Veda Samhitha Manne **1002030416**

CSE 6363: Machine Learning
Project 3: Report

K-means Clustering

K-Means Clustering:

K-means clustering is a Popular unsupervised machine learning algorithm that divides a set of into K clusters. The algorithm operates iteratively by maximizing the clustering goal and by updating the cluster assignments of each data point and the centroids of each cluster.

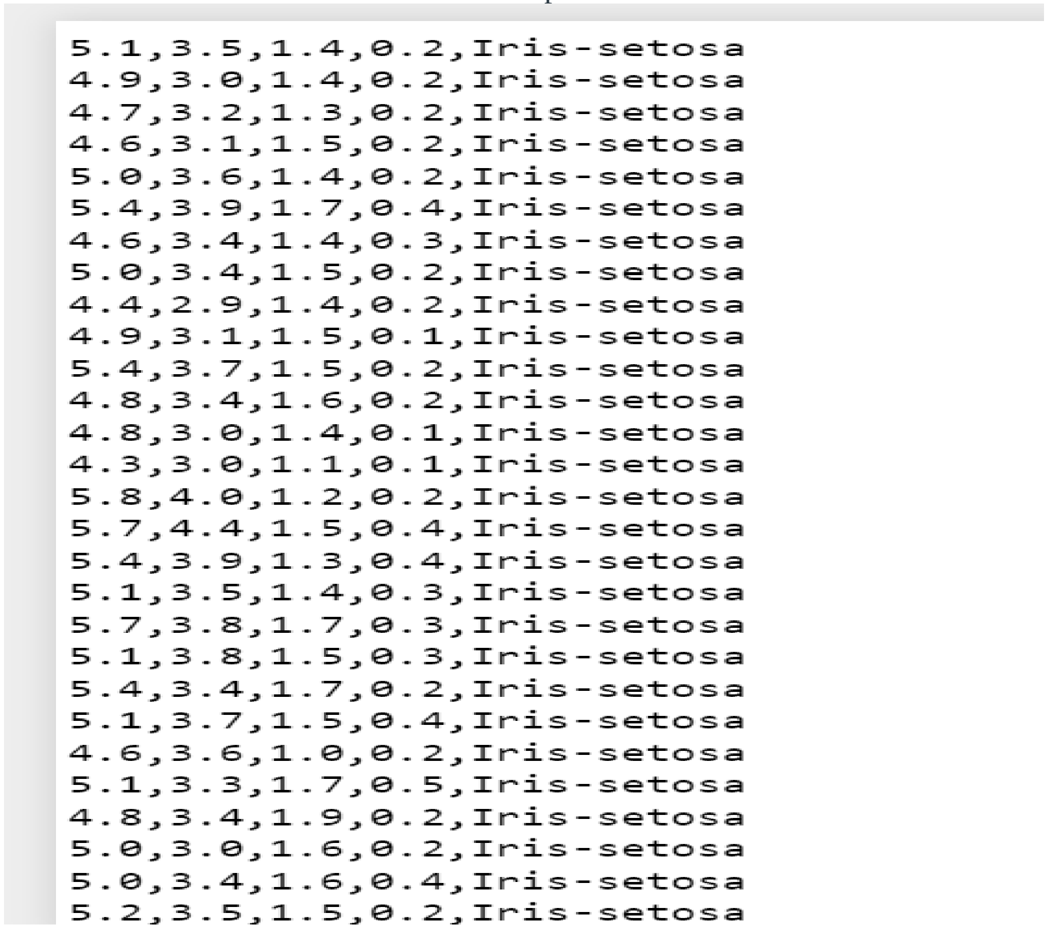
The K-means algorithm aims at minimizing the sum of the squared distance between each data point and the cluster centroid that it is assigned. The algorithm analyzes the distance between every data point and every centroid at each iteration, then as signs every point to the nearest centroid. On the basis of the mean of the data points in each cluster it then recalculates the centroids.

Problem:

The project's objective is to apply the K-means Clustering to the Iris dataset, assess the findings and the report back.

Data:

For this we used the data (http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data). Given below is the screenshot of the sample data that is used.



```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
5.4,3.7,1.5,0.2,Iris-setosa
4.8,3.4,1.6,0.2,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
5.4,3.4,1.7,0.2,Iris-setosa
5.1,3.7,1.5,0.4,Iris-setosa
4.6,3.6,1.0,0.2,Iris-setosa
5.1,3.3,1.7,0.5,Iris-setosa
4.8,3.4,1.9,0.2,Iris-setosa
5.0,3.0,1.6,0.2,Iris-setosa
5.0,3.4,1.6,0.4,Iris-setosa
5.2,3.5,1.5,0.2,Iris-setosa
```

Steps for the procedure:

Step-1: Select the K-Clusters randomly.

Step-2: Find the distance between each datapoint and each center.

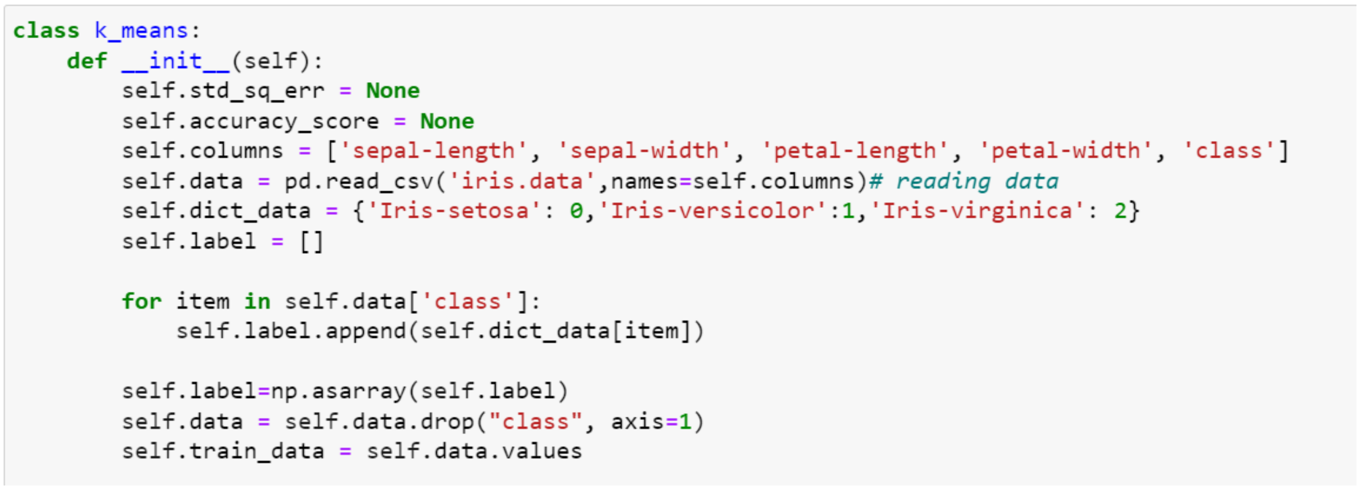
Step-3: Now, find the nearest center for each datapoint with the help of distance value and assign datapoint to that center.

Step-4: Later calculate the new center for each cluster by calculating the average of all datapoints that belong to the same cluster.

Step-5: Repeat Steps 2,3 and 4 until there is no change in their centers.

Problem development:

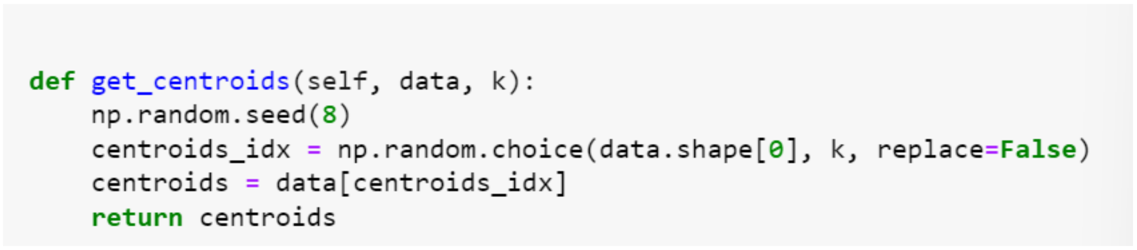
Reading the data:



```python
class k_means:
    def __init__(self):
        self.std_sq_err = None
        self.accuracy_score = None
        self.columns = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
        self.data = pd.read_csv('iris.data',names=self.columns)# reading data
        self.dict_data = {'Iris-setosa': 0,'Iris-versicolor':1,'Iris-virginica': 2}
        self.label = []

        for item in self.data['class']:
            self.label.append(self.dict_data[item])

        self.label=np.asarray(self.label)
        self.data = self.data.drop("class", axis=1)
        self.train_data = self.data.values
```

With the help of Pandas, the data is read from the iris.data file that is uploaded and eliminated the class labels, also converted into numpy array. The panda library is also used to manipulate the data and analyze the tasks.

Randomly selecting k centers:

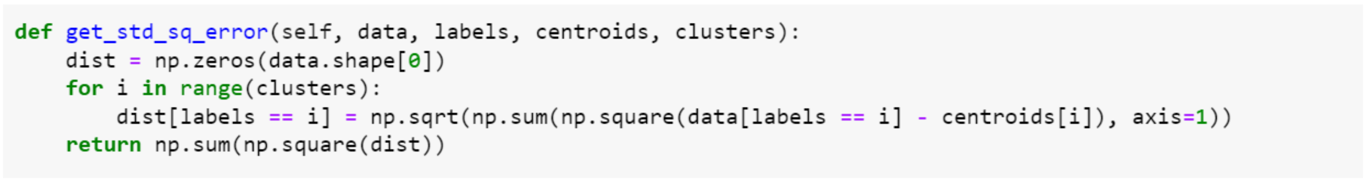


```python
def get_centroids(self, data, k):
    np.random.seed(8)
    centroids_idx = np.random.choice(data.shape[0], k, replace=False)
    centroids = data[centroids_idx]
    return centroids
```

Initial centers are generated using the above function.

Function to calculate Standard Square Error:

To calculate the Standard Square Error (SSE), the below function is used as it takes two inputs: the actual values and predicted values.



```python
def get_std_sq_error(self, data, labels, centroids, clusters):
    dist = np.zeros(data.shape[0])
    for i in range(clusters):
        dist[labels == i] = np.sqrt(np.sum(np.square(data[labels == i] - centroids[i]), axis=1))
    return np.sum(np.square(dist))
```

The function below takes two inputs: **data** and **centroids** and returns the cluster labels for each data point based on their distance to the nearest centroid.

```
def prediction(self, data, centroids):
    distances = np.zeros([len(data), len(centroids)])
    distances = np.sqrt(np.sum(np.square(data[:, np.newaxis, :] - centroids), axis=2))
    labels = np.argmin(distances, axis=1)
    return labels
```

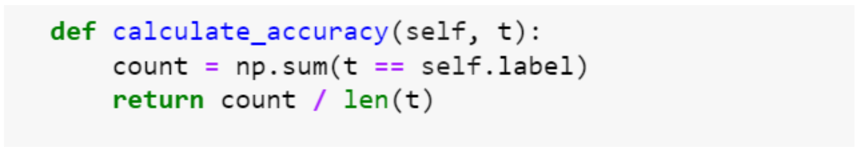
Distance Measuring functions from centroid to datapoints:

```
def cosine_function(self,a,b):
    return distance.cosine(a,b)


def euclidean_function(self,datapoint,cent):
    return sqrt(np.sum((datapoint-cent)*(datapoint-cent)))


def minkowski_function(self,a,b):
    return distance.minkowski(a, b, 6)


def manhattan_function(self,point1, point2):
    return np.sum(np.absolute(point1 - point2))
```

K -means function implementation Algorithm:

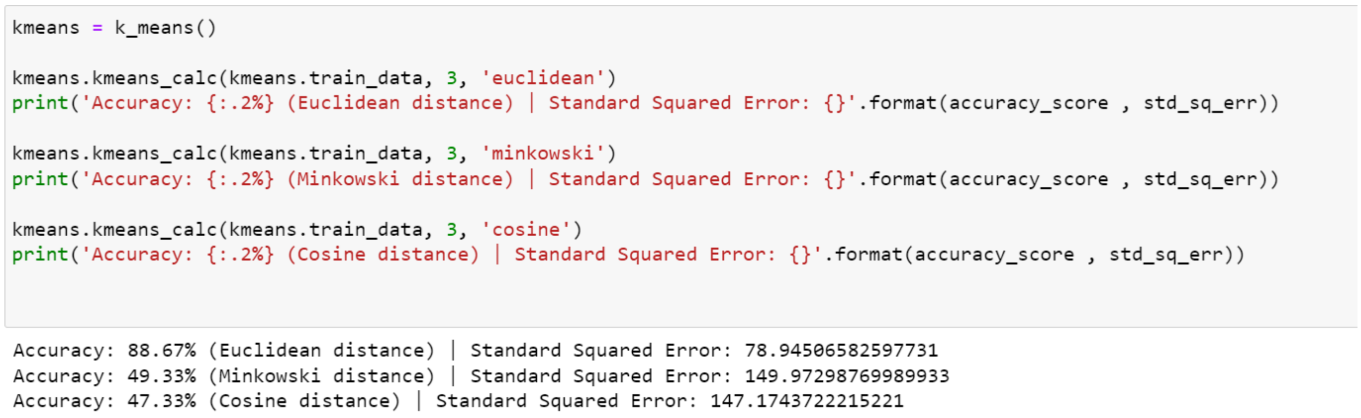
```
def kmeans_calc(self, data, clusters=3, metric='euclidean', iterations=20):
    r, c = data.shape
    global std_sq_err
    global accuracy_score
    old_centroids = self.get_centroids(data, clusters)
    groups = {}

    for _ in range(iterations):
        distance = np.zeros([r, clusters])

        for i in range(r):
            if metric == 'euclidean':
                distance[i] = [self.euclidean_function(data[i], centroid) for centroid in old_centroids]
            elif metric == 'manhattan':
                distance[i] = [self.manhattan_function(data[i], centroid) for centroid in old_centroids]
            elif metric == 'minkowski':
                distance[i] = [self.minkowski_function(data[i], centroid) for centroid in old_centroids]
            elif metric == 'cosine':
                distance[i] = [self.cosine_function(data[i], centroid) for centroid in old_centroids]

            classes = np.argmin(distance, axis=1)

        new_centroids = [np.mean(data[classes == i], axis=0) for i in range(clusters)]
        old_centroids = new_centroids.copy()

    targets = self.prediction(data, old_centroids)
    accuracy_score = self.calculate_accuracy(targets)
    std_sq_err = self.get_std_sq_error(data, targets, old_centroids, clusters)
```

Calculating the accuracy of the cluster assignment:



```python
def calculate_accuracy(self, t):
    count = np.sum(t == self.label)
    return count / len(t)
```

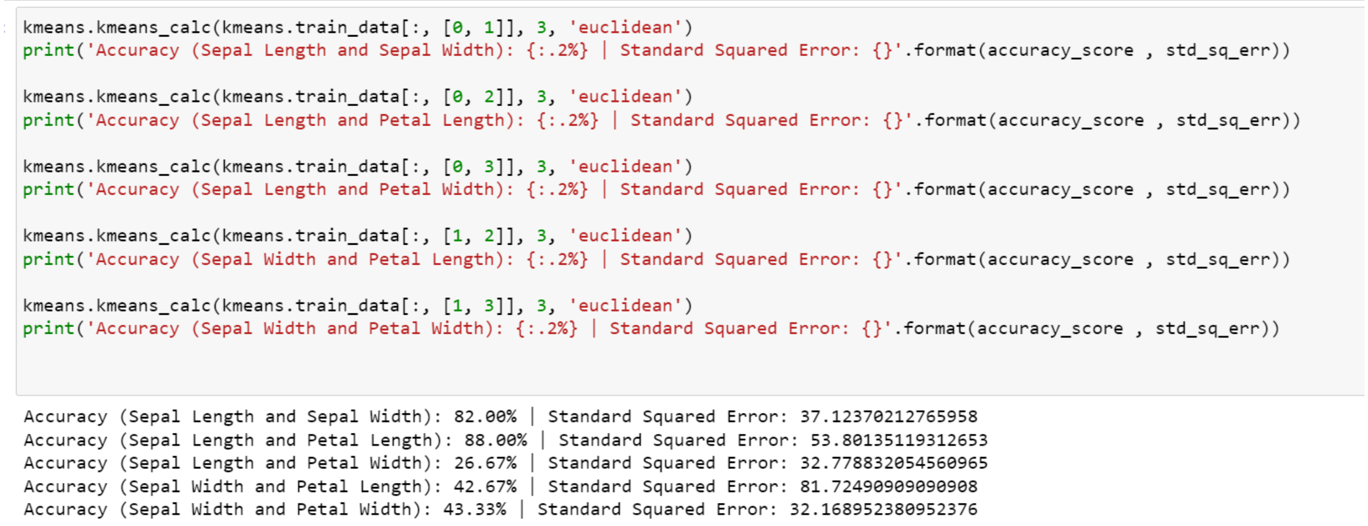
Accuracy for different distance functions:



```python
kmeans = k_means()

kmeans.kmeans_calc(kmeans.train_data, 3, 'euclidean')
print('Accuracy: {:.2%} (Euclidean distance) | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data, 3, 'minkowski')
print('Accuracy: {:.2%} (Minkowski distance) | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data, 3, 'cosine')
print('Accuracy: {:.2%} (Cosine distance) | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))
```

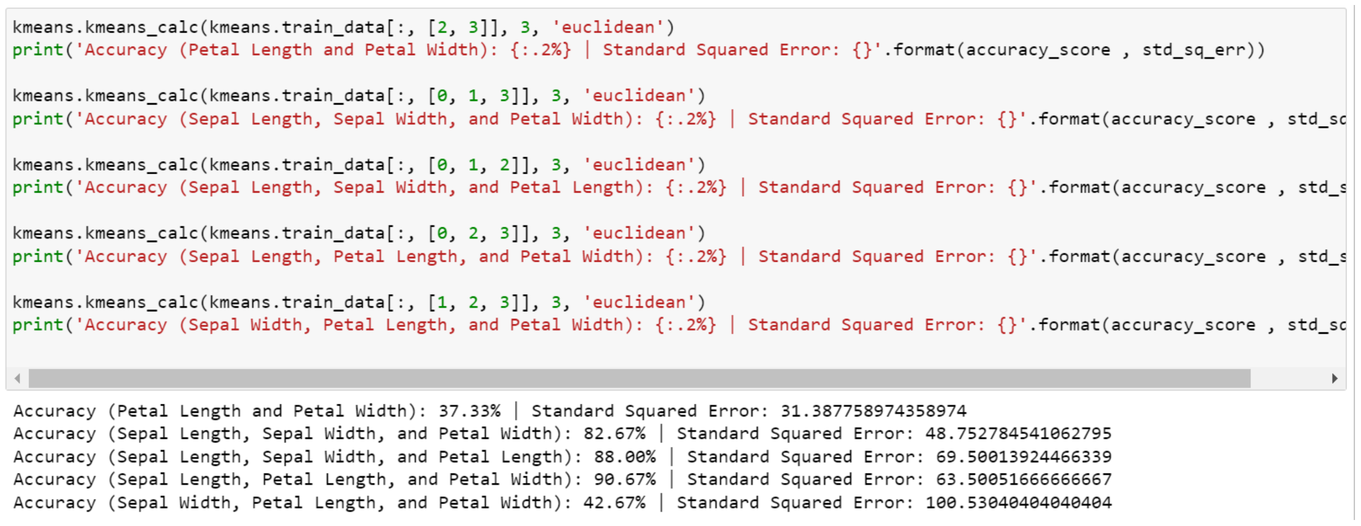
```
Accuracy: 88.67% (Euclidean distance) | Standard Squared Error: 78.94506582597731
Accuracy: 49.33% (Minkowski distance) | Standard Squared Error: 149.97298769989933
Accuracy: 47.33% (Cosine distance) | Standard Squared Error: 147.1743722215221
```

Instead of taking all features, I tried different combination of features and accuracy is as following:



```python
kmeans.kmeans_calc(kmeans.train_data[:, [0, 1]], 3, 'euclidean')
print('Accuracy (Sepal Length and Sepal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data[:, [0, 2]], 3, 'euclidean')
print('Accuracy (Sepal Length and Petal Length): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data[:, [0, 3]], 3, 'euclidean')
print('Accuracy (Sepal Length and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data[:, [1, 2]], 3, 'euclidean')
print('Accuracy (Sepal Width and Petal Length): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data[:, [1, 3]], 3, 'euclidean')
print('Accuracy (Sepal Width and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))
```

```
Accuracy (Sepal Length and Sepal Width): 82.00% | Standard Squared Error: 37.12370212765958
Accuracy (Sepal Length and Petal Length): 88.00% | Standard Squared Error: 53.80135119312653
Accuracy (Sepal Length and Petal Width): 26.67% | Standard Squared Error: 32.778832054560965
Accuracy (Sepal Width and Petal Length): 42.67% | Standard Squared Error: 81.72490909090908
Accuracy (Sepal Width and Petal Width): 43.33% | Standard Squared Error: 32.168952380952376
```



```
kmeans.kmeans_calc(kmeans.train_data[:, [2, 3]], 3, 'euclidean')
print('Accuracy (Petal Length and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sq_err))

kmeans.kmeans_calc(kmeans.train_data[:, [0, 1, 3]], 3, 'euclidean')
print('Accuracy (Sepal Length, Sepal Width, and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sc

kmeans.kmeans_calc(kmeans.train_data[:, [0, 1, 2]], 3, 'euclidean')
print('Accuracy (Sepal Length, Sepal Width, and Petal Length): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_s

kmeans.kmeans_calc(kmeans.train_data[:, [0, 2, 3]], 3, 'euclidean')
print('Accuracy (Sepal Length, Petal Length, and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_s

kmeans.kmeans_calc(kmeans.train_data[:, [1, 2, 3]], 3, 'euclidean')
print('Accuracy (Sepal Width, Petal Length, and Petal Width): {:.2%} | Standard Squared Error: {}'.format(accuracy_score , std_sc
```

```
Accuracy (Petal Length and Petal Width): 37.33% | Standard Squared Error: 31.387758974358974
Accuracy (Sepal Length, Sepal Width, and Petal Width): 82.67% | Standard Squared Error: 48.752784541062795
Accuracy (Sepal Length, Sepal Width, and Petal Length): 88.00% | Standard Squared Error: 69.50013924466339
Accuracy (Sepal Length, Petal Length, and Petal Width): 90.67% | Standard Squared Error: 63.50051666666667
Accuracy (Sepal Width, Petal Length, and Petal Width): 42.67% | Standard Squared Error: 100.53040404040404
```

Between various combinations of input characteristics. A maximum accuracy of 90.67% is obtained using sepal length, sepal width, and petal width as input features. A minimum accuracy of 26.67% is also obtained for sepal length and petal breadth.

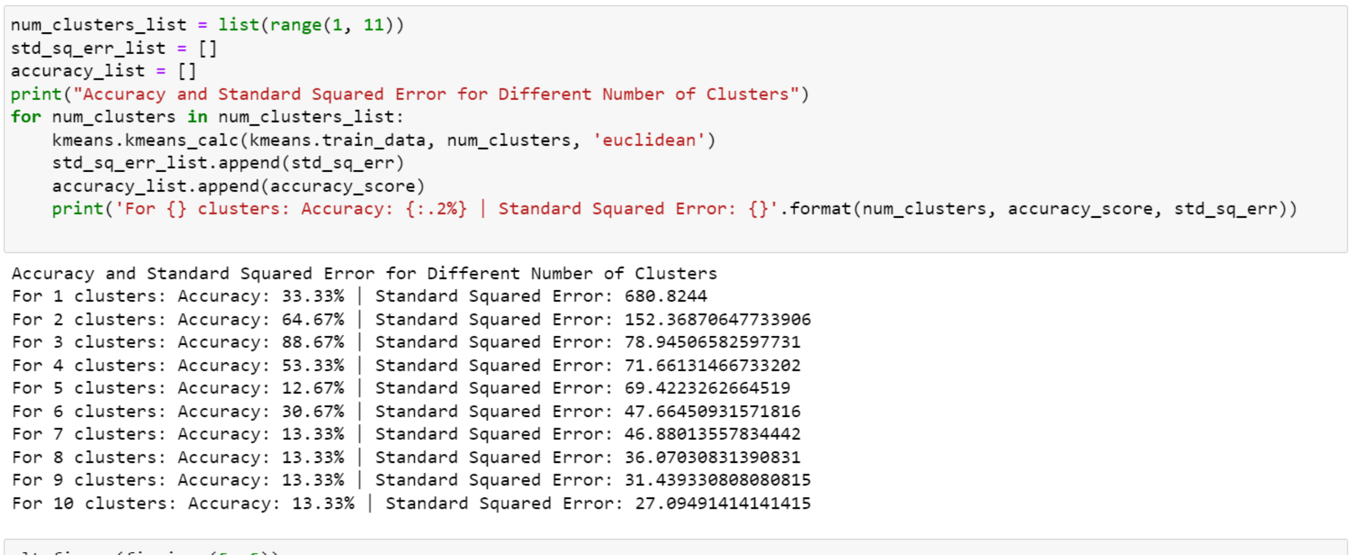
Analysis of Results:

Assumed that the **K-Means** object has already been initialized and fitted to the iris dataset using the **fit()**

Selecting K-value:
K= number of clusters

I used the elbow-method to find k value for efficient output.

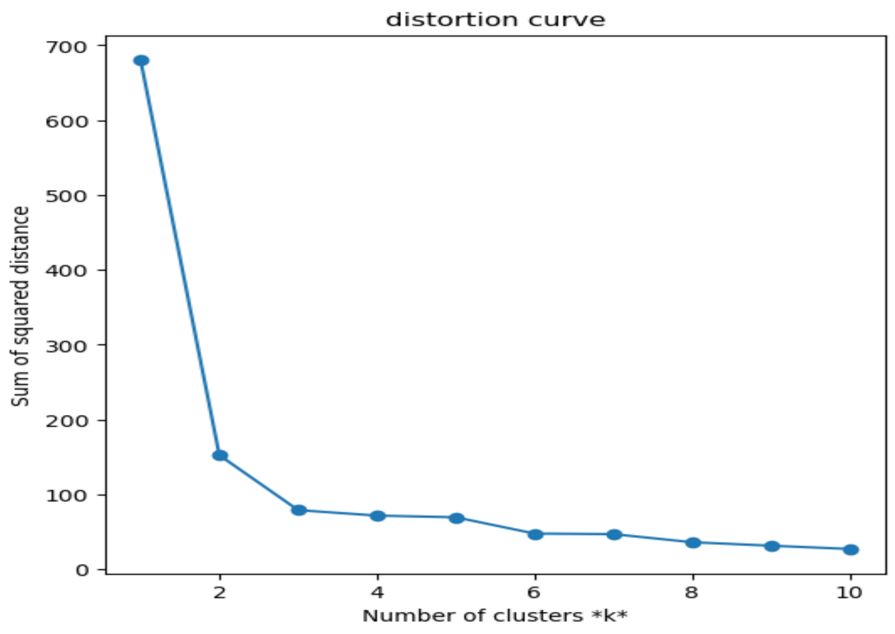
The change in standard square error is relatively small for after k=3. The accuracy utilizing the original labels is 88.67%, the highest for this data when compared to different k values.



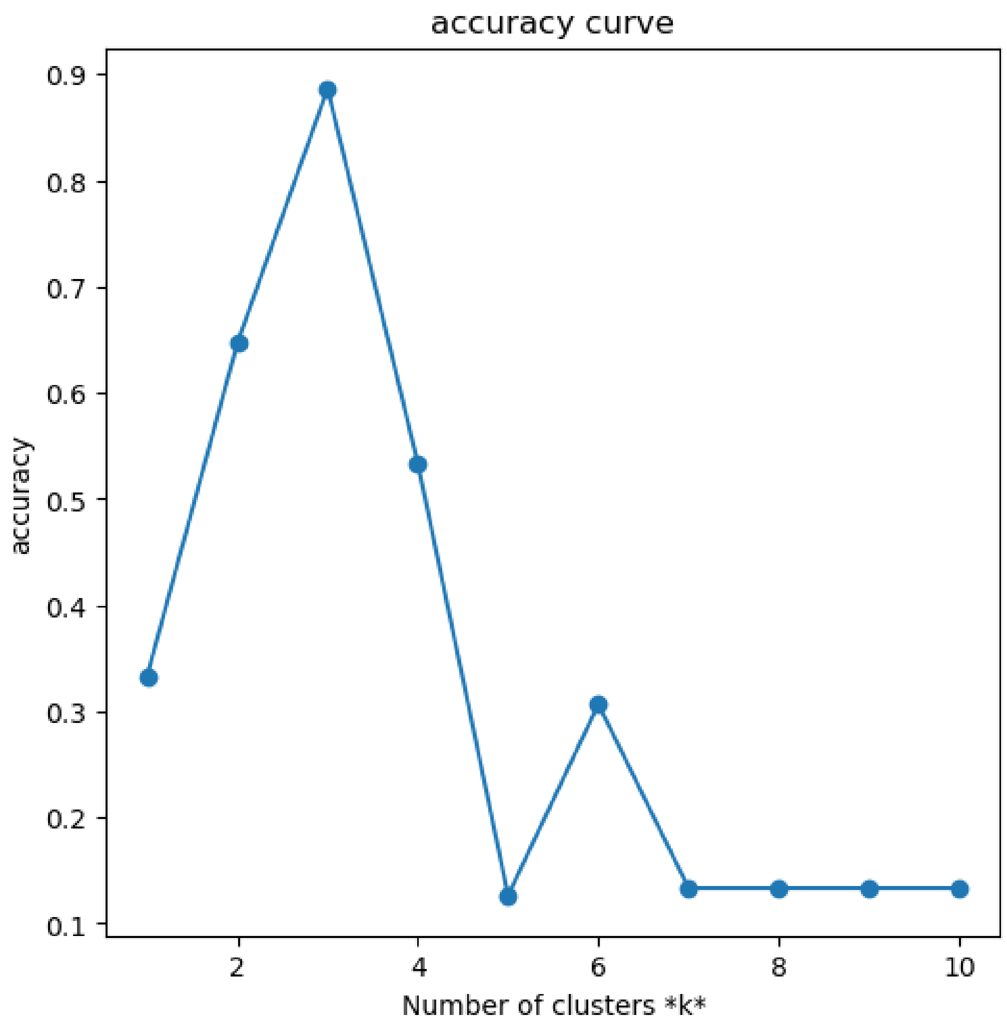
```
num_clusters_list = list(range(1, 11))
std_sq_err_list = []
accuracy_list = []
print("Accuracy and Standard Squared Error for Different Number of Clusters")
for num_clusters in num_clusters_list:
    kmeans.kmeans_calc(kmeans.train_data, num_clusters, 'euclidean')
    std_sq_err_list.append(std_sq_err)
    accuracy_list.append(accuracy_score)
    print('For {} clusters: Accuracy: {:.2%} | Standard Squared Error: {}'.format(num_clusters, accuracy_score, std_sq_err))
```

```
Accuracy and Standard Squared Error for Different Number of Clusters
For 1 clusters: Accuracy: 33.33% | Standard Squared Error: 680.8244
For 2 clusters: Accuracy: 64.67% | Standard Squared Error: 152.36870647733906
For 3 clusters: Accuracy: 88.67% | Standard Squared Error: 78.94506582597731
For 4 clusters: Accuracy: 53.33% | Standard Squared Error: 71.66131466733202
For 5 clusters: Accuracy: 12.67% | Standard Squared Error: 69.4223262664519
For 6 clusters: Accuracy: 30.67% | Standard Squared Error: 47.66450931571816
For 7 clusters: Accuracy: 13.33% | Standard Squared Error: 46.88013557834442
For 8 clusters: Accuracy: 13.33% | Standard Squared Error: 36.07030831390831
For 9 clusters: Accuracy: 13.33% | Standard Squared Error: 31.439330808080815
For 10 clusters: Accuracy: 13.33% | Standard Squared Error: 27.09491414141415
```

At k=3, or 78.945, distortion is greatly reduced. After that, there was a rather small shift in distortion.



Additionally, accuracy is highest at k=3, or 88.67%. In the original data, there are just 3 classifications.



Advantages and Disadvantages:

Advantages:

The following are some advantages of K-Means clustering algorithms −

- It is very easy to understand and implement.
- If we have large number of variables then, K-means would be faster than Hierarchical clustering.
- On re-computation of centroids, an instance can change the cluster.
- Tighter clusters are formed with K-means as compared to Hierarchical clustering.

Disadvantages:

The following are some disadvantages of K-Means clustering algorithms −

- It is a bit difficult to predict the number of clusters i.e. the value of k.
- Output is strongly impacted by initial inputs like number of clusters (value of k).
- Order of data will have strong impact on the final output.
- It is very sensitive to rescaling. If we will rescale our data by means of normalization or standardization, then the output will completely change. Final output.
- It is not good in doing clustering job if the clusters have a complicated geometric shape.

Applications of K-Means Clustering Algorithm

The main goals of cluster analysis are −

- To get a meaningful intuition from the data we are working with.
- Cluster-then-predict where different models will be built for different subgroups.

To fulfill the above-mentioned goals, K-means clustering is performing well enough. It can be used in following applications −

- Market segmentation
- Document Clustering
- Image segmentation
- Image compression
- Customer segmentation
- Analyzing the trend on dynamic data

References:

1. K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks | by Imad Dabbura | Towards Data Science
2. https://www.tutorialspoint.com/machine_learning_with_python/clustering_algorithms_k_means_algorithm.htm
3. https://chat.openai.com/chat
4. K Means Clustering Project | Kaggle