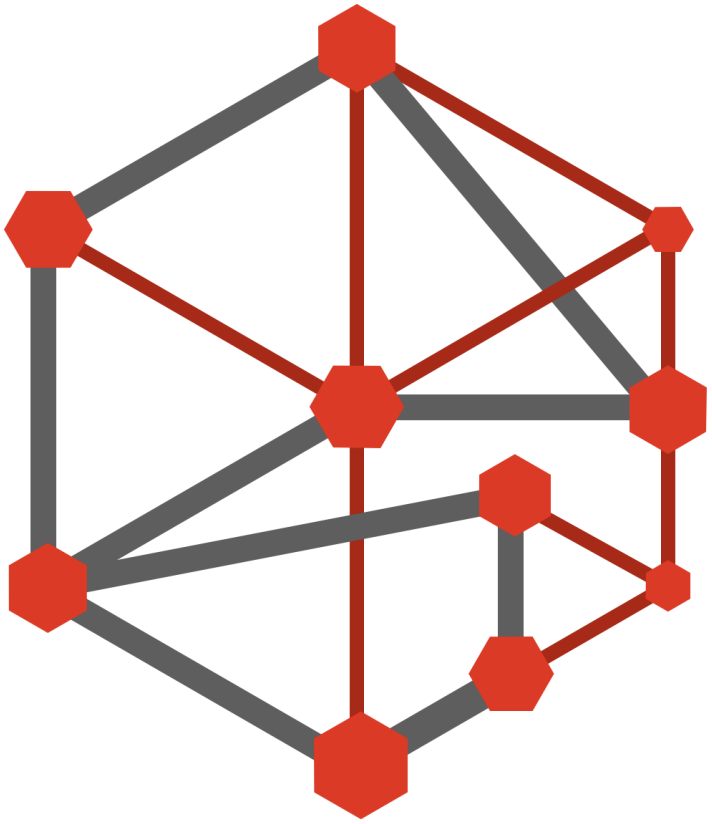


HEXADO1

SOFTWARE DEVELOPMENT TEAM

Aidan Zorbas Dawson Brown Jenny Yu

Jingrun Long Kara Autumn Jiang Vanessa Pierre

Scikit-learn Hard Issues Report

Assignment 4 Deliverable

Table of Contents

Hard Level Scikit-learn Issues

During this developmental phase, our team identified two Hard-Level issues within the scikit-learn source code repository. These issues are as described below:

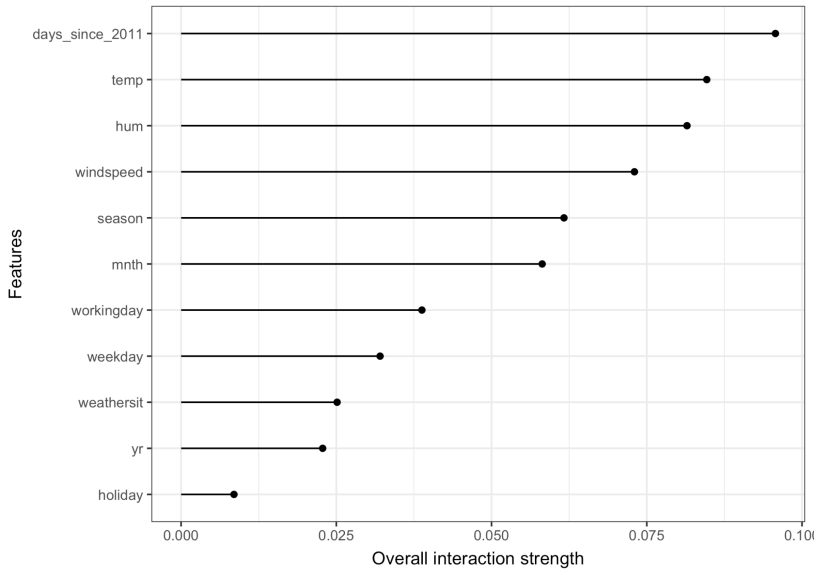
Issue #22383: Implement Friedman's H-Statistic

Issue Description

From a high-level, this issue involves the implementation of Friedman's H-Statistic, which is a measure of whether there's significant interaction between variables in a model given the marginal and pairwise partial dependence; as a library of functions which would be available to the user. More specifically, Friedman's H-Statistic examines the interaction between two features, then determines how strongly these features interact with each other within a model. Using this selection of features, Friedman's H-Statistic can be used to determine how strongly any one feature interacts with any other features within the model.

Interactions can essentially be broken down into two cases – the first being the case in which two features either do not interact with one another, or any other features within the model; and the second being the case where interactions do exist between these features. Both of these cases are addressed using computationally expensive calculations as described on the linked documentation for implementing Friedman's H-Statistic.

Within practical use cases, Friedman's H Statistic can be used as means of predicting features to determine correlation/causation through the analysis of features such as the number of rented bikes given weather and calendrical events based on interaction strength. The following graph from the linked documentation demonstrates the feature analysis between features and overall interaction strength for this example:



Issue Advantages

As an advantage of implementing Friedman's H-Statistic within scikit-learn as a library of functions with underlying theory through the partial dependence decomposition which is defined as the portion of variance explained by the interaction, users would be given access to a complex set of tools which detects various forms of interactions regardless of specific forms, allowing them to analyze arbitrary higher interactions (including over three or more features). Since a correct implementation of Friedman's H-Statistic is non-trivial, this issue resolves a number of challenges that users face when trying to implement Friedman's H-Statistic using existing scikit-learn tools.

Issue Disadvantages

The primary challenge that we faced considering this issue is acknowledging that a correct implementation of Friedman's H-Statistic is non-trivial.

Since there exists many (often conflicting) interpretations on how to implement the algorithm, and which datasets are considered valid input, all existing documentation and implementation attempts point towards high computational complexity with a high degree of instability during execution which present themselves as a plethora of issues related to sample size scaling and testing.

According to the documentation, the algorithm may yield unexpected results over small samples due to not having insufficient data points; but even with larger sample sizes, not all data points are used, which may cause the estimates and subsequent results to become unstable. Finally, it is difficult to determine whether or not a H-statistic is sufficiently large to be considered a 'strong' interaction. These issues present themselves as a challenge for both providing a correct implementation of Friedman's H-Statistic algorithm, as well as in determining acceptance conditions for testing.

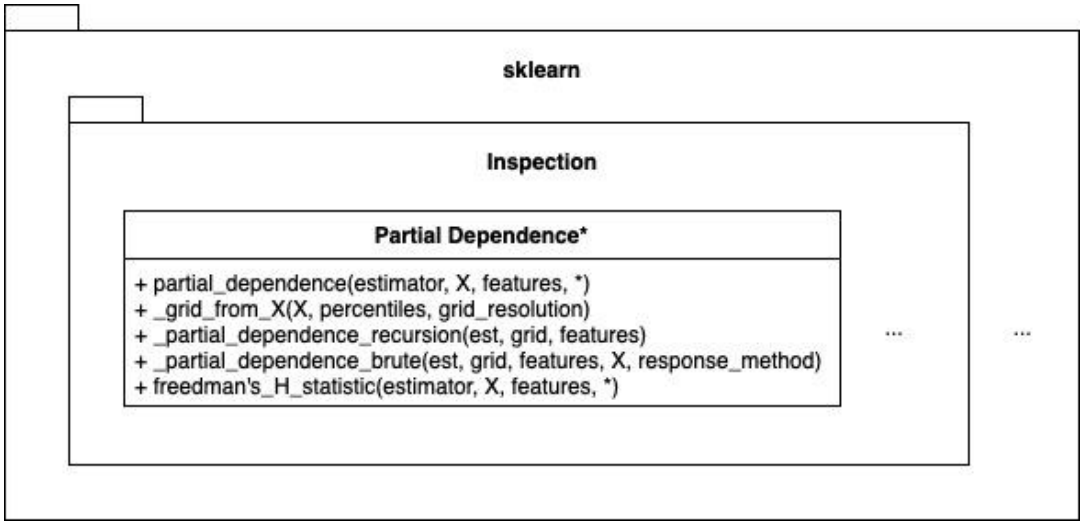
Related Components and Organization

sklearn/inspection/_partial_dependence.py > friedmans_H_statistic()

Since this issue involves the implementation of an interaction statistic which has an underlying theory through the partial dependence decomposition, and much of this functionality currently exists within scikit-learn's _partial_dependence.py file, we propose implementing Friedman's H-Statistic as a new user-accessible function within this file.

As a new function which depends on partial dependence decomposition, a new Friedman's H-Statistic function would be able to conveniently make use of existing functions within this file such as `partial_dependence()`, `_partial_dependence_brute()`, and `_partial_dependence_recursion()` as subroutines, while at the same time providing users with a convenient means of accessing related functionality.

UML Diagram



Issue #14214: Implement Bisecting K-Means

Issue Description

The issue represents a request for an implementation of the Bisecting K-Means clustering algorithm. The Bisecting K-Means (BKM) algorithm is a clustering algorithm which involves repeatedly applying a standard K-means algorithm with a K value of 2. It is used to find natural groupings in datasets with n features.

The algorithm begins by grouping all data points into a single cluster, and then dividing this initial cluster into two clusters (bisection). The cluster with the worst sum-of-squared-errors is selected, and is bisected with the standard K-means clustering algorithm. The bisection of the selected cluster is repeated several (value manually provided) times, and the best split is selected. This process of worst-cluster selection and K-means bisection is repeated until a user defined number of clusters is reached.

Related Components

scikit-learn/sklearn/cluster/_bisecting_k_means.py
scikit-learn/sklearn/cluster/_kmeans.py

Bisecting K-Means Feature

Issue Selection Decision

Having examined a number of hard-level issues within the scikit-learn repository, and consulted with a TA on the difficulty of another un-labelled issue (which was ultimately deemed not to be difficult enough), we decided on the Bisecting K-Means Implementation issue. We found that many issues were reasonable to implement given our understanding of scikit-learn, but were often comprised of numerous highly repetitive tasks. Due to the large size of scikit-learn, these tasks which required repeating for every class which implemented a certain interface, or for all test suites would be impossible to complete on time. Additionally, many issues were broad feature requests without clear acceptance conditions, or simply beyond the scope of our team's understanding of machine learning and statistics. Our issue selection process focused on identifying an issue with clearly defined acceptance criteria which was achievable with our current technical knowledge and a reasonable amount of research.

The Bisecting K-Means algorithm met all the criteria we desired in a hard-level feature. As a clustering algorithm, and a variation on the existing K-means algorithm, BKM has clear and well established desired outcomes which can be straightforwardly tested for correctness. Precedents set by existing clustering algorithms made test design clear. The BKM algorithm is also well-documented and reasonably complex without requiring graduate-level knowledge of machine-learning and statistical esoterica. Some technical references were provided in the initial issue itself. The reliance of the algorithm on existing K-Means clustering also meant that at the outset of development the team had reference material of an algorithm that was both similar, and a component of our new implementation.

With these factors taken into consideration, we felt that the BKM feature was the issue most suited to the course structure, difficulty requirements, and our development capabilities.

Our Feature Implementation

The newly requested feature conveniently fits into the existing structure of scikit-learn. Since Bisecting K-means is a K-means-based clustering algorithm, it follows that it should implement the same interfaces and use cases as existing K-means variants. As such, our implementation extends the `BaseEstimator`, `ClusterMixin`, and `TransformerMixin` classes. We follow this specification by including fit, score, predict, transform, and combination functions such as `fit_predict()` and `fit_transform()`. Our implementation also includes an instance of the base K-means class to use in interactive bisecting steps, favouring composition over inheritance. Our BisectingKMeans class's `__init__()` method takes all the same parameters as a standard K-Means object, using them (`n_clusters` excluded) to initialize the contained K-means instance.

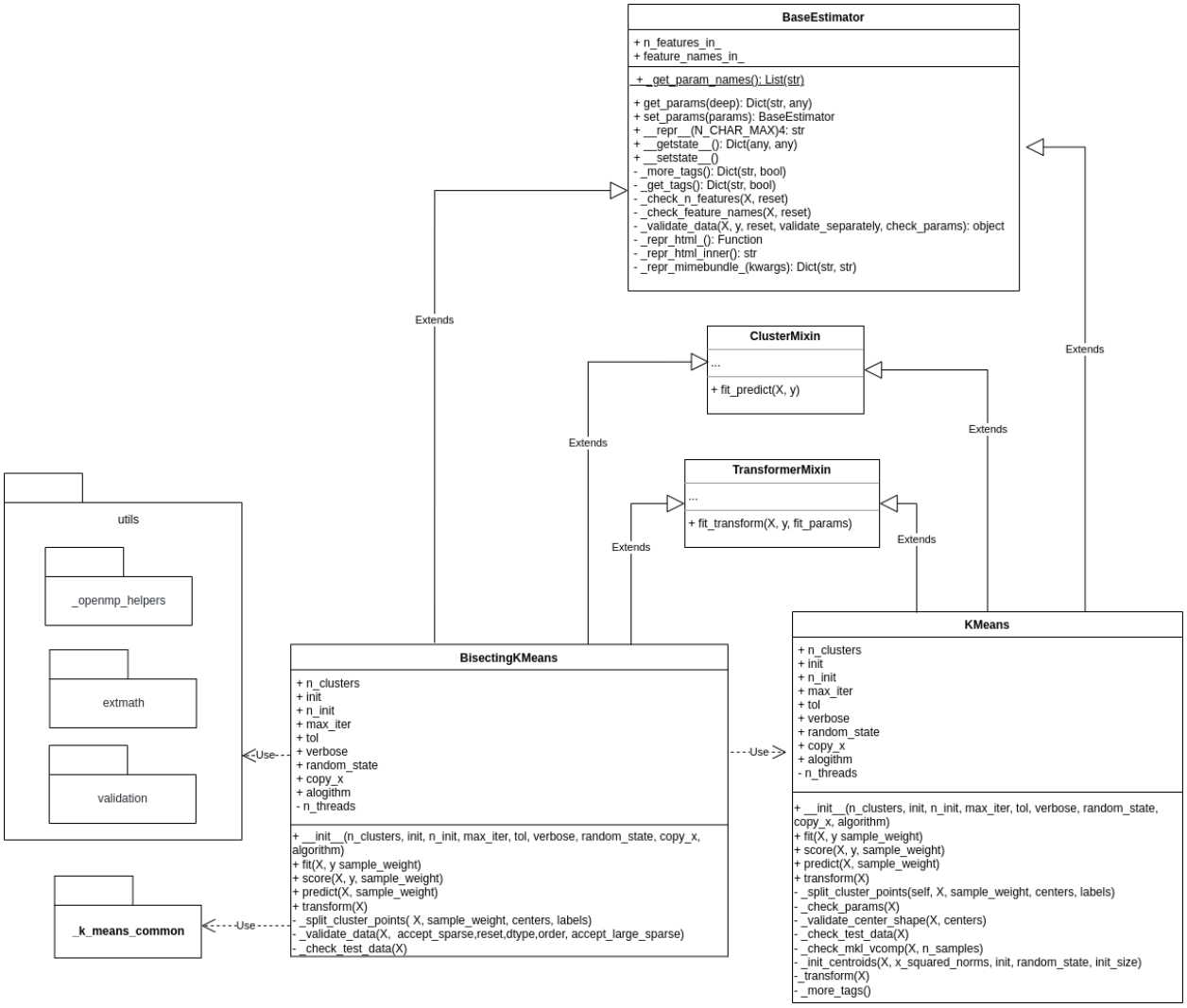
The most important aspect of our implementation was the `fit()` method. This method contains the code responsible for performing the clustering – finding n centroids in the dataset. This method follows the bisecting k-means specification, selecting and splitting the cluster with the worst sum-of-squared errors using a call to the base K-means instance with K=2. The `fit()` method relies on the `_split_cluster_points` method, which in turn uses `_k_means_common` inertia calculation functions. `_split_cluster_points` is used to take the newly bisected cluster and separate it into two distinct data arrays, as well as calculate the inertia for each individual cluster.

Simpler methods include `fit_predict()` and `fit_transform()`, which are simple convenience compositions of existing functions (`fit()` and `predict()` or `fit()` and `transform()` respectively), which returns the euclidean distances of points from their cluster centers. The `predict()` method simply returns which cluster a provided list of points provided by using sum-of-squared-errors as a distance metric.

The `score()` function returns the value opposite (negative) of the value obtained from computing the Bisecting K-Means objective. The objective is simply the smallest possible sum-of-squared-distances between each datapoint and their respective cluster centers.

Organization and UML

The newly created class is included in scikit-learn/sklearn/cluster alongside the existing scikit-learn clustering algorithms. The provided UML outlines which existing classes the new class implements and uses, as well as which packages it draws on for utility functions.



User Guide

Usage of the `BisectingKMeans` class is almost identical to the functionality of the base K-Means class. Abridged details with specifications of minor variation are included as follows for clarity and simplicity:

Initialization Parameters:

`n_clusters`

The number of clusters (and thus centroids) to generate.

`init`

One of `'k-means++'` or `'random'`. Determines which method the internal base k-means uses for centroid initialization. Note that unlike with `KMeans`, this method does not allow for an

array-like or callable to be passed in, as `BisectingKMeans` does not initialize with `n_clusters` centroids.

`n_init`

The number of times the internal base k-means will attempt clustering with different centroid seeds..

`max_iter`

The max number of iterations run by the internal base k-means.

`tol`

Tolerance used in the internal k-means algorithm for determining convergence

`verbose`

The level of verbosity of the internal base k-means.

`random_state`

Sets the random state for centroid initialization.

`copy_x`

Whether or not the internal base k-means will precompute distances.

`algorithm`

One of "`auto`", "`full`", or "`elkan`", determines which variant of the k-means algorithm used in each step.

Public Methods

`fit(X, y, sample_weight)`

Takes X, an array-like or sparse matrix of shape (`n_sample, n_features`), and sample_weight, an optional array-like of shape (`n_samples`). y is included for API consistency but is ignored. X represents the data to perform clustering on, and `sample_weight` represents the weight for each sample in corresponding parameter `X`. `fit`() computes bisecting k-means clustering on X with sample_weight according to the parameters specialized on initialization, and returns the instance of k-means, fitted to the data.

`score(X, y, sample_weight)`

Takes X, an array-like or sparse matrix of shape (`n_sample, n_features`), and sample_weight, an optional array-like of shape (`n_samples`). y is included for API consistency

but is ignored. X represents the data to perform clustering on, and `sample_weight` represents the weight for each sample in corresponding parameter X.

`transform(X)`

Takes X, an array-like or sparse matrix of shape (`n_sample, n_features`). Returns an array-like object of shape (`n_samples, n_clusters`) which contains the euclidean distances of points from their cluster centres.

`predict(X, sample_weight)`

Takes X, an array-like or sparse matrix of shape (`n_sample, n_features`), and sample_weight, an optional array-like of shape (`n_samples`). y is included for API consistency but is ignored. X represents the data to perform clustering on, and `sample_weight` represents the weight for each sample in corresponding parameter X. Returns an array of shape (`n_samples`) with the index of the cluster each sample belongs to.

`fit_predict(X, y, sample_weight)`

Equivalent to `fit(X,y,sample_weight)`, followed by `predict(X,y,sample_weight)`

`fit_transform(X, y, sample_weight)`

Equivalent to `fit(X,y,sample_weight)`, followed by `predict(X)`

Effects on Existing Functionality

No existing functionality within scikit-learn has been affected by our feature additions. Our contributions are entirely additive, comprising a new estimator class. The addition provides users with the option to use the Bisecting K-Means algorithm for clustering in addition to existing options.

Bisecting K-Means Unit Test Suite

Summary of Unit Test Coverage

Dataset Number	Description	Provided Input
Dataset 1	2 data points and 2 clusters	K = 2 n_features = 1 X_f = [[0.2] [0.8]] X_p = [[0.3] [0.9]] X_t = [[0.34] [0.94]]
Dataset 2	4 data points and 2 clusters	K = 2 n_features = 2 X_f = [[0.8 0.5] [0.77 0.58] X_p = [[0.26 0.42] [0.70 0.47] [0.47 0.83]] X_t = [[0.83, 0.92] [0.27, 0.18] [0.49, 0.03]]
Dataset 3	7 data points and 7 clusters	K = 7 n_features = 2 X_f = [[0.80 0.50] [0.74 0.49] [0.77 0.58] [0.34 0.40] [0.20 0.43] [0.25 0.38] [0.63 0.82]] X_p = [[0.26 0.42] [0.70 0.47] [0.47 0.83]] X_t = [[0.34 0.43] [0.94 0.34]]
Dataset 4	7 data points and 3 clusters	K = 3 n_features = 2 X_f = [[0.80 0.50] [0.74 0.49] [0.77 0.58] [0.34 0.40] [0.20 0.43] [0.25 0.38] [0.63 0.82]] X_p = [[0.26 0.42] [0.70 0.47]

		[0.47 0.83]] X_t = [[0.34 0.43] [0.94 0.34]]

When building scikit-learn from source with the bugfix applied, all test sets defined above pass, and additionally, the existing scikit-learn unit test suite also passes.

Unit Test Groups & Acceptance Conditions

Test Group	Method Invoked	Acceptance Conditions
Test Group 1	`fit()`	1.1: Correct values for cluster centers. 1.2: Datapoints belong to the correct cluster. 1.3: Correct dimensions for result.
Test Group 2	`predict()`	2.1: Correct dimensions for result. 2.2: Datapoints belong to the correct cluster.
Test Group 3	`transform()`	3.1: Correct dimensions for result. 3.2: Correct values for result.
Test Group 4	`score()`	4.1: Correct score generated.
Test Group 5	`fit_predict()`	5.1: Correct dimensions for result. 5.2: Correct values for result. 5.3: fit_predict() and fit().predict() are equivalent.
Test Group 6	`fit_transform()`	6.1: Correct dimensions for result. 6.2: Correct values for result. 6.3: fit_transform() and fit().transform() are equivalent.

With each dataset, each test group provides a K value that is used to create a `BisectingKMeans` estimator, which is then fitted using X_f. Once the data is fitted, the test script verifies that each data point was fitting into the correct cluster. Using the fitted estimator, `predict()` and `transform()` are then invoked using the data points X_p and X_t respectively. The test script verifies that each data point in X_p was placed into the correct cluster, and that each datapoint in X_t was correctly transformed into a X_f cluster-distance space. Then, `score()` is called on the fitted estimator with the parameter X_f, and the resulting value is then verified.

Finally, `fit_predict()` and `fit_transform()` are invoked on the base `BisectingKMeans` estimator with the parameter X_f. `fit_predict()` verifies that each data point was assigned to the correct cluster, and that the results are equivalent to invoking `fit()` followed by `predict()`. Similarly, `fit_transform()` verifies that each data point was

correctly transformed into X_f cluster-distance space, and that the results are equivalent to invoking `fit()` followed by `transform()`.

Running and Reading Unit Tests

Our Unit Test suite has been included within a dedicated testing folder within our A4 directory. To run these unit tests, please first build scikit-learn from source using the main branch of our scikit-learn project fork, and then use 'python3 testBisectingKMeansUnit.py' to run the Unit test suite.

Once this test suite has been executed, you will notice that our tests have been grouped by the four datasets that they were invoked on. There are six test groups conducted within each dataset, each with their respective test cases.

For your convenience, each dataset is numbered and labelled according to the descriptions provided in the table above. Similarly, the standard input provided as arguments is given within the dataset to allow for ease-of-reference when reviewing results of individual test groups and cases.

Each test group is labelled according to the specific method being tested, and provided the received output for ease of reference. Within each group, each test case is numbered according to the scheme [test group number].[test case number]. These test cases are preceded by a pass or fail status marker, and followed by their corresponding acceptance condition.

Bisecting K-Means Acceptance Testing Suite

Summary of Acceptance Test Coverage

Test	Plotting Condition	Arguments	Instantiatiation Params
Test 0	Small number of samples and features over a large number of clusters	`n_samples=200, n_features=2, fitted_clusters=20, predicted_clusters=20, n_clusters=20, n_init=10, max_iter=300, fit_random_state=0, predict_random_state=1`	`n_clusters=20, init='random', n_init=10, max_iter=300, tol=1e-04, random_state=0`
Test 1	Fewer generated clusters than provided value for n_clusters	`n_samples=2000, n_features=2, fitted_clusters=20, predicted_clusters=20, n_clusters=30 n_init=10, max_iter=300, fit_random_state=0, predict_random_state=1`	`n_clusters=30, init='random', n_init=10, max_iter=300, tol=1e-04, random_state=0`
Test 2	More generated clusters than provided value for n_clusters	`n_samples=2000, n_features=2, fitted_clusters=30, predicted_clusters=30, n_clusters=20, n_init=10, max_iter=300, fit_random_state=0, predict_random_state=1`	`n_clusters=20, init='random', n_init=10, max_iter=300, tol=1e-04, random_state=0`

Since the use of the `BisectingKmeans` class is very similar to existing `Kmeans` class, and these classes can be used interchangeably, our acceptance test suite was designed to allow users to compare plotted results of calling `fit()` and `predict()` using either Bisecting K-Means or K-Means given the same datasets and configuration.

By calling `fit()` and `predict()`, our acceptance testing suite was designed to produce two plots for each test case per selected estimator. These plots are labelled according to their respective test case number, class, and invoked method. By visually comparing the results of invoking the `fit()` and `predict()` methods within the `BisectingKmeans` class, we can

verify that our implementation works correctly, and generates results comparable to those of the built in scikit-learn `Kmeans` class.

Running and Reading Acceptance Tests

Our Acceptance Test script has been included within a dedicated testing folder within our A4 directory. To run this script, please first build scikit-learn from source using the main branch of our scikit-learn project fork, and then use 'python3 testBisectingKMeansAcceptance.py' to run the acceptance test script.

When this script is executed, you will be prompted to generate and view plots for either Bisecting K-Means or K-Means. Once an algorithm has been selected, the script will systematically output conditions and arguments of the test(s) being run, followed by the system status as the selected class is instantiated and used to fit/predict the provided data as well as the inertia for the fitted data. Once this is finished, four plots will be generated in the Tests directory. These plots will be numbered and labelled as described above. These steps can be repeated to (re)generate and view plots for both Bisecting K-Means and K-Means in order to compare algorithmic performance.