apricot

Release 0.5.0

Feb 28, 2020

Getting Started

Submodular optimization for machine learning

apricot is a Python package that implements submodular optimization for the purpose of summarizing massive data sets into representative subsets. These subsets are widely useful, but perhaps the most relevant usage of these subsets are either to visualize the modalities that exist in massive data sets, or for training accurate machine learning in a fraction of the time and compute power.

CHAPTER 1

Installation

apricot can be installed using *pip install apricot-select*.

1.1 Sparse Matrices

There is built-in support for sparse matrices for both feature-based and graph-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 it can be significantly faster to explicitly use a sparse matrix representation. For feature-based functions, missing values are assumed to be feature values of 0 and thus do not increase the gain of an example. For graph-based functions, missing values are assumed to be similarities of 0 between examples. It is infrequent in reality to see a similarity of exactly 0 when the similarities are derived from feature values, so this is most likely the case when one is using a pre-defined similarity matrix or an approximation of the dense similarity matrix.

Here is an example of using a feature-based function on a very sparse matrix.

Here is an example of using a graph-based function on a very sparse matrix.

..code::python

from apricot import FacilityLocationSelection from scipy.sparse import csr_matrix

X = <a dense matrix that has many 0s in it> X_sparse = csr_matrix(X)

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

1.2 Feature-based Functions

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
$mathcalO(n^2)$ sized matrix of similarities, they can easily scale to data sets with millions of examples.

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.

1.2.1 API Reference

This file contains code that implements feature based submodular selection algorithms.

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

A selector based off a feature based submodular function.

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

This selector will optimize a feature based submodular function. Feature based functions are those that use feature values of the examples directly, like most machine learning methods do, rather than only using them indirectly through the calculation of similarity matrices, as kernel methods and facility location functions do.

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) 'min' : min(X, 1) '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, 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.

Attributes

n_samples [int] The number of samples to select.

ranking [numpy.array int] 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.

gains [numpy.array float] 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.

fit(*self*, *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 [FeatureBasedSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.3 Maximum Coverage

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.

The general form of a feature-based 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.

1.3.1 API Reference

This file contains code that implements a selector based on the maximum coverage submodular function.

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

A selector based off a coverage function.

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

This function measures the coverage of the features in a data set. The approach simply counts the number of features that take a value of 1 in at least one example in the selected set. Due to this property, it is likely that the function will saturate fairly quickly when selecting many examples unless there are also many features.

This object can be used to solve the set coverage problem, which is to identify as small a set of examples as possible that cover the entire set of features. One would simply run this approach until the gain is 0, at which point all features have been covered.

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.

Attributes

n_samples [int] The number of samples to select.

ranking [numpy.array int] 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.

gains [numpy.array float] 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.

fit(*self*, *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 [FeatureBasedSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.4 Facility Location

Facility location functions are general purpose submodular functions that, when maximized, choose examples that represent the space of the data well. 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

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

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.

1.4.1 API Reference

This file contains code that implements facility location submodular selection algorithms.

class apricot.functions.facilityLocation.**FacilityLocationSelection**(*n_samples=10*, *metric='euclidean'*, *initial_subset=None*, *optimizer='two-stage'*, *optimizer_kwds={}*, *n_neighbors=None*, *n_jobs=1*, *random_state=None*, *verbose=False*)

A selector based off a facility location submodular function.

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

This selector uses a facility location submodular function to perform selection. 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.

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, optional] Arguments to pass into the optimizer object upon initialization. Default is {}.

n_neighbors [int or None, optional] The number of nearest neighbors to keep in the KNN graph, discarding all other neighbors. This process can result in a speedup but is an approximation.

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.

Attributes

n_samples [int] The number of samples to select.

ranking [numpy.array int] The selected samples in the order of their gain.

gains [numpy.array float] 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.

fit (*self*, *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 [FacilityLocationSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.5 Graph Cut

Graph cut functions.

The general form of a graph cut function is

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

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.

1.5.1 API Reference

This code implements the graph cut function.

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

A saturated coverage submodular selection algorithm.

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

This function uses a saturated coverage based submodular selection algorithm to identify a representative subset of the data. This function works on pairwise measures between each of the samples. These measures can be the correlation, a dot product, or any other such function where a higher value corresponds to a higher similarity and a lower value corresponds to a lower similarity.

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'.

n_naive_samples [int, optional] The number of samples to perform the naive greedy algorithm on before switching to the lazy greedy algorithm. The lazy greedy algorithm is faster once features begin to saturate, but is slower in the initial few selections. This is, in part, because the naive greedy algorithm is parallelized whereas the lazy greedy algorithm currently is not. 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

'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 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

Default is 'naive'.

epsilon [float, optional] 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. Only used for stochastic greedy. Default is 0.9.

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.

Attributes

n_samples [int] The number of samples to select.

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

ranking [numpy.array int] The selected samples in the order of their gain.

gains [numpy.array float] 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.

fit(*self*, *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 [GraphCutSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.6 Sum Redundancy

Sum redundancy functions.

The general form of a sum redundancy function is

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

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.

1.6.1 API Reference

This code implements the graph cut function.

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

A selector based off a sum redundancy submodular function.

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

This selector uses a sum redundancy function to perform selection. The sum redundancy function is based on maximizing the difference between the

This selector uses a sum redundancy submodular function to perform selection. The sum redundancy 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.

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'.

n_naive_samples [int, optional] The number of samples to perform the naive greedy algorithm on before switching to the lazy greedy algorithm. The lazy greedy algorithm is faster once features begin to saturate, but is slower in the initial few selections. This is, in part, because the naive greedy algorithm is parallelized whereas the lazy greedy algorithm currently is not. 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

> '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 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

Default is 'naive'.

epsilon [float, optional] 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. Only used for stochastic greedy. Default is 0.9.

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.

Attributes

n_samples [int] The number of samples to select.

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

ranking [numpy.array int] The selected samples in the order of their gain.

gains [numpy.array float] 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.

fit (*self*, *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 [SumRedundancySelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.7 Saturated Coverage

Saturated coverage functions.

The general form of a sum redundancy function is

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

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.

1.7.1 API Reference

This code implements saturated coverage functions.

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*, *optimizer_kwds={}*, *verbose=False*)

A saturated coverage submodular selection algorithm.

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

This function uses a saturated coverage based submodular selection algorithm to identify a representative subset of the data. This function works on pairwise measures between each of the samples. These measures can be the correlation, a dot product, or any other such function where a higher value corresponds to a higher similarity and a lower value corresponds to a lower similarity.

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'.

n_naive_samples [int, optional] The number of samples to perform the naive greedy algorithm on before switching to the lazy greedy algorithm. The lazy greedy algorithm is faster once features begin to saturate, but is slower in the initial few selections. This is, in part, because the naive greedy algorithm is parallelized whereas the lazy greedy algorithm currently is not. 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

'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 'greedi' : the GreeDi distributed algorithm 'bidirectional' : the bidirectional greedy algorithm

Default is 'naive'.

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.

Attributes

n_samples [int] The number of samples to select.

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

ranking [numpy.array int] The selected samples in the order of their gain.

gains [numpy.array float] 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.

fit(*self*, *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 [SaturatedCoverageSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.8 Mixtures

A convenient property of submodular functions is that the addition of two of them is still a submodular function. 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.

1.8.1 API Reference

This file contains code that implements mixtures of submodular functions.

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

A selection approach based on a mixture of submodular functions.

This class implements a simple mixture of submodular functions for the purpose of selecting a representative subset of the data. The user passes in a list of instantiated submodular functions and their respective weights to the initialization. At each iteration in the selection procedure the gains from each submodular functions will be scaled by their respective weight and added together.

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.

submodular_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, 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, optional] Whether to print output during the selection process.

Attributes

pq [PriorityQueue] The priority queue used to implement the lazy greedy algorithm.

n_samples [int] The number of samples to select.

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

weights [numpy.ndarray] The weights of each submodular function.

ranking [numpy.array int] 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.

gains [numpy.array float] 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.

fit(*self*, *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 [MixtureSelection] The fit step returns this selector object.

fit_transform(*self*, *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(*self*, *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.

1.9 Naive Greedy

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.

1.9.1 API Reference

The naive greedy algorithm for optimization.

This optimization approach is the naive greedy algorithm. At each iteration of selection it will simply calculate the gain one would get from adding each example, and then will select the example that has the highest gain. This algorithm is conceptually simple and easy to parallelize and put on a GPU, but can be slower than other alternatives because it involves repeatedly evaluating examples that are not likely to be selected next.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.10 Lazy Greedy

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.

1.10.1 API Reference

The lazy/accelerated greedy algorithm for optimization.

This optimization approach is the lazy/accelerated greedy algorithm. It will return the same subset as the naive greedy algorithm, but it uses a priority queue to store the examples to prevent the repeated evaluation of examples that are unlikely to be the next selected one. The benefit of using this approach are that using a priority queue can significantly improve the speed of optimization, but the downsides are that maintaining a priority queue can be costly and that it's difficult to parallelize the approach or put it on a GPU.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.pq [utils.PriorityQueue] The priority queue used to order examples for evaluation.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.11 Two-Stage Greedy

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.

1.11.1 API Reference

An approach that uses both naive and lazy greedy algorithms.

This optimization approach starts off by using the naive greedy algorithm to select the first few examples and then switches to use the lazy greedy algorithm. This two stage procedure is designed to overcome the limitations of the

lazy greedy algorithm—specifically, the inability to parallelize the implementation. Additionally, early iterations frequently require iterating over most of the examples, which is particularly costly to do on a priority queue. Thus, one can frequently see large speed gains simply by doing the first few iterations using the naive greedy algorithm and then switching to the lazy greedy algorithm.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

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

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

1.12 Approximate Lazy Greedy

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.

1.12.1 API Reference

The approximate lazy/accelerated greedy algorithm for optimization.

This optimization approach is the lazy/accelerated greedy algorithm. It will return the same subset as the naive greedy algorithm, but it uses a priority queue to store the examples to prevent the repeated evaluation of examples that are unlikely to be the next selected one. The benefit of using this approach are that using a priority queue can significantly improve the speed of optimization, but the downsides are that maintaining a priority queue can be costly and that it's difficult to parallelize the approach or put it on a GPU.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.pq [utils.PriorityQueue] The priority queue used to order examples for evaluation.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.13 Stochastic Greedy

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.

1.13.1 API Reference

The stochastic greedy algorithm for optimization.

This optimization approach is the stochastic greedy algorithm proposed by Mirzasoleiman et al. (https://las.inf.ethz.ch/files/mirzasoleiman15lazier.pdf). This approach is conceptually similar to the naive greedy algorithm except that it only evaluates a subset of examples at each iteration. Thus, it is easy to parallelize and amenable to acceleration using a GPU while maintaining nice theoretical guarantees.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

epsilon [float, optional] 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.

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

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.14 Sample Greedy

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.

1.14.1 API Reference

The stochastic greedy algorithm for optimization.

This optimization approach is the stochastic greedy algorithm proposed by Mirzasoleiman et al. (https://las.inf.ethz.ch/files/mirzasoleiman15lazier.pdf). This approach is conceptually similar to the naive greedy algorithm except that it only evaluates a subset of examples at each iteration. Thus, it is easy to parallelize and amenable to acceleration using a GPU while maintaining nice theoretical guarantees.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

epsilon [float, optional] The sampling probability to use when constructing the subset. A subset of size n * epsilon will be selected from.

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

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.15 GreeDi

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.

1.15.1 API Reference

An approach for optimizing submodular functions in parallel.

This optimizer implements the GreeDi method for selecting sets in parallel by Mirzasoleiman et al. (https://papers.nips.cc/paper/5039-distributed- submodular-maximization-identifying-representative-elements-in-massive- data.pdf).

Briefly, this approach splits the data into m partitions uniformly at random, selects l exemplars from each partition, and then runs a second iteration of greedy selection on the union of l*m exemplars from each partition to get the top k.

Parameters

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

m [int] The number of partitions to split the data into.

l [int] 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.

optimizer1 [str or base.Optimizer, optional] The optimizer to use in the first stage of the process where l exemplars are selected from each partition. Default is 'lazy'.

optimizer2 [str or base.Optimizer, optional] 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'.

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

verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.BaseSelection or None] 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.

self.m [int] The number of partitions that the data will be split into.

self.l [int] The number of exemplars that will be selected from each partition.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.16 Modular Greedy

The modular greedy optimizer uses the modular upper-bounds for the gain of each example to do 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.

1.16.1 API Reference

The stochastic greedy algorithm for optimization.

This optimization approach is the stochastic greedy algorithm proposed by Mirzasoleiman et al. (https://las.inf.ethz.ch/files/mirzasoleiman15lazier.pdf). This approach is conceptually similar to the naive greedy algorithm except that it only evaluates a subset of examples at each iteration. Thus, it is easy to parallelize and amenable to acceleration using a GPU while maintaining nice theoretical guarantees.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

epsilon [float, optional] 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.

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

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

1.17 Bidirectional Greedy

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. 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.

1.17.1 API Reference

The bidirectional greedy algorithm.

This is a stochastic algorithm for the optimization of submodular functions that are not monotone, i.e., where f(A union {b}) is not necessarily greater than f(A). When these functions are not monotone, the greedy algorithm does not have the same good guarantees on convergence, whereas the bidirectional greedy algorithm does.

Generally, it is difficult to control the number of examples that are returned by the bidirectional greedy algorithm. This can be done by tuning a hyperparameter that regularizes the gains until you get the right set size. Here we take another approach and randomly select examples for consideration until we achieve the appropriate set size. This means that not all examples will be considered for addition.

Parameters

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

Attributes

self.function [base.SubmodularSelection] A submodular function that implements the *_calculate_gains* and *_select_next* methods. This is the function that will be optimized.

self.verbose [bool] Whether to display a progress bar during the optimization process.

self.gains_ [numpy.ndarray or None] The gain that each example would give the last time that it was evaluated.

Python Module Index

a

Index

S

T