Extra Credit for Assignments 1-4

* **Due** May 4 by 6pm

* **Points** 0

* **Available** until May 4 at 6pm

**These extra credit problems only count toward your homework grade and can only bring you up to a maximum of 100% over the first 4 assignments (that considers any EC you did for those problems as well):**

These problems may take you a long time, **SO DON'T PROCRASTINATE.** But for people who have missed a few assignments, this is a chance to bring your grade up if you're determined to stay in this class. These are all or none problems. No partial credit. You must complete the first task in order to do the second and third. **Please put all of your code for these tasks in a single python file named extra\_credit.py inside of a folder named Extra\_Credit in your CLASS REPO. These are popular algorithms and I know that you will find sample code on the internet. I can find it too. I will check to make sure you've written your own code for these. PLEASE do not copy someone else's code (from the internet or a classmate). Anyone caught doing so will be reported immediately to the Honor Board. I've created this EC work to help some of you pull up your grade. Don't make me regret doing so. You will learn a lot from these tasks.**

**Task #1:** (10 pts) You will implement a KD-tree data structure for finding the nearest neighbors in a data set for any given point. There are many ML algorithms which use this. DBSCAN, kNN classifiers, and k-means clustering are some examples. You should begin by watching[this video (Links to an external site.)](https://www.youtube.com/watch?v=qLnMoPPMX9Q)[](https://www.youtube.com/watch?v=qLnMoPPMX9Q)

You will create a class called KDtree. The \_\_init\_\_ method should have one argument: an array of data. This should work for any number of dimensions (hence the K). You don't have to worry about things like a limit on the recursion depth. I won't test it on any crazy data. But I will test it on data with multiple dimensions and make sure that it works.

The KDtree class should have an instance method called **nn\_within\_r(point, radius)**. It takes two arguments:

1. point: this is an array of values representing a single query point that you wish to find the nearest neighbors of. For example, it could be a single row in your data set.
2. radius: this is a float representing the radius that you will search around a query point (a hypersphere) for nearest neighbors. This similar to the box query that Sedgewick gave as an example. You can just consider the smallest hypercube that circumscribes the hypersphere and manually check all the points to make sure they're within a distance <= radius of the query point. Or do it however you like.

This method returns an array of the points that are within a distance <= radius of the query point.

The KDtree class should have another instance method called **k\_nearest\_neighbors(point, k, distance\_func)**. It takes two arguments:

1. point: this is an array of values representing a single query point that you wish to find the nearest neighbors of. For example, it could be a single row in your data set.
2. k: this is the number of the nearest neighbors to return. For instance, if k=3, then you should return the 3 points in the KDtree instance which are closest to the query point. **You must use**[**this (Links to an external site.)**](https://pymotw.com/2/bisect/) to store the values as you traverse the tree.
3. distance\_func: this is a function that takes in 2 points (which are arrays of values) and returns a [distance (Links to an external site.)](https://docs.scipy.org/doc/scipy-0.18.1/reference/spatial.distance.html), see the scipy distance methods as an example. You will use this distance function to calculate the distance between two points. E.g. distance = distance\_func(point1, point2).

This method returns an array of the k points that are nearest (according to distance\_func) to the query point. After using bisect to store the values in a list, it will be as easy as returning the\_list\_of\_value[:k]. However, you must traverse the tree intelligently! No brute force...

**Task #2)**: (10 pts) You will manually implement the DBSCAN algorithm. You must use your own KDtree object that you created in Task #1 to search for the nearest neighbors within epsilon of each point. Start with[this video (Links to an external site.)](https://www.youtube.com/watch?v=5E097ZLE9Sg)[](https://www.youtube.com/watch?v=5E097ZLE9Sg)You will create a function **dbscan(data, epsilon, min\_points)**. It takes three arguments.

1. data: this is a pandas DataFrame representing the data. You can assume that the data will only have numerical features.
2. epsilon: this is a float representing the distance from a given point that you will search for neighbors within.
3. min\_points: this is the minimum number of neighboring points that must be within a distance epsilon from any point in order for it to be considered a core point.

This function should return the original DataFrame with two additional columns added to it: point\_type and cluster\_label

1. point\_type: the value in this column represents the type of data point based on DBSCAN's criteria. The value will be one of the following strings: 'core', 'boundary', 'noise'
2. cluster\_label: the value in this column represents which cluster the corresponding data point belongs to. The value will be a string. Cluster labels should be in the following format "Cluster *N*" where N is a number from 1 - inf. The first cluster you find N=1, the second N=2, and so on. All noise points should have a cluster\_label value of "NOISE."

**Task #3:** (10 pts) You will create a class for a kNN classfier/regressor called **KNN**. I will cover this in class on March 23rd. Or you can just watch[this video (Links to an external site.)](https://www.youtube.com/watch?v=k_7gMp5wh5A)[](https://www.youtube.com/watch?v=k_7gMp5wh5A)It's a simple algorithm. Don't panic. For this implementation, you must use your own KDtree that you created in Task #1 for finding the nearest neighbors.

The \_\_init\_\_ method of this class takes 2 arguments: X, Y and k

* X: this is a pandas DataFrame with the data to be used for the kNN model.
* Y: this is a pandas Series with the labels for the data. This can be class labels in a classification problem or numerical values in a regression problem
* k: this is the number of nearest neighbors you will use when performing a class prediction.

The KNN class should have an instance method called **predict(data\_point)** which takes one argument:

* data\_point: this is an array of values. You should find the k nearest neighbors to this point and take the most frequent class or find the average numerical value. **For classification, If there is a tie**, sort the class labels and return the first one as the answer.

If the labels (Y) used in training the model were class labels, the predict method should return a tuple that contains:

1. a string representing the predicted class label for the data\_point passed
2. the posterior probability of the class label returned (i.e. (# of points with the class label)/k)

If the labels (Y) used in training the model were numerical values, the predict method should return a tuple that contains:

1. the average value (\muμ) of the k nearest neighbors
2. the average absolute distance of the data points \frac{\sum\left|x_i-\mu\right|}{k}&Sum;|xi−μ|k

**Task #4:** (10 pts) You will create a class for k-means clustering called **KMeans**. You don't have to implement the filtering improvement (no KDtrees necessary) unless you want to challenge yourself :)

The \_\_init\_\_ method for this class will take 2 arguments:

* data: this will be a pandas DataFrame containing the data you wish to find clusters within.
* k: this will be an integer representing the number of clusters you want to find

The KMeans class should have an **class method** called **scree\_plot(X, max\_k)**which plots the number of clusters (k) vs the average [silhouette score (Links to an external site.)](https://en.wikipedia.org/wiki/Silhouette_(clustering)) of the clustering, from k=1 to k=max\_k. You can use scikit's [silhouette\_score (Links to an external site.)](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html" \l "sklearn.metrics.silhouette_score" \o "" \t "_blank) for calculations **BUT** the labels you pass to silhouette\_score must be calculated by your k-means implementation, not scikit's. [This (Links to an external site.)](http://stats.stackexchange.com/questions/12819/how-to-draw-a-scree-plot-in-python) will help with the plotting. [This (Links to an external site.)](http://www.sthda.com/english/wiki/determining-the-optimal-number-of-clusters-3-must-known-methods-unsupervised-machine-learning#average-silhouette-method) is similar to the scree plot that I'm looking for (scroll down a little till you see the plots).

The KMeans class should have an **instance method** called **cluster\_data** which takes no arguments and returns a pandas DataFrame with the original data plus an additional column named **"cluster\_label"** that contains a string for the cluster that each datum (row in the data set) belongs to. The cluster labels should be formatted like "Cluster *n*" where n is a value from 1 to k.