apricot

Release 0.6.0

Sep 30, 2020

apricot

1 Sparse Matrices 5

2 Knapsack Constraints 7

3 Streaming Optimization 9

4 Feature-based Functions 11

5 Maximum Coverage 15

6 Facility Location 19

7 Graph Cut 23

8 Sum Redundancy 27

9 Saturated Coverage 31

10 Mixtures 37

11 Naive Greedy 41

12 Lazy Greedy 43

13 Two-Stage Greedy 45

14 Approximate Lazy Greedy 47

15 Stochastic Greedy 49

16 Sample Greedy 51

17 GreeDi 53

18 Modular Greedy 55

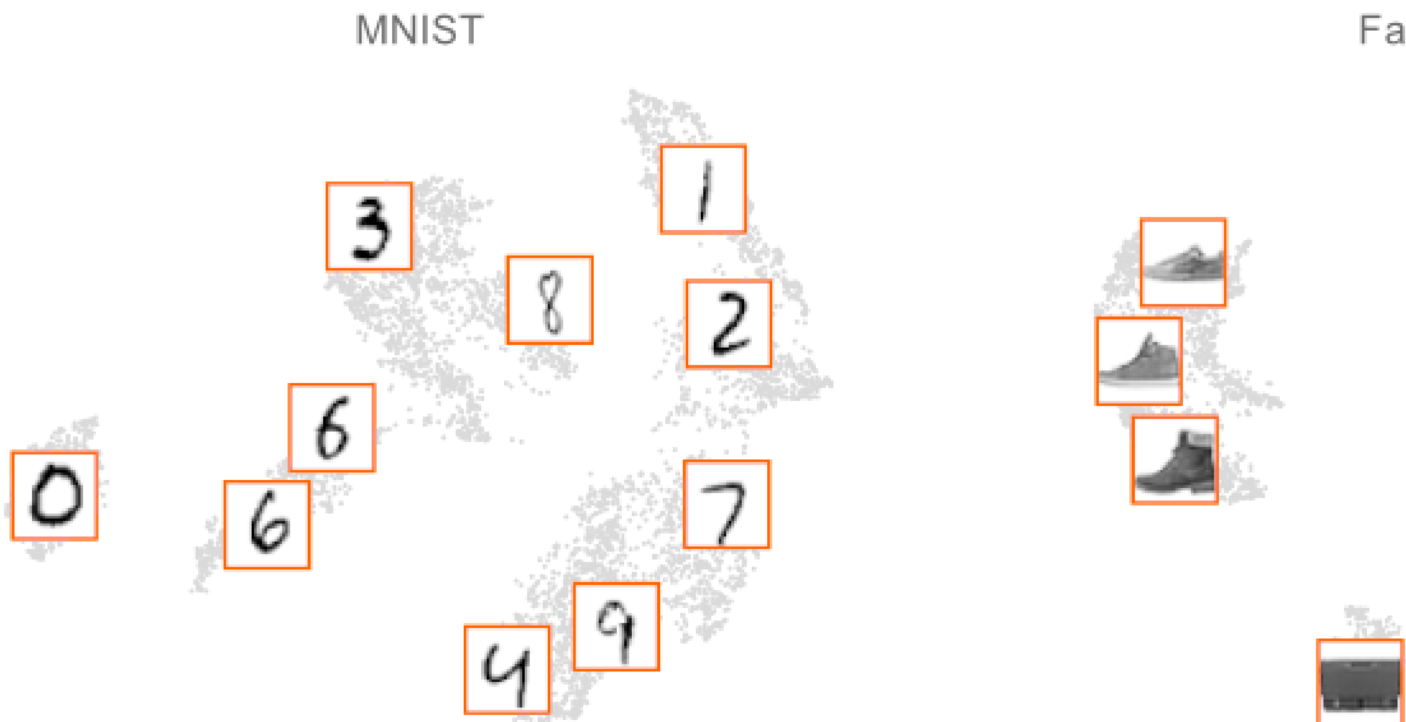
19 Bidirectional Greedy 57

Python Module Index 59

Index 61

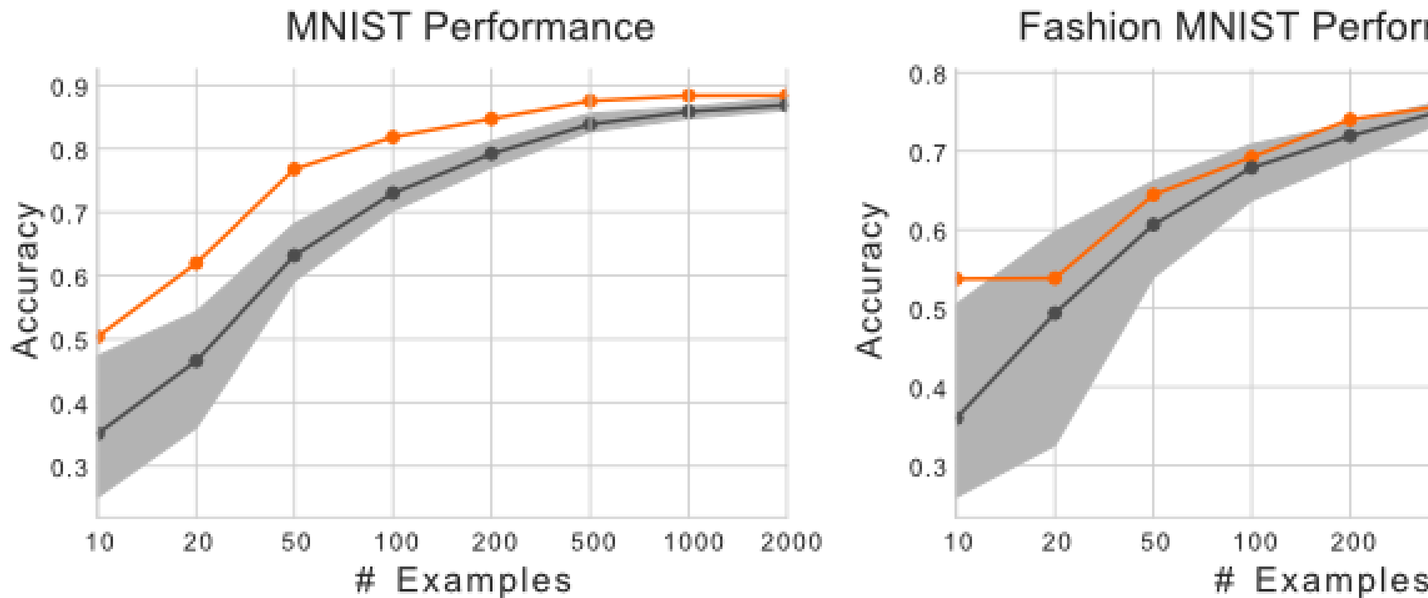
apricot is a package for submodular optimization brought to you in a convenient sklearn-like format. The package focuses on (1) being flexible enough for allow users to easily define their own functions and optimizers just as you can in neural network libraries, and (2) being fast by using numba to speed up the computationally intensive aspects.

The primary purpose of apricot is to summarize massive data sets into useful subsets. A simple way to use these subsets is to visualize the modalities of data. When apricot is used to summarize the MNIST and Fashion-MNIST data sets into ten elements, those selected represent the space well and cover almost all the classes without even having access to the labels.



These subsets can also be useful when training machine learning models. It might seem counterintuitive at first to train a model using only a fraction of your data. Unfortunately, the compute required to train models on huge data sets might not be available to everyone. Instead of relying on random subsampling, one could instead select a subset using

submodular optimization. Using the same data sets as before, we can see that training a logistic regression model using the subsets selected by apricot results in a model with higher accuracy than randomly selected subsets.



At a high level, submodular functions operate on sets of elements and optimizing them involves selecting a subset of elements. The functions implemented in apricot return a value that is inversely related to the redundancy of the selected elements, meaning that maximizing them involves selecting a set of non-redundant elements. A key property of these functions is that they are *submodular*, meaning that the gain in the function value that a particular element would yield either decreases or stays the same each time an element is added to the subset (this is also known as the diminishing returns property).

You can install apricot with `pip install apricot-select`.

The field of submodular optimization involves functions of the form $f : 2^V \rightarrow \mathcal{R}$. These functions take in a set of elements and output a real value that measures the quality of that set. Set functions are *submodular* when the returned value has the diminishing returns property such that for each $X \subseteq Y$

$$f(X + v) - f(X) \geq f(Y + v) - f(Y)$$

The diminishing returns property means that the gain of adding in a particular element v decreases or stays the same each time another element is added to the subset.

Note: Set functions are called *modular* when the returned value ignores the size of the subset and *supermodular* when the returned value increases with subset size.

An intuitive example of the diminishing returns property arises in the context of text data. In this situation, each example would be a blob of text, e.g. a sentence, and the set function being employed calculates the number of unique words across all selected sentences. Consider a particular example v, an already selected set of examples X, and a calculated gain g_v of adding v to X. If some other example, u is added to X instead of v, the gain of adding v (or any other example) to $X \cup u$ must be smaller or equal to the gain of adding v to X: either u has new words that would also

be new from v, in which case the benefit of adding in v would decrease, or it doesn't, in which case the gain would stay the same.

Here is an example of optimizing a maximum coverage function on text data, as described.

```
from apricot import MaxCoverageSelection
from keras.datasets import reuters

(X_, _), (_, _) = reuters.load_data(num_words=5000)
X = numpy.zeros((X_.shape[0], max(map(max, X_))+1))
for i, x in enumerate(X_):
    X[i][x] = 1

model = MaxCoverageSelection(250, optimizer='naive')
model.fit(X)
```

It is NP-hard to optimize submodular functions exactly and so a greedy algorithm is usually employed. This algorithm works by selecting examples one at a time based on the gain they would yield if added. In practice, the greedy algorithm finds solutions that are very close to the optimal solution, and Nemhauser (1978) showed that the quality of the subset cannot be worse than $1 - e^{-1}$ of the optimal value when selected using a greedy algorithm. More recently, many algorithms have been proposed that allow optimization to scale to massive data sets or be distributed across machines but only find approximations of the greedy solution.

CHAPTER 1

Sparse Matrices

apricot has built-in support for sparse matrices. This support makes it possible to summarize massive data sets whose dense representations are not feasible to work with, e.g. sparse similarity graphs when using graph-based functions or word counts for text data and feature-based functions. Using a sparse matrix will always (except when ties exist) give the same result that using a dense matrix populated with 0s would, but operating on sparse matrices can be significantly faster process.

Here is an example of using a feature-based function on a very sparse feature matrix. Note that each row can be thought of as an example with binary feature values.

```
from apricot import FeatureBasedSelection
from scipy.sparse import csr_matrix

X = numpy.random.randint(2, size=(10000, 100), p=[0.99, 0.01])
X_sparse = csr_matrix(X)

selector = FeatureBasedSelection(100, 'sqrt')
selector.fit(X_sparse)
```

Here is an example of using a graph-based function on a very sparse similarity matrix. Note that X is a pre-computed similarity matrix and must be specified as such to the function.

```
from apricot import FacilityLocationSelection
from scipy.sparse import csr_matrix

X = numpy.random.randint(10, size=(1000, 1000), p=[0.99, 0.01])
X = numpy.abs((X + X.T) / 2)
X_sparse = csr_matrix(X)

selector = FacilityLocationSelection(100, 'precomputed')
selector.fit(X_sparse)
```

CHAPTER 2

Knapsack Constraints

A knapsack constraint is an alternate to a cardinality constraint (the default in apricot) where each element has a cost and the selected items cannot exceed a total budget. The name (according to Krause (2012)) is a reference to the knapsack problem where one maximizes a modular function subject to a modular cost. In apricot, one is maximizing a *submodular* function subject to a modular cost. There are several ideas that costs might represent but the most intuitive is the simple monetary cost associated with each element. Consider the task of placing sensors around an area given a fixed budget to purchase sensors, where smaller sensors are cheaper than larger sensors. Submodular optimization subject to knapsack constraints is an intuitive way to determine what type of sensor should be placed and where without exceeding a given budget.

Stated more formally, the optimization problem given knapsack constraints is

$$\max_S f(S) \;\; s.t. \sum_{v \in S} c(v) \leq B$$

for a cost function c, a submodular function f, a subset S, and a budget B. Further, rather than optimizing the gain directly, the gain is divided by the cost of the element, resulting in the "cost-benefit" gain. Essentially, elements with a high gain and a high cost are prioritized lower than those with the same gain but a lower cost. This means that the optimization process is just a

```
from apricot import FeatureBasedSelection

X = numpy.random.randint(2, size=(10000, 100), p=[0.99, 0.01])
sample_cost = numpy.abs(numpy.random.randn(10000))

selector = FeatureBasedSelection(100, 'sqrt')
selector.fit(X, sample_cost=sample_cost)
```

CHAPTER 3

Streaming Optimization

Streaming optimization involves optimizing a submodular function on a stream of data. The key assumptions of these methods are that the data cannot all fit in memory at one time and, more restrictively, that each example can only be seen once: after seeing each example, the algorithm must decide whether to keep it or not before moving on to the next example.

Although this problem is difficult, principled approaches for solving it have been proposed that have theoretical guarantees. One such example, sieve streaming, involves making many estimates of what the objective score of the optimal subset would be, and selecting elements based on these elements in parallel. When estimates are found to be too low, the corresponding subsets are discarded, and when certain estimates that were initially too high are found to be plausible, the corresponding subset begins to be populated.

In apricot, streaming optimization can be used with any of the built-in functions by using the *partial_fit* method instead of the *fit* method. Although the algorithm is designed to be applied to one example at a time in a streaming setting, in practice applying the algorithm to batches of data can be much faster while still providing the same answer.

Note: Streaming optimization is implemented for mixtures of functions, but not for sum redundancy or saturated coverage functions.

You can read more about streaming optimization in this tutorial.

```
from apricot import FeatureBasedSelection

X = numpy.random.randint(2, size=(10000, 100), p=[0.99, 0.01])
sample_cost = numpy.exp(numpy.random.randn(10000))

selector = FeatureBasedSelection(100, 'sqrt')
selector.partial_fit(X, sample_cost=sample_cost)
```

CHAPTER 4

Feature-based Functions

class apricot.functions.featureBased.**FeatureBasedSelection**(*n_samples*, *concave_func='sqrt'*, *initial_subset=None*, *optimizer='two-stage'*, *optimizer_kwds={}*, *reservoir=None*, *max_reservoir_size=1000*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selector based off a feature based submodular function.

Feature-based functions are those that operate on the feature values of examples directly, rather than on a similarity matrix (or graph) derived from those features, as graph-based functions do. Because these functions do not require calculating and storing a $\mathcal{O}(n^2)$ sized matrix of similarities, they can easily scale to data sets with millions of examples.

Note: All values in your data must be positive for this selection to work.

The general form of a feature-based function is:

$$f(X) = \sum_{d=1}^{D} \phi\left(\sum_{i=1}^{N} X_{i,d}\right)$$

where f indicates the function that uses the concave function ϕ and is operating on a subset X that has N examples and D dimensions. Importantly, X is the subset and not the ground set, meaning that the time it takes to evaluate this function is proportional only to the size of the selected subset and not the size of the full data set, like it is for graph-based functions.

See https://ieeexplore.ieee.org/document/6854213 for more details on feature based functions.

Parameters

- **n_samples** (*int*) – The number of samples to return.
- **concave_func** (*str or callable*) – The type of concave function to apply to the feature values. You can pass in your own function to apply. Otherwise must be one of the following:

  'log' : log(1 + X) 'sqrt' : sqrt(X) 'sigmoid' : X / (1 + X)
- **initial_subset** (*list, numpy.ndarray or None*) – If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional.
- **optimizer** (*string or optimizers.BaseOptimizer, optional*) – The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

  'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

  Default is 'two-stage'.
- **optimizer_kwds** (*dict or None*) – A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.
- **reservoir** (*numpy.ndarray or None*) – The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.
- **max_reservoir_size** (*int*) – The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.
- **n_jobs** (*int*) – The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.
- **random_state** (*int or RandomState or None, optional*) – The random seed to use for the random selection process. Only used for stochastic greedy.
- **verbose** (*bool*) – Whether to print output during the selection process.

n_samples

The number of samples to select.

Type int

ranking

The selected samples in the order of their gain with the first number in the ranking corresponding to the index of the first sample that was selected by the greedy procedure.

Type numpy.array int

gains
The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

fit (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *FeatureBasedSelection*

fit_transform (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 5

Maximum Coverage

A maximum coverage function.

class apricot.functions.maxCoverage.**MaxCoverageSelection**(*n_samples*, *threshold=1.0*, *initial_subset=None*, *optimizer='two-stage'*, *optimizer_kwds={}*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selector based off a coverage function.

Maximum coverage functions aim to maximize the number of features that have a non-zero element in at least one selected example—there is no marginal benefit to observing a variable in two examples. If each variable is thought to be an item in a set, and the data is a binary matrix where a 1 indicates the item is present in the example and 0 indicates it is not, optimizing a maximum coverage function is a solution to the set coverage problem. These functions are useful when the space of variables is massive and each example only sees a small subset of them, which is a common situation when analyzing text data when the variables are words. The maximum coverage function is an instance of a feature-based function when the concave function is minimum.

Note: All values in your data must be binary for this selection to work.

The general form of a coverage function is:

$$f(X) = \sum_{d=1}^{D} \min \left(\sum_{n=1}^{N} X_{i,d}, 1 \right)$$

where f indicates the function that operates on a subset X that has N examples and D dimensions. Importantly, X is the subset and not the ground set, meaning that the time it takes to evaluate this function is proportional only to the size of the selected subset and not the size of the full data set, like it is for graph-based functions.

See https://www2.cs.duke.edu/courses/fall17/compsci632/scribing/scribe2.pdf where the problem is described as maximum coverage.

Parameters

- **n_samples** (*int*) – The number of examples to return.
- **initial_subset** (*list, numpy.ndarray or None*) – If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional.
- **optimizer** (*string or optimizers.BaseOptimizer, optional*) – The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

  'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

  Default is 'two-stage'.
- **optimizer_kwds** (*dict, optional*) – Arguments to pass into the optimizer object upon initialization. Default is {}.
- **n_jobs** (*int, optional*) – The number of cores to use for processing. This value is multiplied by 2 when used to set the number of threads. If set to -1, use all cores and threads. Default is -1.
- **random_state** (*int or RandomState or None, optional*) – The random seed to use for the random selection process. Only used for stochastic greedy.
- **verbose** (*bool*) – Whether to print output during the selection process.

n_samples

The number of samples to select.

Type int

ranking

The selected samples in the order of their gain with the first number in the ranking corresponding to the index of the first sample that was selected by the greedy procedure.

Type numpy.array int

gains

The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

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

Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *FeatureBasedSelection*

fit_transform(*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)

Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 6

Facility Location

class apricot.functions.facilityLocation.**FacilityLocationSelection**(*n_samples*, *metric='euclidean'*, *initial_subset=None*, *optimizer='lazy'*, *optimizer_kwds={}*, *n_neighbors=None*, *reservoir=None*, *max_reservoir_size=1000*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selector based off a facility location submodular function.

Facility location functions are general purpose submodular functions that, when maximized, choose examples that represent the space of the data well. The facility location function is based on maximizing the pairwise similarities between the points in the data set and their nearest chosen point. The similarity function can be species by the user but must be non-negative where a higher value indicates more similar.

Note: All ~pairwise~ values in your data must be non-negative for this selection to work.

In many ways, optimizing a facility location function is simply a greedy version of k-medoids, where after the first few examples are selected, the subsequent ones are at the center of clusters. The function, like most graph-based functions, operates on a pairwise similarity matrix, and successively chooses examples that are similar to examples whose current most-similar example is still very dissimilar. Phrased another way, successively chosen examples are representative of underrepresented examples.

The general form of a facility location function is

$$f(X,Y) = \sum_{yinY} \max_{xinX} \phi(x,y)$$

where f indicates the function, X is a subset, Y is the ground set, and ϕ is the similarity measure between two examples. Like most graph-based functons, the facility location function requires access to the full ground set.

This implementation allows users to pass in either their own symmetric square matrix of similarity values, or a data matrix as normal and a function that calculates these pairwise values.

For more details, see https://las.inf.ethz.ch/files/krause12survey.pdf page 4.

Parameters

- **n_samples** (*int*) – The number of samples to return.
- **metric** (*str, optional*) – The method for converting a data matrix into a square symmetric matrix of pairwise similarities. If a string, can be any of the metrics implemented in sklearn (see https://scikit-learn.org/stable/modules/ generated/sklearn.metrics.pairwise_distances.html), including "precomputed" if one has already generated a similarity matrix. Note that sklearn calculates distance matrices whereas apricot operates on similarity matrices, and so a distances.max() - distances transformation is performed on the resulting distances. For backcompatibility, 'corr' will be read as 'correlation'. Default is 'euclidean'.

initial_subset [list, numpy.ndarray or None, optional] If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional. Default is None.

optimizer [string or optimizers.BaseOptimizer, optional] The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

Default is 'two-stage'.

optimizer_kwds [dict or None] A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.

n_neighbors [int or None] When constructing a similarity matrix, the number of nearest neighbors whose similarity values will be kept. The result is a sparse similarity matrix which can significantly speed up computation at the cost of accuracy. Default is None.

reservoir [numpy.ndarray or None] The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.

max_reservoir_size [int] The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.

n_jobs [int] The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.

random_state [int or RandomState or None, optional] The random seed to use for the random selection process. Only used for stochastic greedy.

verbose [bool] Whether to print output during the selection process.

n_samples
The number of samples to select.

Type int

ranking
The selected samples in the order of their gain.

Type numpy.array int

gains
The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

fit (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *FacilityLocationSelection*

fit_transform (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 7

Graph Cut

class apricot.functions.graphCut.**GraphCutSelection**(*n_samples=10*, *metric='euclidean'*, *alpha=1*, *initial_subset=None*, *optimizer='naive'*, *optimizer_kwds={}*, *n_neighbors=None*, *reservoir=None*, *max_reservoir_size=1000*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selector based on using a graph-cut function.

Graph cuts are canonical class of functions that involves selecting examples that split the similarity matrix into subgraphs well.

Note: All ~pairwise~ values in your data must be non-negative for this selection to work.

The general form of a graph cut function is

$$f(X,V) = \lambda \sum_{v \in V} \sum_{x \in X} \phi(x,v) - \sum_{x,y \in X} \phi(x,y)$$

where f indicates the function, X is a subset, V is the ground set, and ϕ is the similarity measure between two examples. Like most graph-based functons, the graph-cut function requires access to the full similarity matrix.

This implementation allows users to pass in either their own symmetric square matrix of similarity values, or a data matrix as normal and a function that calculates these pairwise values.

Parameters

- **n_samples** (*int*) – The number of samples to return.
- **metric** (*str, optional*) – The method for converting a data matrix into a square symmetric matrix of pairwise similarities. If a string, can be any of

the metrics implemented in sklearn (see https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise_distances.html), including "precomputed" if one has already generated a similarity matrix. Note that sklearn calculates distance matrices whereas apricot operates on similarity matrices, and so a distances.max() - distances transformation is performed on the resulting distances. For backcompatibility, 'corr' will be read as 'correlation'. Default is 'euclidean'.

- **alpha** (*float*) – The weight of the first term in the graph-cut objective, which measures the how representative the selected examples are of the ground set. The larger this is, the more likely examples are chosen that are near points in the ground set, even if there are other already-selected examples that are similar. Default is 1.
- **initial_subset** (*list, numpy.ndarray or None, optional*) – If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional. Default is None.
- **optimizer** (*string or optimizers.BaseOptimizer, optional*) – The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

  'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

  Default is 'two-stage'.
- **optimizer_kwds** (*dict or None*) – A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.
- **n_neighbors** (*int or None*) – When constructing a similarity matrix, the number of nearest neighbors whose similarity values will be kept. The result is a sparse similarity matrix which can significantly speed up computation at the cost of accuracy. Default is None.
- **reservoir** (*numpy.ndarray or None*) – The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.
- **max_reservoir_size** (*int*) – The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.
- **n_jobs** (*int*) – The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.
- **random_state** (*int or RandomState or None, optional*) – The random seed to use for the random selection process. Only used for stochastic greedy.
- **verbose** (*bool*) – Whether to print output during the selection process.

n_samples
The number of samples to select.

Type int

pairwise_func
A function that takes in a data matrix and converts it to a square symmetric matrix.

Type callable

ranking
The selected samples in the order of their gain.

Type numpy.array int

gains
The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

fit (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *GraphCutSelection*

fit_transform (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 8

Sum Redundancy

```
class apricot.functions.sumRedundancy.SumRedundancySelection(n_samples=10, metric='euclidean', initial_subset=None, optimizer='two-stage', n_neighbors=None, reservoir=None, max_reservoir_size=1000, n_jobs=1, random_state=None, optimizer_kwds={}, verbose=False)
```

A selector based off a sum redundancy submodular function.

The sum redundancy function is a graph-based function that penalizes redundancy among the selected set. This approach is straightforward, in that it simply involved a sum. It is also fast in comparison to a facility location function because it involves only performing calculation over the selected set as opposed to the entire ground set. Because the sum of the similarities is not submodular, it is subtracted from the sum of the entire similarity matrix, such that examples that are highly similar to each other result in a lower value than examples that are not very similar.

Note: All ~pairwise~ values in your data must be positive for this selection to work.

The general form of a sum redundancy function is

$$f(X, V) = \sum_{x,y \in V} \phi(x, y) - \sum_{x,y \in X} \phi(x, y)$$

where f indicates the function, X is the selected subset, V is the ground set, and ϕ is the similarity measure between two examples. While sum redundancy functions involves calculating the sum of the entire similarity

matrix in principle, in practice if one is only calculating the gains this step can be ignored.

This implementation allows users to pass in either their own symmetric square matrix of similarity values, or a data matrix as normal and a function that calculates these pairwise values.

Parameters

- **n_samples** (*int*) – The number of samples to return.
- **metric** (*str, optional*) – The method for converting a data matrix into a square symmetric matrix of pairwise similarities. If a string, can be any of the metrics implemented in sklearn (see https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise_distances.html), including "precomputed" if one has already generated a similarity matrix. Note that sklearn calculates distance matrices whereas apricot operates on similarity matrices, and so a distances.max() - distances transformation is performed on the resulting distances. For backcompatibility, 'corr' will be read as 'correlation'. Default is 'euclidean'.

initial_subset [list, numpy.ndarray or None, optional] If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional. Default is None.

optimizer [string or optimizers.BaseOptimizer, optional] The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

> 'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

Default is 'two-stage'.

optimizer_kwds [dict or None] A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.

n_neighbors [int or None] When constructing a similarity matrix, the number of nearest neighbors whose similarity values will be kept. The result is a sparse similarity matrix which can significantly speed up computation at the cost of accuracy. Default is None.

reservoir [numpy.ndarray or None] The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.

max_reservoir_size [int] The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.

n_jobs [int] The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.

random_state [int or RandomState or None, optional] The random seed to use for the random selection process. Only used for stochastic greedy.

verbose [bool] Whether to print output during the selection process.

n_samples
The number of samples to select.

Type int

pairwise_func
A function that takes in a data matrix and converts it to a square symmetric matrix.

Type callable

ranking
The selected samples in the order of their gain.

Type numpy.array int

gains
The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

fit (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *SumRedundancySelection*

fit_transform (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 9

Saturated Coverage

class apricot.functions.saturatedCoverage.**SaturatedCoverageSelection**(*n_samples=10*, *metric='euclidean'*, *alpha=0.1*, *initial_subset=None*, *optimizer='two-stage'*, *n_neighbors=None*, *n_jobs=1*, *random_state=None*, *reservoir=None*, *max_reservoir_size=None*, *optimizer_kwds={}*, *verbose=False*)

A saturated coverage submodular selection algorithm.

The saturated coverage function is a graph-based function where the gain is the sum of the similarities between the candidate example and the entire ground set up until a certain point. Essentially, each example in the ground set contributes to the overall function until some prespecified level of "coverage" is met. Once an example is similar enough to the selected examples, it stops contributing to the function.

Note: All ~pairwise~ values in your data must be positive for this selection to work.

The general form of a saturated coverage function is

$$f(X, V) = \sum_{v \in V} \min\{\sum_{x \in X} \phi(x, v), \alpha\}$$

where f indicates the function, X is a subset, V is the ground set, and ϕ is the similarity measure between two examples, and α is a parameter that, for each point in the ground set, specifies the maximum similarity that each example can have to the selected set (the saturation). Like most graph-based functons, the saturated coverage function requires access to the full ground set.

Parameters

- **n_samples** (*int*) – The number of samples to return.
- **metric** (*str, optional*) – The method for converting a data matrix into a square symmetric matrix of pairwise similarities. If a string, can be any of the metrics implemented in sklearn (see https://scikit-learn.org/stable/modules/ generated/sklearn.metrics.pairwise_distances.html), including "precomputed" if one has already generated a similarity matrix. Note that sklearn calculates distance matrices whereas apricot operates on similarity matrices, and so a distances.max() - distances transformation is performed on the resulting distances. For backcompatibility, 'corr' will be read as 'correlation'. Default is 'euclidean'.
- **alpha** (*float*) – The threshold at which to saturate. Larger values of alpha are closer to a modular function and smaller values of alpha are closer to all examples providing the same, small, gain. Default is 0.1.
- **initial_subset** (*list, numpy.ndarray or None, optional*) – If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional. Default is None.
- **optimizer** (*string or optimizers.BaseOptimizer, optional*) – The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

  'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

  Default is 'two-stage'.
- **optimizer_kwds** (*dict or None*) – A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.
- **n_neighbors** (*int or None*) – When constructing a similarity matrix, the number of nearest neighbors whose similarity values will be kept. The result is a sparse similarity matrix which can significantly speed up computation at the cost of accuracy. Default is None.
- **reservoir** (*numpy.ndarray or None*) – The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.

- **max_reservoir_size** (*int*) – The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.
- **n_jobs** (*int*) – The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.
- **random_state** (*int or RandomState or None, optional*) – The random seed to use for the random selection process. Only used for stochastic greedy.
- **verbose** (*bool*) – Whether to print output during the selection process.

n_samples

The number of samples to select.

Type int

pairwise_func

A function that takes in a data matrix and converts it to a square symmetric matrix.

Type callable

ranking

The selected samples in the order of their gain.

Type numpy.array int

gains

The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

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

Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *SaturatedCoverageSelection*

fit_transform(*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)

Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.

- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 10

Mixtures

class apricot.functions.mixture.**MixtureSelection**(*n_samples*, *functions*, *weights=None*, *metric='ignore'*, *initial_subset=None*, *optimizer='two-stage'*, *optimizer_kwds={}*, *n_neighbors=None*, *reservoir=None*, *max_reservoir_size=1000*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selection approach based on a mixture of submodular functions.

A convenient property of submodular functions is that any linear combination of submodular functions is still submodular. More generally, the linear combination of any number of submodular functions (assuming non-negative weights) is still a submodular function. Because of this, a mixture of submodular functions can be optimized using the same algorithms as an individual submodular function. Mixtures can be useful in situations where there are different important aspects of data that are each submodular.

The general form of a mixture function is

$$f(X) = \sum_{i=1}^{M} \alpha_i g_i(X)$$

where f indicates the mixture function, M is the number of functions in the mixture, X is a subset, α_i is the weight of the i-th function and g_i is the i-th function.

Note: There must be at least two components to the mixture, and all α must be non-negative.

This class can also be used to add regularizers to the selection procedure. If a submodular function is mixed with another submodular function that acts as a regularizer, such as feature based selection mixed with a custom function measuring some property of the selected subset.

Parameters

- **n_samples** (*int*) – The number of samples to return.

- **functions** (*list*) – The list of submodular functions to mix together. The submodular functions should be instantiated.
- **weights** (*list, numpy.ndarray or None, optional*) – The relative weight of each submodular function. This is the value that the gain from each submodular function is multiplied by before being added together. The default is equal weight for each function.
- **initial_subset** (*list, numpy.ndarray or None, optional*) – If provided, this should be a list of indices into the data matrix to use as the initial subset, or a group of examples that may not be in the provided data should beused as the initial subset. If indices, the provided array should be one-dimensional. If a group of examples, the data should be 2 dimensional.
- **optimizer** (*string or optimizers.BaseOptimizer, optional*) – The optimization approach to use for the selection. Default is 'two-stage', which makes selections using the naive greedy algorithm initially and then switches to the lazy greedy algorithm. Must be one of

  'random' : randomly select elements (dummy optimizer) 'modular' : approximate the function using its modular upper bound 'naive' : the naive greedy algorithm 'lazy' : the lazy (or accelerated) greedy algorithm 'approximate-lazy' : the approximate lazy greedy algorithm 'two-stage' : starts with naive and switches to lazy 'stochastic' : the stochastic greedy algorithm 'sample' : randomly take a subset and perform selection on that 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

  Default is 'two-stage'.
- **optimizer_kwds** (*dict or None*) – A dictionary of arguments to pass into the optimizer object. The keys of this dictionary should be the names of the parameters in the optimizer and the values in the dictionary should be the values that these parameters take. Default is None.
- **reservoir** (*numpy.ndarray or None*) – The reservoir to use when calculating gains in the sieve greedy streaming optimization algorithm in the *partial_fit* method. Currently only used for graph-based functions. If a numpy array is passed in, it will be used as the reservoir. If None is passed in, will use reservoir sampling to collect a reservoir. Default is None.
- **max_reservoir_size** (*int*) – The maximum size that the reservoir can take. If a reservoir is passed in, this value is set to the size of that array. Default is 1000.
- **n_jobs** (*int*) – The number of threads to use when performing computation in parallel. Currently, this parameter is exposed but does not actually do anything. This will be fixed soon.
- **random_state** (*int or RandomState or None, optional*) – The random seed to use for the random selection process. Only used for stochastic greedy.
- **verbose** (*bool, optional*) – Whether to print output during the selection process.

pq

The priority queue used to implement the lazy greedy algorithm.

Type PriorityQueue

n_samples

The number of samples to select.

Type int

submodular_functions
A concave function for transforming feature values, often referred to as phi in the literature.

Type list

weights
The weights of each submodular function.

Type numpy.ndarray

ranking
The selected samples in the order of their gain with the first number in the ranking corresponding to the index of the first sample that was selected by the greedy procedure.

Type numpy.array int

gains
The gain of each sample in the returned set when it was added to the growing subset. The first number corresponds to the gain of the first added sample, the second corresponds to the gain of the second added sample, and so forth.

Type numpy.array float

fit (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run submodular optimization to select the examples.

This method is a wrapper for the full submodular optimization process. It takes in some data set (and optionally labels that are ignored during this process) and selects *n_samples* from it in the greedy manner specified by the optimizer.

This method will return the selector object itself, not the transformed data set. The *transform* method will then transform a data set to the selected points, or alternatively one can use the ranking stored in the *self.ranking* attribute. The *fit_transform* method will perform both optimization and selection and return the selected items.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and th corresponding labels for the rows that have been selected.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The weight of each example. Currently ignored in apricot but included to maintain compatibility with sklearn pipelines.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns **self** – The fit step returns this selector object.

Return type *MixtureSelection*

fit_transform (*X*, *y=None*, *sample_weight=None*, *sample_cost=None*)
Run optimization and select a subset of examples.

This method will first perform the *fit* step and then perform the *transform* step, returning a transformed data set.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.
- **sample_cost** (*list or numpy.ndarray or None, shape=(n,), optional*) – The cost of each item. If set, indicates that optimization should be performed with respect to a knapsack constraint.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

transform(*X*, *y=None*, *sample_weight=None*)

Transform a data set to include only the selected examples.

This method will return a selection of X and optionally selections of y and sample_weight. The default setting is to select items based on the ranking determined in the *fit* step with examples in the same order as that ranking. Optionally, the whole data set can be returned, with the weights corresponding to samples that were not selected set to 0. This setting can be controlled by setting *pipeline=True*.

Parameters

- **X** (*list or numpy.ndarray, shape=(n, d)*) – The data set to transform. Must be numeric.
- **y** (*list or numpy.ndarray or None, shape=(n,), optional*) – The labels to transform. If passed in this function will return both the data and the corresponding labels for the rows that have been selected. Default is None.
- **sample_weight** (*list or numpy.ndarray or None, shape=(n,), optional*) – The sample weights to transform. If passed in this function will return the selected labels (y) and the selected samples, even if no labels were passed in. Default is None.

Returns

- **X_subset** (*numpy.ndarray, shape=(n_samples, d)*) – A subset of the data such that n_samples < n and n_samples is the integer provided at initialization.
- **y_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The labels that match with the indices of the samples if y is passed in. Only returned if passed in.
- **sample_weight_subset** (*numpy.ndarray, shape=(n_samples,), optional*) – The weight of each example.

CHAPTER 11

Naive Greedy

The naive greedy algorithm for optimization.

The naive greedy algorithm is the simplest greedy approach for optimizing submodular functions. The approach simply iterates through each example in the ground set that has not already been selected and calculates the gain in function value that would result from adding that example to the selected set. This process is embarassingly parallel and so is extremely amenable both to parallel processing and distributed computing. Further, because it is conceptually simple, it is also simple to implement.

The naive greedy algorithm can be specified for any function by passing in *optimizer='naive'* to the relevant selector object. Here is an example of specifying the naive greedy algorithm for optimizing a feature-based function.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.function`

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.verbose`

Whether to display a progress bar during the optimization process.

Type bool

`self.gains_`

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 12

Lazy Greedy

The lazy/accelerated greedy algorithm for optimization.

The lazy (or accelerated) greedy algorithm is a fast alternate to the naive greedy algorithm that results in the same items being selected. This algorithm exploits the diminishing returns property of submodular functions in order to avoid re-evaluating examples that are known to provide little gain. If an example has a small gain relative to other examples, it is unlikely to be the next selected example because that gain can only go down as more items are selected. Thus, the example should only be re-evaluated once the gains of other examples have gotten smaller.

The key idea of the lazy greedy algorithm is to maintain a priority queue where the examples are the elements in the queue and the priorities are the gains the last time they were evaluated. The algorithm has two steps. The first step is to calculate the gain that each example would have if selected first (the modular upper bound) and populate the priority queue using these values. The second step is to recalculate the gain of the first example in the priority queue and then add the example back into the queue. If the example remains at the front of the queue it is selected because no other example could have a larger gain once re-evaluated (due to the diminishing returns property).

While the worst case time complexity of this algorithm is the same as the naive greedy algorithm, in practice it can be orders of magnitude faster. Empirically, it appears to accelerate graph-based functions much more than it does feature-based ones. Functions also seem to be more accelerated the more curved they are.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

`self.pq`
The priority queue used to order examples for evaluation.

Type utils.PriorityQueue

`self.gains_`
The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 13

Two-Stage Greedy

An approach that switches between two optimizers midway through.

The two-stage greedy optimizer is a general purpose framework for combining two optimizers by making the first n selections out of k total selections using one optimizer, and then making the remainder using the other. When the first optimizer is random selection and the second approach is naive/lazy greedy, this becomes partial enumeration. By default, the first algorithm is the naive greedy optimizer and the second algorithm is the lazy greedy. This combination results in the same selection as either optimizer individually but replaces the computationally intensive first few steps for the priority queue, where the algorithm may require scanning through almost the entire queue, with the parallelizable naive greedy algorithm. While, in theory, the lazy greedy algorithm will never perform more function calls than the naive greedy algorithm, there are costs associated both with maintaining a priority queue and with evaluating a single example instead of a batch of examples.

This optimizer, with the naive greedy optimizer first and the lazy greedy optimizer second, is the default optimizer for apricot selectors.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param self.n_first_selections The number of selections to perform using the naive greedy algorithm before populating the priority queue and using the lazy greedy algorithm.

type self.n_first_selections int

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

CHAPTER 14

Approximate Lazy Greedy

The approximate lazy/accelerated greedy algorithm for optimization.

The approximate lazy greedy algorithm is a simple extension of the lazy greedy algorithm that, rather than requiring that an element remains at the top of the priority queue after being re-evaluated, only requires that the gain is within a certain user-defined percentage of the best gain to be selected. The key point in this approach is that finding the very best element while maintaining the priority queue may be expensive, but finding elements that are good enough is simple. While the best percentage to use is data set specific, even values near 1 can lead to large savings in computation.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.function`
A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.verbose`
Whether to display a progress bar during the optimization process.

Type bool

`self.pq`
The priority queue used to order examples for evaluation.

Type utils.PriorityQueue

`self.gains_`
The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 15

Stochastic Greedy

The stochastic greedy algorithm for optimization.

The stochastic greedy algorithm is a simple approach that, for each iteration, randomly selects a subset of data and then finds the best next example within that subset. The distinction between this approach and the sample greedy algorithm is that this subset changes at each iteration, meaning that the algorithm does cover the entire data set. In contrast, the sample greedy algorithm is equivalent to manually subsampling the data before running a selector on it. The size of this subset is proportional to the number of examples that are chosen and determined in a manner that results in the same amount of computation being done no matter how many elements are selected. A key idea from this approach is that, while the exact ranking of the elements may differ from the naive/lazy greedy approaches, the set of selected elements is likely to be similar despite the limited amount of computation.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param epsilon The inverse of the sampling probability of any particular point being included in the subset, such that 1 - epsilon is the probability that a point is included. Default is 0.9.

type epsilon float, optional

param random_state The random seed to use for the random selection process.

type random_state int or RandomState or None, optional

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

`self.gains_`

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 16

Sample Greedy

The sample greedy algorithm for optimization.

The sample greedy algorithm is a simple approach that subsamples the full data set with a user-defined sampling probability and then runs an optimization on that subset. This subsampling can lead to obvious speed improvements because fewer elements as selected, but will generally find a lower quality subset because fewer elements are present. This approach is typically used a baseline for other approaches but can save a lot of time on massive data sets that are known to be highly redundant.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param epsilon The sampling probability to use when constructing the subset. A subset of size n * epsilon will be selected from.

type epsilon float, optional

param random_state The random seed to use for the random selection process.

type random_state int or RandomState or None, optional

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

`self.`**`gains_`**

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 17

GreeDi

An approach for optimizing submodular functions on massive data sets.

GreeDi is an optimizer that was designed to work on data sets that are too large to fit into memory. The approach involves first partitioning the data into m equally sized chunks without any overlap. Then, l elements are selected from each chunk using a standard optimizer like naive or lazy greedy. Finally, these ml examples are merged and a standard optimizer selects k examples from this set. In this manner, the algorithm sacrifices exactness to ensure that memory limitations are not an issue.

There are a few considerations to keep in mind when using GreeDi. Naturally, ml must both be larger than k and also small enough to fit into memory. The larger l, the closer the solution is to the exact solution but also the more compute time is required. Conversely, the larger m is, the less exact the solution is. When using a graph-based function, increasing m can dramatically reduce the amount of computation that needs to be performed, as well as the memory requirements, because the similarity matrix becomes smaller in size. However, feature-based functions are likely to see less of a speed improvement because the cost of evaluating an example is independent of the size of ground set.

param function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type function base.BaseSelection

param m The number of partitions to split the data into.

type m int

param l The number of exemplars to select from each partition. l*m must be larger than the number of exemplars k that will be selected later on.

type l int

param optimizer1 The optimizer to use in the first stage of the process where l exemplars are selected from each partition. Default is ‘lazy’.

type optimizer1 str or base.Optimizer, optional

param optimizer2 The optimizer to use in the second stage where k exemplars are selected from the l*m exemplars returned from the first stage. Default is ‘lazy’.

type optimizer2 str or base.Optimizer, optional

param random_state The random seed to use for the random selection process.

type random_state int or RandomState or None, optional

param verbose Whether to display a progress bar during the optimization process.

type verbose bool

self.**function**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized. If None, will be set by the selector when passed in.

Type base.BaseSelection or None

self.**m**

The number of partitions that the data will be split into.

Type int

self.**l**

The number of exemplars that will be selected from each partition.

Type int

self.**verbose**

Whether to display a progress bar during the optimization process.

Type bool

self.**gains_**

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 18

Modular Greedy

An approach that approximates a submodular function as modular.

This approach approximates the submodular function by using its modular upper-bounds to do the selection. Essentially, a defining characteristic of submodular functions is the *diminishing returns* property where the gain of an example decreases with the number of selected examples. In contrast, modular functions have constant gains for examples regardless of the number of selected examples. Thus, approximating the submodular function as a modular function can serve as an upper-bound to the gain for each example during the selection process. This approximation makes the function simple to optimize because one would simply calculate the gain that each example yields before any examples are selected, sort the examples by this gain, and select the top k examples. While this approach is fast, this approach is likely best paired with a traditional optimization algorithm after the first few examples are selected.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param epsilon The inverse of the sampling probability of any particular point being included in the subset, such that 1 - epsilon is the probability that a point is included. Default is 0.9.

type epsilon float, optional

param random_state The random seed to use for the random selection process.

type random_state int or RandomState or None, optional

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

`self.gains_`

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

CHAPTER 19

Bidirectional Greedy

The bidirectional greedy algorithm.

Most submodular optimizers assume that the function is *monotone*, i.e., that the gain from each successive example is positive. However, there are some cases where the key diminishing returns property holds, but the gains are not necessarily positive. The most obvious of these is a difference in submodular functions. In these cases, the naive greedy algorithm is not guaranteed to return a good result.

The bidirectional greedy algorithm was developed to optimize non-monotone submodular functions. While it has a guarantee that is lower than the naive greedy algorithm has for monotone functions, it generally returns better sets than the greedy algorithm.

param self.function A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

type self.function base.BaseSelection

param self.verbose Whether to display a progress bar during the optimization process.

type self.verbose bool

`self.`**`function`**

A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

Type base.BaseSelection

`self.`**`verbose`**

Whether to display a progress bar during the optimization process.

Type bool

`self.`**`gains_`**

The gain that each example would give the last time that it was evaluated.

Type numpy.ndarray or None

Python Module Index

a

apricot.functions.facilityLocation, 19
apricot.functions.featureBased, 11
apricot.functions.graphCut, 23
apricot.functions.maxCoverage, 15
apricot.functions.mixture, 37
apricot.functions.saturatedCoverage, 31
apricot.functions.sumRedundancy, 27
apricot.optimizers.ApproximateLazyGreedy, 47
apricot.optimizers.BidirectionalGreedy, 57
apricot.optimizers.GreeDi, 53
apricot.optimizers.LazyGreedy, 43
apricot.optimizers.ModularGreedy, 55
apricot.optimizers.NaiveGreedy, 41
apricot.optimizers.SampleGreedy, 51
apricot.optimizers.StochasticGreedy, 49
apricot.optimizers.TwoStageGreedy, 45

Index

A

apricot.functions.facilityLocation (*module*), 19
apricot.functions.featureBased (*module*), 11
apricot.functions.graphCut (*module*), 23
apricot.functions.maxCoverage (*module*), 15
apricot.functions.mixture (*module*), 37
apricot.functions.saturatedCoverage (*module*), 31
apricot.functions.sumRedundancy (*module*), 27
apricot.optimizers.ApproximateLazyGreedy (*module*), 47
apricot.optimizers.BidirectionalGreedy (*module*), 57
apricot.optimizers.GreeDi (*module*), 53
apricot.optimizers.LazyGreedy (*module*), 43
apricot.optimizers.ModularGreedy (*module*), 55
apricot.optimizers.NaiveGreedy (*module*), 41
apricot.optimizers.SampleGreedy (*module*), 51
apricot.optimizers.StochasticGreedy (*module*), 49
apricot.optimizers.TwoStageGreedy (*module*), 45

F

FacilityLocationSelection (*class in apricot.functions.facilityLocation*), 19
FeatureBasedSelection (*class in apricot.functions.featureBased*), 11
fit() (*apricot.functions.facilityLocation.FacilityLocationSelection method*), 21
fit() (*apricot.functions.featureBased.FeatureBasedSelection method*), 13
fit() (*apricot.functions.graphCut.GraphCutSelection method*), 25
fit() (*apricot.functions.maxCoverage.MaxCoverageSelection method*), 16
fit() (*apricot.functions.mixture.MixtureSelection method*), 39
fit() (*apricot.functions.saturatedCoverage.SaturatedCoverageSelection method*), 33
fit() (*apricot.functions.sumRedundancy.SumRedundancySelection method*), 29
fit_transform() (*apricot.functions.facilityLocation.FacilityLocationSelection method*), 21
fit_transform() (*apricot.functions.featureBased.FeatureBasedSelection method*), 13
fit_transform() (*apricot.functions.graphCut.GraphCutSelection method*), 25
fit_transform() (*apricot.functions.maxCoverage.MaxCoverageSelection method*), 17
fit_transform() (*apricot.functions.mixture.MixtureSelection method*), 39
fit_transform() (*apricot.functions.saturatedCoverage.SaturatedCoverageSelection method*), 34
fit_transform() (*apricot.functions.sumRedundancy.SumRedundancySelection method*), 29
function (*apricot.optimizers.ApproximateLazyGreedy.self attribute*), 47
function (*apricot.optimizers.BidirectionalGreedy.self attribute*), 57
function (*apricot.optimizers.GreeDi.self attribute*), 54
function (*apricot.optimizers.LazyGreedy.self attribute*), 43
function (*apricot.optimizers.ModularGreedy.self attribute*), 55
function (*apricot.optimizers.NaiveGreedy.self at-*

tribute), 41
function (*apricot.optimizers.SampleGreedy.self attribute*), 51
function (*apricot.optimizers.StochasticGreedy.self attribute*), 49
function (*apricot.optimizers.TwoStageGreedy.self attribute*), 45

G

gains (*apricot.functions.facilityLocation.FacilityLocationSelection attribute*), 21
gains (*apricot.functions.featureBased.FeatureBasedSelection attribute*), 12
gains (*apricot.functions.graphCut.GraphCutSelection attribute*), 25
gains (*apricot.functions.maxCoverage.MaxCoverageSelection attribute*), 16
gains (*apricot.functions.mixture.MixtureSelection attribute*), 39
gains (*apricot.functions.saturatedCoverage.SaturatedCoverageSelection attribute*), 33
gains (*apricot.functions.sumRedundancy.SumRedundancySelection attribute*), 29
gains_ (*apricot.optimizers.ApproximateLazyGreedy.self attribute*), 47
gains_ (*apricot.optimizers.BidirectionalGreedy.self attribute*), 57
gains_ (*apricot.optimizers.GreeDi.self attribute*), 54
gains_ (*apricot.optimizers.LazyGreedy.self attribute*), 44
gains_ (*apricot.optimizers.ModularGreedy.self attribute*), 56
gains_ (*apricot.optimizers.NaiveGreedy.self attribute*), 41
gains_ (*apricot.optimizers.SampleGreedy.self attribute*), 51
gains_ (*apricot.optimizers.StochasticGreedy.self attribute*), 50
GraphCutSelection (*class in apricot.functions.graphCut*), 23

L

l (*apricot.optimizers.GreeDi.self attribute*), 54

M

m (*apricot.optimizers.GreeDi.self attribute*), 54
MaxCoverageSelection (*class in apricot.functions.maxCoverage*), 15
MixtureSelection (*class in apricot.functions.mixture*), 37

N

n_samples (*apricot.functions.facilityLocation.FacilityLocationSelection attribute*), 21
n_samples (*apricot.functions.featureBased.FeatureBasedSelection attribute*), 12
n_samples (*apricot.functions.graphCut.GraphCutSelection attribute*), 24
n_samples (*apricot.functions.maxCoverage.MaxCoverageSelection attribute*), 16
n_samples (*apricot.functions.mixture.MixtureSelection attribute*), 38
n_samples (*apricot.functions.saturatedCoverage.SaturatedCoverageSele attribute*), 33
n_samples (*apricot.functions.sumRedundancy.SumRedundancySelection attribute*), 28

P

pairwise_func (*apricot.functions.graphCut.GraphCutSelection attribute*), 25
pairwise_func (*apricot.functions.saturatedCoverage.SaturatedCoverageSelection attribute*), 33
pairwise_func (*apricot.functions.sumRedundancy.SumRedundancySelection attribute*), 29
pq (*apricot.functions.mixture.MixtureSelection attribute*), 38
pq (*apricot.optimizers.ApproximateLazyGreedy.self attribute*), 47
pq (*apricot.optimizers.LazyGreedy.self attribute*), 44

R

ranking (*apricot.functions.facilityLocation.FacilityLocationSelection attribute*), 21
ranking (*apricot.functions.featureBased.FeatureBasedSelection attribute*), 12
ranking (*apricot.functions.graphCut.GraphCutSelection attribute*), 25
ranking (*apricot.functions.maxCoverage.MaxCoverageSelection attribute*), 16
ranking (*apricot.functions.mixture.MixtureSelection attribute*), 39
ranking (*apricot.functions.saturatedCoverage.SaturatedCoverageSelectio attribute*), 33
ranking (*apricot.functions.sumRedundancy.SumRedundancySelection attribute*), 29

S

SaturatedCoverageSelection (*class in apricot.functions.saturatedCoverage*), 31
submodular_functions (*apricot.functions.mixture.MixtureSelection attribute*), 38
SumRedundancySelection (*class in apricot.functions.sumRedundancy*), 27

T

transform() (*apricot.functions.facilityLocation.FacilityLocationSelection method*), 22
transform() (*apricot.functions.featureBased.FeatureBasedSelection method*), 14
transform() (*apricot.functions.graphCut.GraphCutSelection method*), 26
transform() (*apricot.functions.maxCoverage.MaxCoverageSelection method*), 17
transform() (*apricot.functions.mixture.MixtureSelection method*), 40
transform() (*apricot.functions.saturatedCoverage.SaturatedCoverageSelection method*), 34
transform() (*apricot.functions.sumRedundancy.SumRedundancySelection method*), 30

V

verbose (*apricot.optimizers.ApproximateLazyGreedy.self attribute*), 47
verbose (*apricot.optimizers.BidirectionalGreedy.self attribute*), 57
verbose (*apricot.optimizers.GreeDi.self attribute*), 54
verbose (*apricot.optimizers.LazyGreedy.self attribute*), 43
verbose (*apricot.optimizers.ModularGreedy.self attribute*), 55
verbose (*apricot.optimizers.NaiveGreedy.self attribute*), 41
verbose (*apricot.optimizers.SampleGreedy.self attribute*), 51
verbose (*apricot.optimizers.StochasticGreedy.self attribute*), 49
verbose (*apricot.optimizers.TwoStageGreedy.self attribute*), 45

W

weights (*apricot.functions.mixture.MixtureSelection attribute*), 39