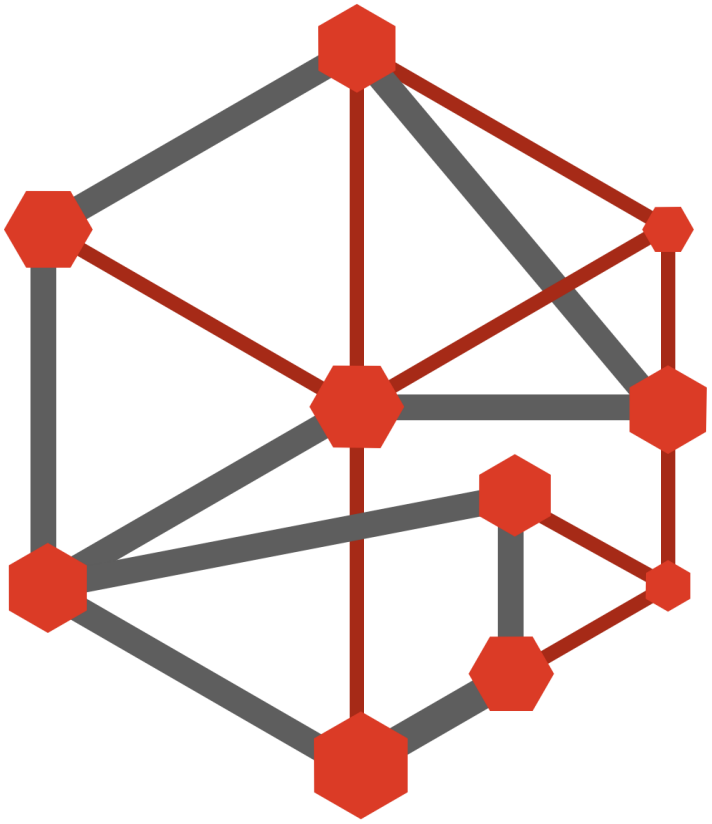


HEXADO1

SOFTWARE DEVELOPMENT TEAM

Aidan Zorbas Dawson Brown Jenny Yu

Jingrun Long Kara Autumn Jiang Vanessa Pierre

Scikit-learn Architecture Report

Assignment 1 Deliverable

Table of Contents

Table of Contents **3**

Scikit-learn Architecture **4**

Overview 4

External Dependencies 4

Package Structure 5

Estimators, Predictors and Transformers 6

Estimators 6

Predictors 6

Transformer 6

Aggregate Types 6

Structural UML Diagram 7

Scikit-learn Design Patterns **8**

Factory Design Pattern 8

Design Pattern Overview 8

Factory in scikit-learn 8

Iterator Design Pattern **10**

Design Pattern Overview 10

Iterator in scikit-learn 10

Strategy Design Pattern **13**

Design Pattern Overview 13

Advantages 13

Disadvantages 13

Strategy in scikit-learn 13

Scikit-learn Architecture

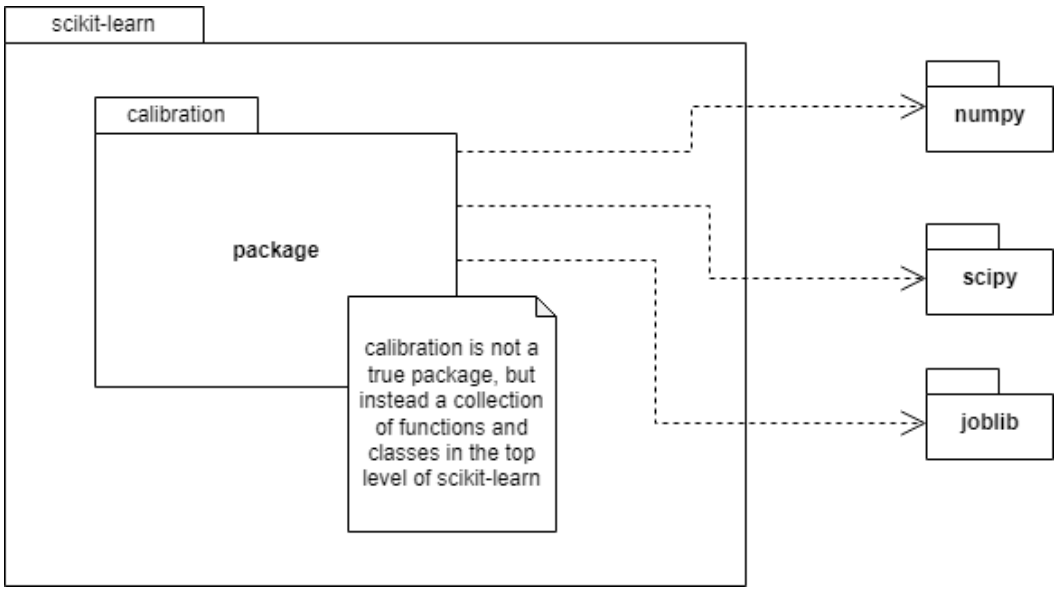
Overview

From an overarching perspective, scikit-learn provides a library of machine learning and data processing functions built on a structure of estimators, predictors, and transformers. Almost every module in the library is a representation of at least one of these three interfaces, and the methods they expose are the primary means through which users interact with the library. These functions provide end-users with six basic categories for modelling including classification, regression, clustering, dimensionality reduction, model reduction, and preprocessing.

Within each of these categories, there are models and algorithms represented as estimators, predictors, transformers, as well as their composite subclasses that are exposed to the user. In addition to the concrete classes derived from these three main interfaces, scikit-learn also provides a number of helper functions that assists in the simplifying of workflows by allowing for the construction of pipes and datastreams.

External Dependencies

Scikit-learn contains an “externals” package which contains bundled external dependencies such as numpy, scipy, and joblib. Dependencies within these packages are updated periodically and imported by numerous other packages within scikit-learn. For example, the calibration package (shown below) relies on numpy for an implementation of array functionality, scipy for various math functions, and joblib to run tasks in parallel. These dependencies are echoed throughout other packages in the library.



Package Structure

Scikit-learn provides a number of packages that encapsulate the algorithms, tools, helpers, and warnings that facilitate the six base modelling categories. According to the sci-kit learn documentation module, the roles of some of the most important packages are as follows:

Package Name	Key Roles and Features
`utils`	Contains tools, tests, and helper functions
`tree`	Contains algorithms and models such as decision/extreme trees
`semi_supervised`	Semi-supervised learning algorithm package
`preprocessing`	Scikit-learn's main data preprocessing package
`neutral_network`	Contains key neural network models
`neighbors`	Contains K-nearest neighbours algorithms
`model_selection`	Scikit-learn's focus training package
`metrics`	Contains measurement tools such as prediction accuracy scores
`linear_model`	Contains linear models including linear regression, logical regression etc
`inspection`	Provides tools used to check and verify models
`gaussian_process`	Uses the Gaussian process to perform classification and regression
`feature_selection`	Contains feature selection algorithms, such as single variable filtering and recursive feature deletion which can be used for dimension reduction
`externals`	Contains bundled external dependencies
`covariance`	Calculates covariance between features
`compose`	Contains meta learners used to synthesise models
`cluster`	Contains clustering algorithm implementations

Estimators, Predictors and Transformers

Estimators

Estimators are provided in scikit-learn as universal means of initialising both learning and data-processing algorithms. Both machine learning tasks as well as supervised and unsupervised learning algorithms used for tasks such as feature extraction/selection, classification, and clustering are provided to users as objects that implement the Estimator interface. As such, estimators form the core of the library and provide both object installation mechanisms as well as exposing a fit method which users can invoke to train learning models using data which is supplied as training data X and corresponding labels y.

A key design feature of the Estimator interface is that the initialization of its derived subclasses is decoupled from any given dataset required to facilitate the learning process. When an estimator is constructed, the training data is not directly passed into the constructor along with hyper-parameters, but rather through the inherited fit method. The fit method is a coupling function that maps provided data, in the form of a feature vector containing samples and a set of target labels, to a working model of that data. When invoked, it first runs a learning algorithm and uses this training data to set model-specific attributes onto an Estimator object implementation. Since fit() always returns the Estimator object it was invoked within, this estimator effectively becomes the model of its provided input and is able to make predictors and transformers of this data.

Predictors

Predictors are extensions of estimators, with the added requirement that once they are fitted to a training dataset (X, y), they provide the user with a method predict() which takes in a new dataset X′ and guesses the corresponding y′. The output may simply be the calculated values for each input, or include extra information such as the confidence of each prediction. Finally, as different predictors may perform differently on a given dataset, predictors contain a score() function which predicts the outputs for a test dataset and compares the predicted values with a set of given results, returning an approximation on how accurate the predictions might be.

Transformer

Similarly to estimators, transformers are decoupled objects that require fitting. Once data has been fit and an estimator has been returned, a transformer's exposed transform method can be invoked on some input dataset X to produce a modified dataset X′ with desirable properties (such as fewer dimensions or normalised data). In scikit-learn, the implementation of a variety of learning algorithms such as PCA and manifold learning; as well as preprocessing, dimensionality reduction, and feature extraction implement the transformer interface.

Aggregate Types

Scikit-learn also includes a number of additional components such as meta-estimators, pipelines, and model selectors.

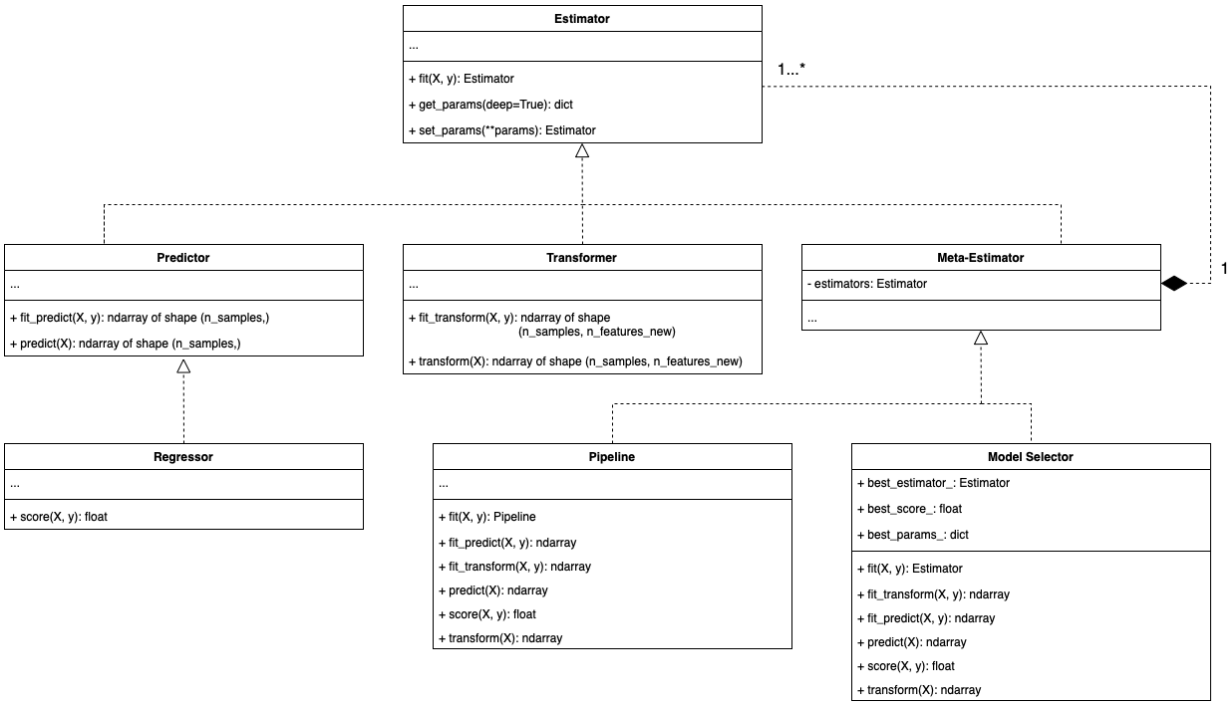
Meta-estimators are essentially compound estimators which combine a number of independent estimator and predictor objects into a single multi-classification estimator. This compound data structure allows for the reuse of predictors in an ensemble method.

Similarly, pipelines form a chain of transformers with their final predictors into a compound estimator which exposes a single method that combines the fit, transform, and predict methods of the encapsulated transformers and predictors into a single method which can more accurately generate a prediction.

Model selectors are a subclass of meta-estimators which repeatedly train their encapsulated estimators with varying values given the same hyper-parameters when the fit method is invoked. When this process is complete, the model with the best score will be returned.

Structural UML Diagram

The UML diagram below provides an approximation of the relationship between the scikit-learn Estimator, Predictor, and Transformer interfaces, as well as the aggregate types.



Scikit-learn Design Patterns

Factory Design Pattern

Design Pattern Overview

The factory method design pattern is a creational pattern which provides a way to instantiate objects without revealing specific creation details to the client. When a client requires an object, instead of instantiating the object directly, a factory method is invoked and provided with specifications about the required object. Based on this information, the factory method then determines the type of the object to instantiate and return to the client. Encouraging the client to interact with the factory method opposed to concrete implementations reduces coupling.

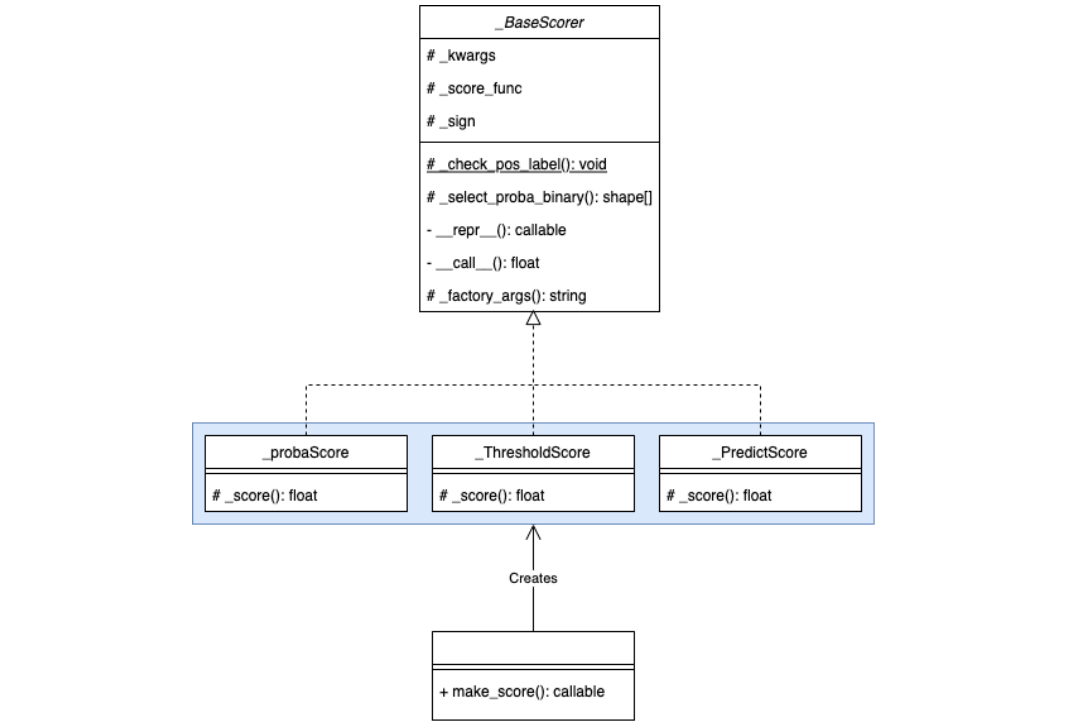
Factory in scikit-learn

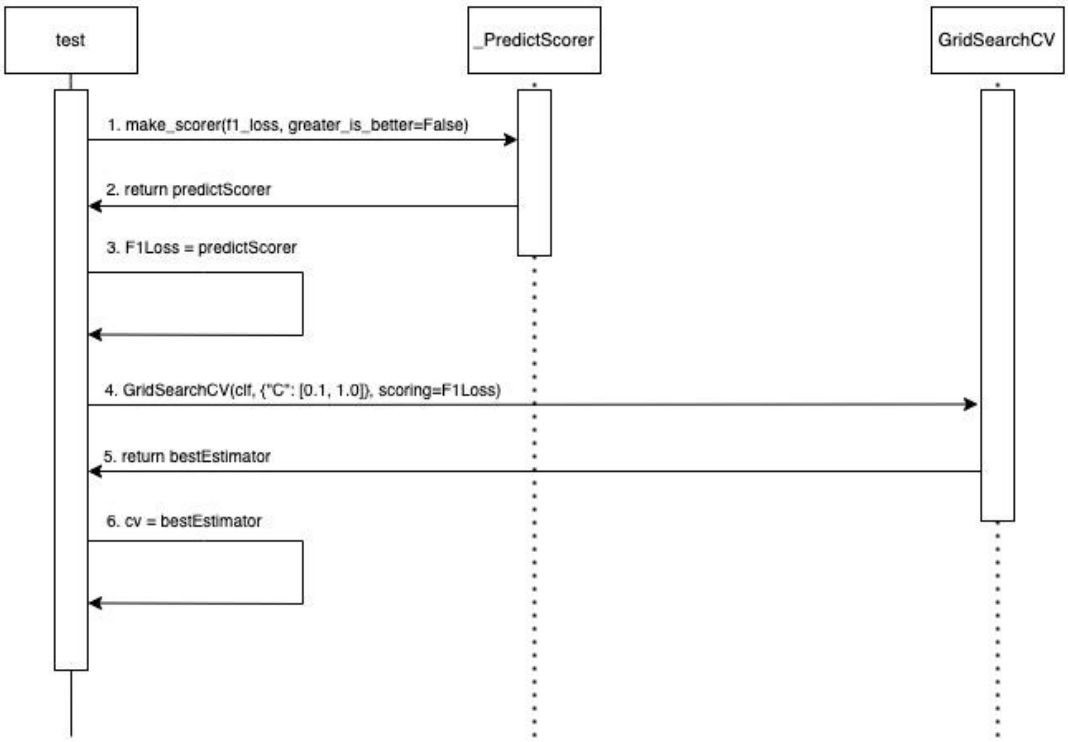
One such example of the factory method design pattern can be seen in _scorer.py. The function make_scorer() in sklearn/metrics/_scorer.py is a factory method; the function uses its parameters to determine which _BaseScorer object to instantiate and return. The following code snippet from make_scorer.py illustrates this behaviour:

```
664         sign = 1 if greater_is_better else -1
665         if needs_proba and needs_threshold:
666             raise ValueError(
667                 "Set either needs_proba or needs_threshold to True, but not both."
668             )
669         if needs_proba:
670             cls = _ProbaScorer
671         elif needs_threshold:
672             cls = _ThresholdScorer
673         else:
674             cls = _PredictScorer
675         return cls(score_func, sign, kwargs)
```

Both needs_proba and needs_threshold are optional parameters the client can specify, otherwise their values are set to false by default. As shown in lines 669 to 674 the values of needs_proba and needs_threshold dictate which object is created. The _ProbaScorer, _ThresholdScorer and _PredictScorer classes all extend the common abstract class _BaseScorer. Thus, make_scorer() returns an abstraction of _BaseScorer to the client.

The following class diagram showcases the relationship between the factory method, _BaseScorer, and its subclasses; and the sequence diagram illustrates key behaviour between GridSearchCV, _BaseScorer's subclass _PredictScorer, and test:





Iterator Design Pattern

Design Pattern Overview

The iterator pattern is a behavioural design pattern that “provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation." (from Design Patterns: Elements of Reusable Object-Oriented Software).

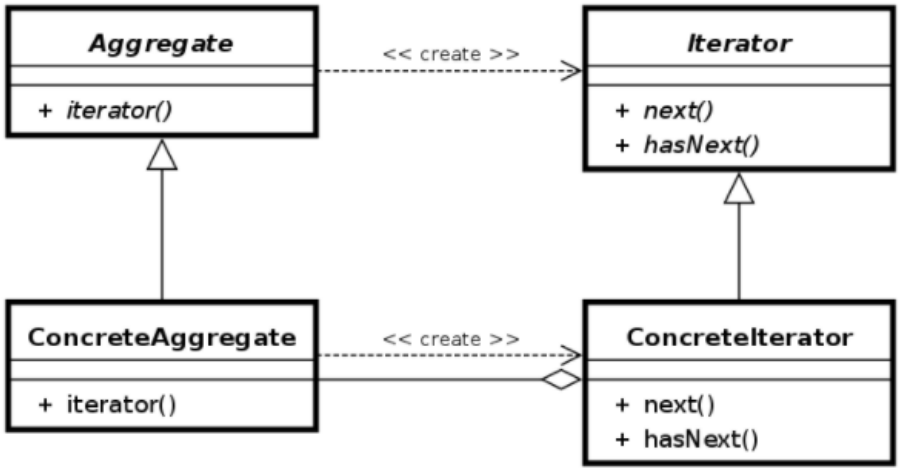
Iterator in scikit-learn

Examples of the Iterator design pattern in scikit-learn are classes that contain the __iter__() method and therefore are Python Iterables. One such case is the class ParameterGrid in sklearn/model_selection/_search.py.

```
133     def __iter__(self):
134         """Iterate over the points in the grid.
135 
136         Returns
137         -------
138         params : iterator over dict of str to any
139             Yields dictionaries mapping each estimator parameter to one of its
140             allowed values.
141         """
142         for p in self.param_grid:
143             # Always sort the keys of a dictionary, for reproducibility
144             items = sorted(p.items())
145             if not items:
146                 yield {}
147             else:
148                 keys, values = zip(*items)
149                 for v in product(*values):
150                     params = dict(zip(keys, v))
151                     yield params
```

The method is a generator function that returns a generator, a Python feature that simplifies the creation of Iterators. Calling __iter__() returns a generator which can be treated like any other Python Iterator, as it has its own __iter__() that returns itself, and __next__() that returns the next element based on the generator function.

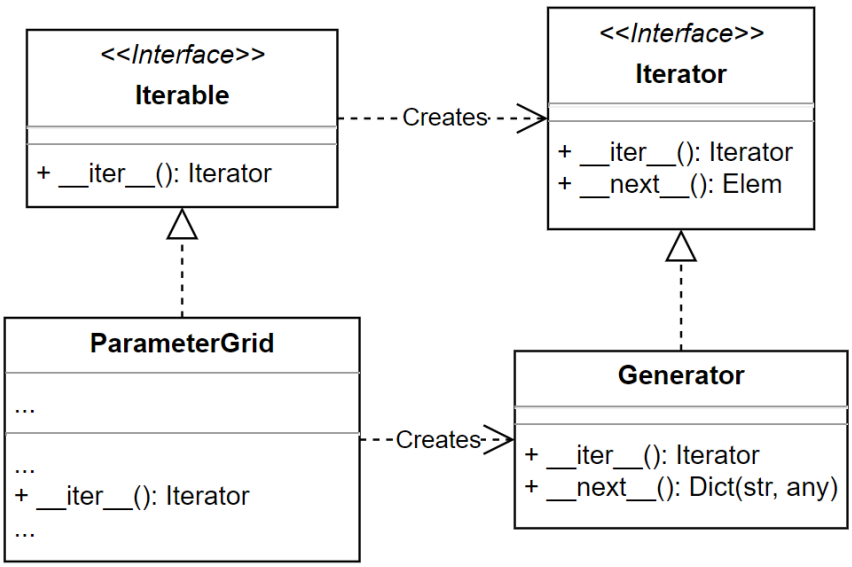
On the first __next__() call to the generator returned by ParameterGrid’s __iter__(), the code in the method is run from the start until the first yield is reached which returns the yielded value. Then on subsequent calls to __next__(), the code continues until a yield is reached which returns a value, or the end of the generator function has been reached, which returns a StopIteration exception.



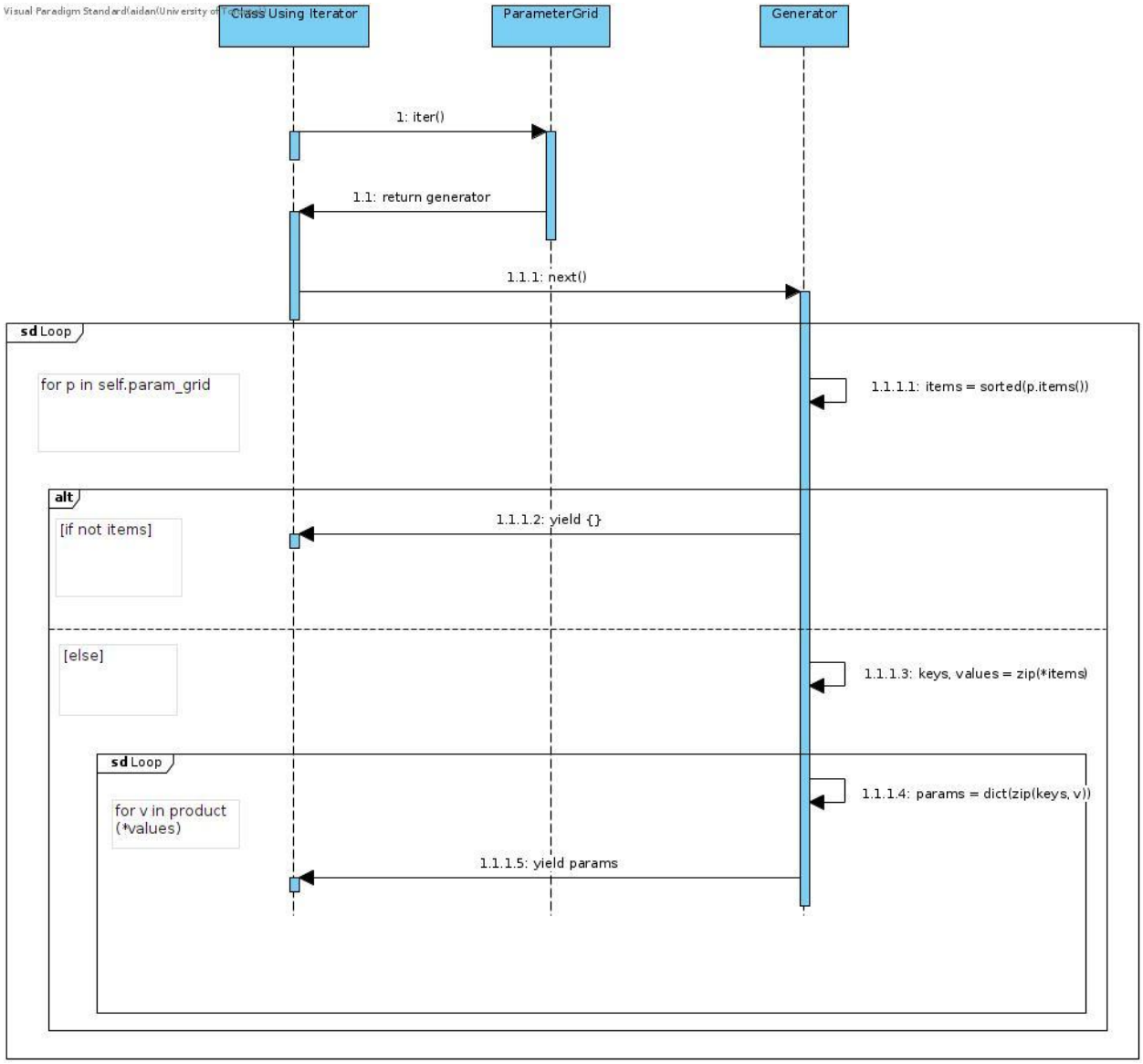
General UML for the Iterator design pattern
From: https://en.wikipedia.org/wiki/Iterator_pattern

The abstract Aggregate and Iterator correspond to Python's built-in Iterable and Iterator protocols which respectively require __iter__(), and __iter__() plus __next__(). Python Iterators do not implement hasNext() natively, but the StopIteration exception returned at the end of iterating or the optional parameter for next() can be used to check if the iterator is finished, similar to typical use cases of hasNext(). Note that Iterable and Iterator are examples of protocols in Python, which are similar to interfaces in other languages.

The ConcreteAggregate and ConcreteIterator correspond to ParameterGrid and the generator returned by __iter__() method. In this case, the generator and ParameterGrid do not have an aggregation relationship since the generator returns dictionaries instead of instances of ParameterGrid.



Sequence Diagram of Iterator design pattern for ParameterGrid in scikit-learn:



Strategy Design Pattern

Design Pattern Overview

The Strategy design pattern is a behavioural pattern that allows a program to select one algorithm out of a family of interchangeable algorithms to be executed at runtime. This is done using the Open-Closed Principle by capturing any abstraction in interfaces, and then implementing different strategic details within their deriving classes. This hierarchy allows a family of algorithms to be used interchangeably to adjust application behaviour without needing to change its architecture. At the same time, by encapsulating each derived algorithm within its own class, new algorithms can easily be created and added to a family through compliance with their corresponding interface(s).

Advantages

Since any data structures necessary for implementing each algorithm are completely encapsulated within their respective algorithm classes, and the algorithm family interface provides all deriving classes with common attributes and methods, any application making use of the Strategy design pattern inherently
allows its users to switch strategies/algorithms at run time by choosing the most appropriate algorithm for their needs without necessitating the use of case statements through the principle of polymorphic substitution.

Disadvantages

There are two inherent complexities and/or disadvantages to the Strategy Design Pattern however, which are that the application must maintain multiple instances of interchangeable strategy objects in the place of one; and that the strategy family base classes are required to expose interfaces for all required behaviour for their deriving classes, some of which may not necessarily be applicable to the implementation.

Strategy in scikit-learn

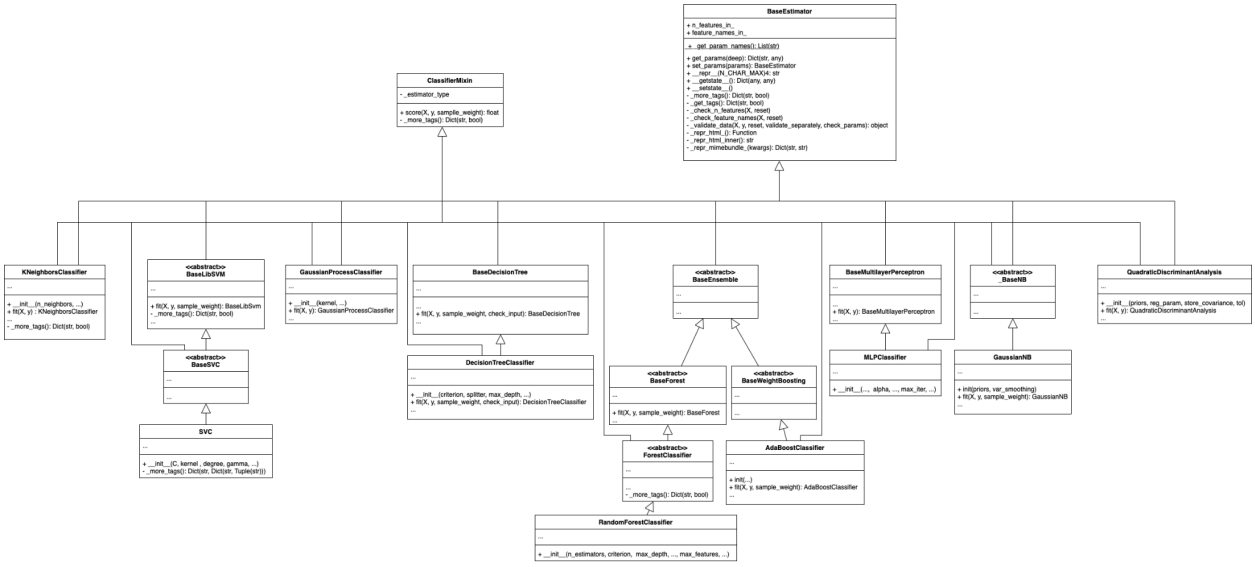
Within the scikit-learn code base, we see an example of the Strategy design pattern in use within plot_classifier_comparison.py, where various instances of interchangeable Classifier objects are instantiated, stored within a list, then each executed in sequence to compute results on the same dataset using common methods within a loop. These results are then plotted, providing a behavioural comparison of each of these algorithms. The embedded screenshots below illustrate this behaviour:

```
46  names = [
47      "Nearest Neighbors",
48      "Linear SVM",
49      "RBF SVM",
50      "Gaussian Process",
51      "Decision Tree",
52      "Random Forest",
53      "Neural Net",
54      "AdaBoost",
55      "Naive Bayes",
56      "QDA",
57  ]
58
59  classifiers = [
60      KNeighborsClassifier(3),
61      SVC(kernel="linear", C=0.025),
62      SVC(gamma=2, C=1),
63      GaussianProcessClassifier(1.0 * RBF(1.0)),
64      DecisionTreeClassifier(max_depth=5),
65      RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
66      MLPClassifier(alpha=1, max_iter=1000),
67      AdaBoostClassifier(),
68      GaussianNB(),
69      QuadraticDiscriminantAnalysis(),
70  ]
```

After the individual but interchangeable Classifiers are configured on lines 45 through 70, each of these classes' fit() methods are invoked between lines 119 and 122 using the same training dataset, and again scored using the same testing data by calling their score() method.

```
118     # iterate over classifiers
119     for name, clf in zip(names, classifiers):
120         ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
121         clf.fit(X_train, y_train)
122         score = clf.score(X_test, y_test)
```

In order to better understand why the fit() and score() methods are able to be invoked in the same manner for each of these Classifier object implementations, we have provided the structural diagram below. Due to the size of the diagram, it has also been provided separately as a PNG image within our A1 deliverables folder for additional readability:



Since all Classifiers implement the Estimator interface through duck typing by extending BaseEstimator, the base class for all Estimator objects in scikit-learn, each Classifier object is required to have a fit() method which accepts at least the X and y training datasets, allowing them to be invoked the same way, as seen on line 121.

Similarly, since Classifiers are, by definition, estimators with additional Classifier functionality mixed in, each Classifier additionally extends ClassifierMixin which provides similarly interchangeable methods including score() and _more_tags(). As such, each deriving Classifier's score() method is again able to be invoked the same way on line 122.

Thus, as a result of each Classifier object including both BaseEstimator and ClassifierMixin as superclasses in their inheritance tree, the functionality that they exhibit allows for multiple instances of various Classifier types to be instantiated and used interchangeably by leveraging the Strategy behavioural design pattern and its application of the Open-Closed Principle.