** – Does nothing. Here for sklearn compatibility.

Returns Members that need to be set for this object.

set_params (***params*)
Set the params for this object. Used for serialization.

Also used to init this object when automatic tuning is not used.

Parameters ****params** – Members to set from get_params.

Returns self.

class Tuner(*pipeline=None*, *params=None*, *optimizer=None*, *optimizer_kwargs={}*)
Tunes the Foreshadow object using a ParamSpec and Optimizer.

fit(*X*, *y*, ***fit_params*)
Optimize self.pipeline using self.optimizer.

Parameters

- **X** – input points
- **y** – input labels
- ****fit_params** – params to optimizer fit method.

Returns self

transform(*pipeline*)
Transform pipeline using best_pipeline.

Parameters **pipeline** – input pipeline

Returns best_pipeline.

class RandomSearchCV(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise'*, *return_train_score='warn'*, *max_tries=100*)
Optimize Foreshadow.pipeline and/or its sub-objects.

get(*optimizer*, ***optimizer_kwargs*)
Get optimizer from foreshadow.optimizers package.

Parameters

- **optimizer** – optimizer name or class
- ****optimizer_kwargs** – kwargs used in instantiation.

Returns Corresponding instantiated optimizer using kwargs.

7.1.7 Utils

Common Foreshadow utilities.

get_cache_path()
Get the cache path which is in the config directory.

Note: This function also makes the directory if it does not already exist.

Returns str; The path to the cache directory.

get_config_path()
Get the default config path.

Note: This function also makes the directory if it does not already exist.

Returns The path to the config directory.

Return type str

get_transformer(*class_name*, *source_lib=None*)
Get the transformer class from its name.

Note: In case of name conflict, internal transformer is preferred over external transformer import. This should only be using in internal unit tests, get_transformer from serialization should be preferred in all other cases. This was written to decouple registration from unit testing.

Parameters

- **class_name** (*str*) – The transformer class name
- **source_lib** (*str*) – The string import path if known

Returns Imported class

Raises TransformerNotFound – If class_name could not be found in internal or external transformer library pathways.

check_df(*input_data*, *ignore_none=False*, *single_column=False*, *single_or_empty=False*)
Convert non dataframe inputs into dataframes.

Parameters

- **input_data** (pandas.DataFrame, numpy.ndarray, list) – input to convert
- **ignore_none** (*bool*) – allow None to pass through check_df
- **single_column** (*bool*) – check if frame is of a single column and return series
- **single_or_empty** (*bool*) – check if the frame is a single column or an empty DF.

Returns Converted and validated input dataframes

Return type DataFrame

Raises

- ValueError – Invalid input type
- ValueError – Input dataframe must only have one column

check_series(*input_data*)
Convert non series inputs into series.

This is function is to be used in situations where a series is expected but cannot be guaranteed to exist. For example, this function is used in the metrics package to perform computations on a column using functions that only work with series.

Note: This is not to be used in transformers as it will break the standard that enforces only DataFrames as input and output for those objects.

Parameters **input_data** (*iterable*) – The input data

Returns pandas.Series

Raises

- ValueError – If the data could not be processed
- ValueError – If the input is a DataFrame and has more than one column

check_module_installed(*name*)
Check whether a module is available for import.

Parameters **name** (*str*) – module name

Returns Whether the module can be imported

Return type bool

check_transformer_imports(*printout=True*)
Determine which transformers were automatically imported.

Parameters **printout** (*bool, optional*) – Whether to output to stdout

Returns A tuple of the internal transformers and the external transformers

Return type tuple(list)

is_transformer(*value*, *method='isinstance'*)
Check if the class is a transformer class.

Parameters

- **value** – Class or instance
- **method** (*str*) – Method of checking. Options are *'issubclass'* or *'isinstance'*

Returns True if transformer, False if not.

Raises ValueError – if method is neither issubclass or isinstance

is_wrapped(*transformer*)
Check if a transformer is wrapped.

Parameters **transformer** – A transformer instance

Returns True if transformer is wrapped, otherwise False.

Return type bool

dynamic_import(*attribute*, *module_path*)
Import attribute from module found at module_path at runtime.

Parameters

- **attribute** – the attribute of the module to import (class, function, ...)
- **module_path** – the path to the module.

Returns attribute from module_path.

mode_freq(*s*, *count=10*)

get_outliers(*s*, *count=10*)

standard_col_summary(*df*)

class ConfigureColumnSharerMixin
Mixin that configure column sharer.

configure_column_sharer(*column_sharer*)
Configure the column sharer attribute if exists.

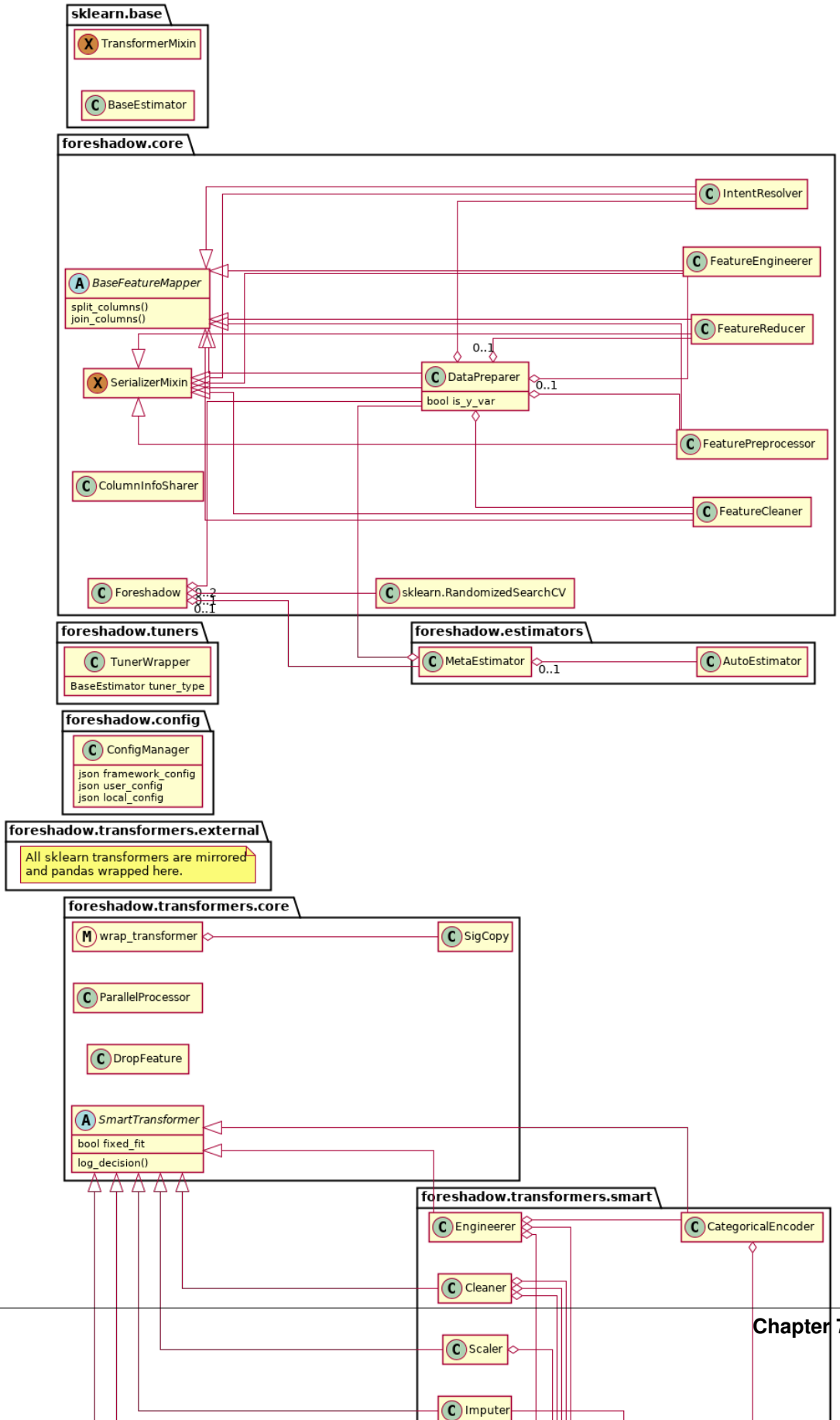
Parameters **column_sharer** – a column sharer instance

7.1.8 Core

7.2 Project Architecture

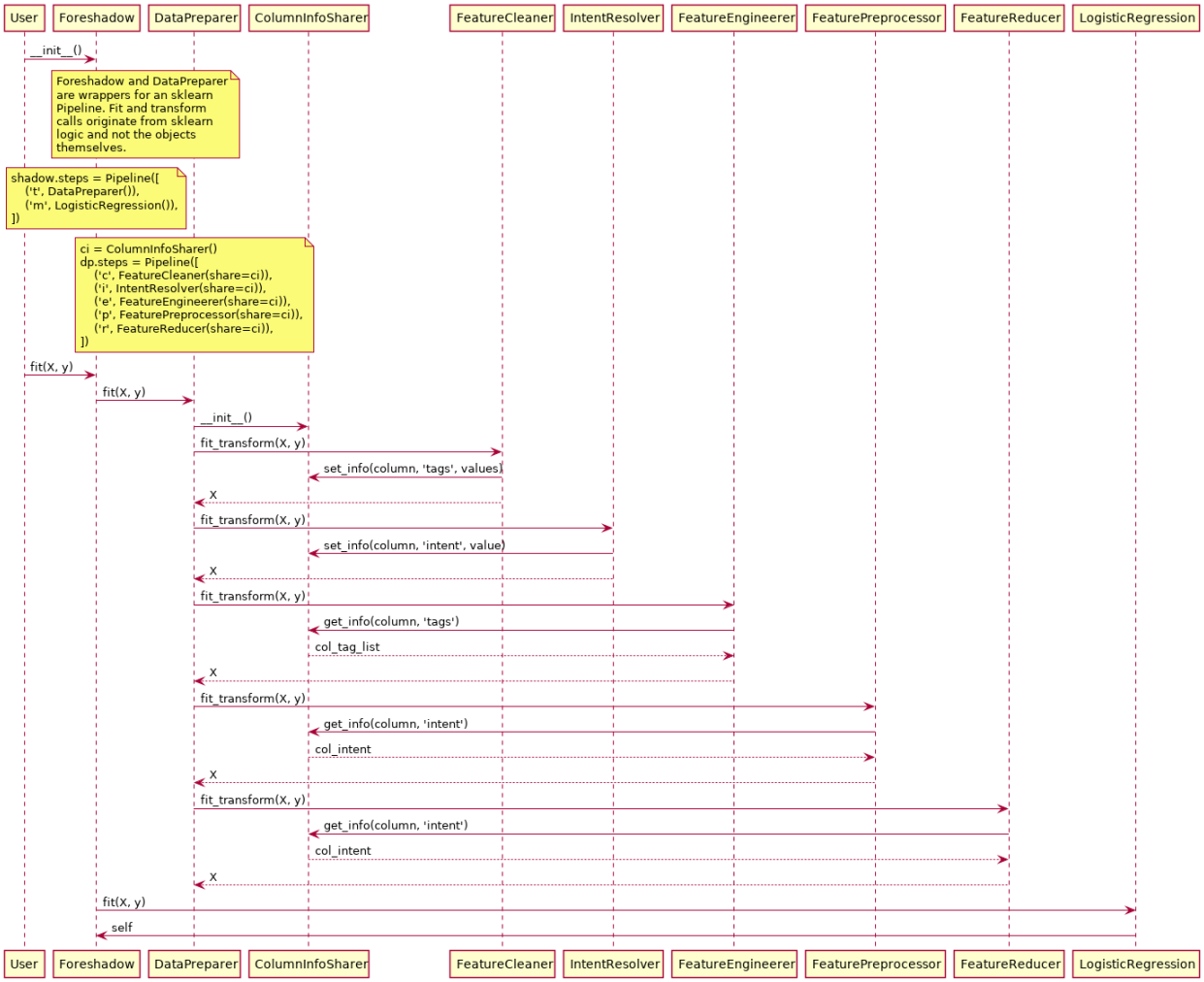
Note: Open the diagram in a new tab to see the full details.

7.2.1 UML Class Diagram

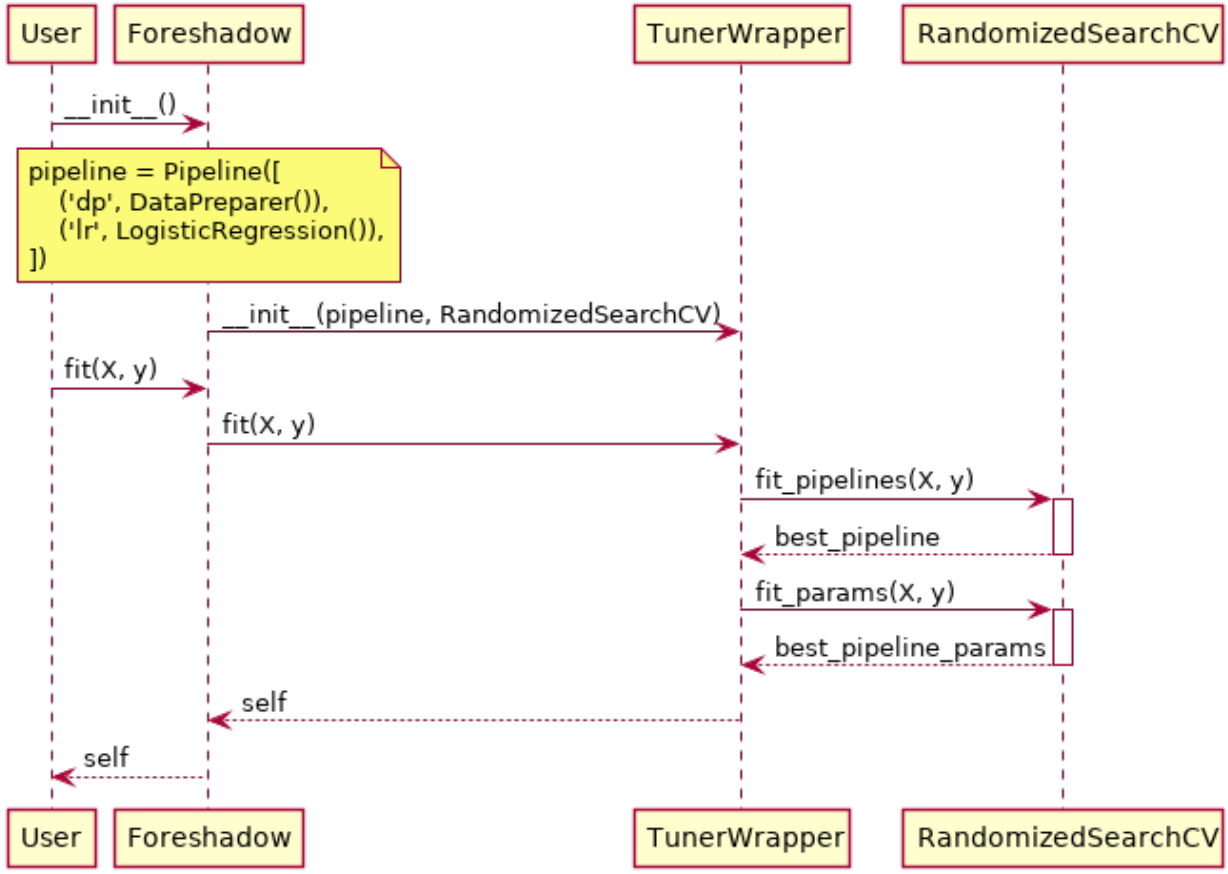


7.2.2 UML Sequence Diagrams

Main Sequence Diagram



Hyperparameter Optimization Sequence Diagram



CHAPTER 8

Changelog

8.1 Foreshadow 0.2.1 (2019-09-26)

8.1.1 Features

- Bug fix of pick_transformer may transform dataframe in place, causing inconsistency between the data and intended downstream logic. (bug-fix)

8.2 Foreshadow 0.2.0 (2019-09-24)

8.2.1 Features

- Add feature_summarizer to produce statistics about the data after intent resolving to show the users why such decisions are made. (data-summarization)
- Foreshadow is able to run end-to-end with level 1 optimization with the tpot auto-estimator. (level1-optimization)
- Add Feature Reducer as a passthrough transformation step. (pass-through-feature-reducer)
- Multiprocessing: 1. Enable multiprocessing on the dataset. 2. Collect changes from each process and update the original columnsharer. (process-safe-columnsharer)
- Serialization and deserialization: 1. Serialization of the foreshadow object in a non-verbose format. 2. Deserialization of the foreshadow object. (serialization)
- Adding two major components: 1. usage of metrics for any statistic computation 2. changing functionality of wrapping sklearn transformers to give them DataFrame capabilities. This now uses classes and metaclasses, which should be easier to maintain (#74)
- Adding ColumnSharer, a lightweight wrapper for a dictionary that functions as a cache system, to be used to pass information in the foreshadow pipeline. (#79)

- Creating DataPreparer to handle data preprocessing. Data Cleaning is the first step in this process. (#93)
- Adds skip resolve functionality to SmartTransformer, restructure utils, and add is_wrapped to utils (#95)
- Add serializer mixin and resture package import locations. (#96)
- Add configuration file parser. (#99)
- Add Feature Engineerer as a passthrough transformation step. (#112)
- Add Intent Mapper and Metric wrapper features. (#113)
- Add Preprocessor step to DataPreparer (#118)
- Create V2 architecture shift. (#162)

8.3 Foreshadow 0.1.0 (2019-06-28)

8.3.1 Features

- Initial release. (#71)

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

foreshadow.estimators, 35
foreshadow.foreshadow, 31
foreshadow.intents, 33
foreshadow.optimizers, 38
foreshadow.utils, 39

Index

A

auto (*AutoEstimator attribute*), 36
AutoEstimator (*class in foreshadow.estimators*), 35

B

BaseIntent (*class in foreshadow.intents*), 35

C

Categoric (*class in foreshadow.intents*), 33
check_df() (*in module foreshadow.utils*), 40
check_module_installed() (*in module foreshadow.utils*), 41
check_series() (*in module foreshadow.utils*), 40
check_transformer_imports() (*in module foreshadow.utils*), 41
column_summary() (*foreshadow.intents.BaseIntent class method*), 35
column_summary() (*foreshadow.intents.Categoric class method*), 34
column_summary() (*foreshadow.intents.Numeric class method*), 34
column_summary() (*foreshadow.intents.Text class method*), 35
confidence_computation (*Categoric attribute*), 33
confidence_computation (*Numeric attribute*), 34
confidence_computation (*Text attribute*), 34
configure_column_sharer() (*ConfigureColumnSharerMixin method*), 41
configure_estimator() (*AutoEstimator method*), 36
ConfigureColumnSharerMixin (*class in foreshadow.utils*), 41
convert() (*ParamSpec method*), 38

D

dict_deserialize() (*foreshadow.foreshadow.Foreshadow class method*), 33
dict_serialize() (*Foreshadow method*), 33
dict_serialize() (*MetaEstimator method*), 37
dynamic_import() (*in module foreshadow.utils*), 41

E

estimator (*Foreshadow attribute*), 32
estimator_kwargs (*AutoEstimator attribute*), 36

F

fit() (*AutoEstimator method*), 36
fit() (*Categoric method*), 33
fit() (*Foreshadow method*), 32
fit() (*MetaEstimator method*), 37
fit() (*Numeric method*), 34
fit() (*Text method*), 34
fit() (*Tuner method*), 39
Foreshadow (*class in foreshadow.foreshadow*), 31
foreshadow.estimators (*module*), 35
foreshadow.foreshadow (*module*), 31
foreshadow.intents (*module*), 33
foreshadow.optimizers (*module*), 38
foreshadow.utils (*module*), 39

G

get() (*in module foreshadow.optimizers*), 39
get_cache_path() (*in module foreshadow.utils*), 39
get_confidence() (*foreshadow.intents.BaseIntent class method*), 35
get_config_path() (*in module foreshadow.utils*), 39
get_outliers() (*in module foreshadow.utils*), 41
get_params() (*Foreshadow method*), 33
get_params() (*ParamSpec method*), 38
get_transformer() (*in module foreshadow.utils*), 40

I

is_transformer() (*in module foreshadow.utils*), 41
is_wrapped() (*in module foreshadow.utils*), 41

M

MetaEstimator (*class in foreshadow.estimators*), 37
mode_freq() (*in module foreshadow.utils*), 41

N

Numeric (*class in foreshadow.intents*), 34

O

optimizer (*Foreshadow attribute*), 32

P

ParamSpec (*class in foreshadow.optimizers*), 38
predict() (*AutoEstimator method*), 36
predict() (*Foreshadow method*), 32
predict() (*MetaEstimator method*), 37
predict_proba() (*AutoEstimator method*), 36
predict_proba() (*Foreshadow method*), 32
predict_proba() (*MetaEstimator method*), 37
problem_type (*AutoEstimator attribute*), 35

R

RandomSearchCV (*class in foreshadow.optimizers*), 39

S

score() (*AutoEstimator method*), 36
score() (*Foreshadow method*), 32
score() (*MetaEstimator method*), 37
set_params() (*Foreshadow method*), 33
set_params() (*ParamSpec method*), 38
standard_col_summary() (*in module foreshadow.utils*), 41

T

Text (*class in foreshadow.intents*), 34
transform() (*Categoric method*), 34
transform() (*Numeric method*), 34
transform() (*Text method*), 34
transform() (*Tuner method*), 39
Tuner (*class in foreshadow.optimizers*), 39

X

X_preparer (*Foreshadow attribute*), 31

Y

y_preparer (*Foreshadow attribute*), 32