CSE 6363-001-Machine Learning

Fall 2022

Project 3 Report

Instructor's Name: Dr. Dajiang Zhu

Student Name: Siva Srinivasa Sameer Miriyala

Student ID: 1002024871

Given Problem Statement:

A data collection with 150 samples from three different flower classes—"Iris-setosa," "Iris-versicolor," and "Iris-virginica"—is given to us. Choose the amount of clusters you'll utilize and justify your choice. Apply the K-Means Clustering Algorithm to the IRIS-3 class data set to accurately place each data point in the cluster to which it belongs. Utilize the dataset's labels for assessment.

Dataset:

This data has been extracted from (http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data)

	SepalLength	SepalWidth	PetalLength	PetalWidth
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

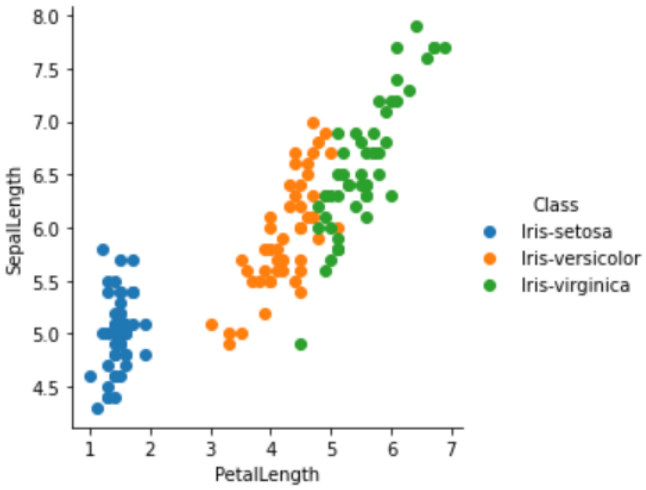
Analyzing the dataset:

To examine the distribution of data points among each class, a histogram plot is used to depict each data point.

1. The total data set has just 3 classes, and there are an equal amount of samples in each class.

2. The following scatter plot is created by taking into account the full data set and their class labels. Red, Green, and Blue are used to distinguish one class from another.

3. The code for the scatter plot of the data set with 150 samples is presented in figure 3 below, along with the scatter plot itself.



Cluster Size and Interpretation:

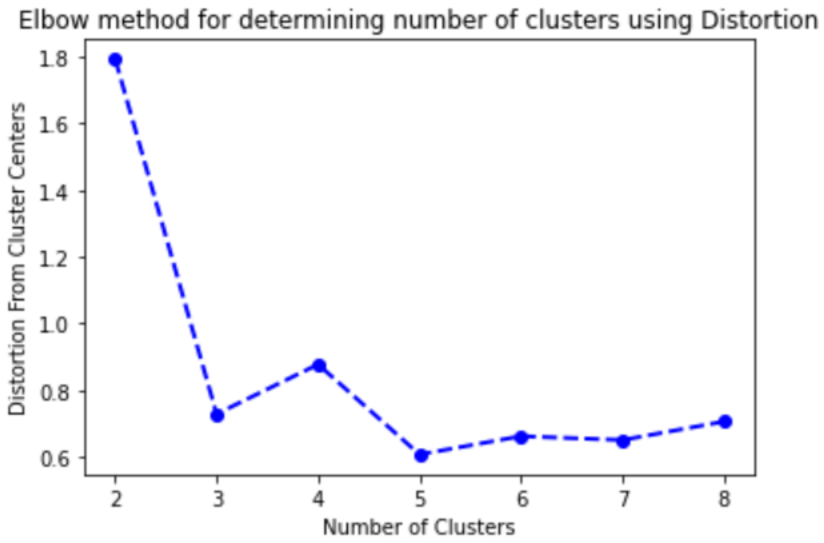
Since there are 3 classes total in the data set, selecting the value of the number of clusters as 3 would be a suitable decision.

Counting clusters using the elbow method:

The variation or distortion is represented as a function of the number of clusters. In addition, the number of clusters is chosen to be at the curve's elbow. The value for the number of clusters as 3 obtained using the Elbow approach is shown in Figure below. The distortion is estimated across a range of values between 1 and 10, as shown in figure. Below Figure shows that the elbow point is at K=3, indicating that 3 clusters were selected to perform the k-means clustering process.

The distortion values for the provided data set for a range of k values are shown in the following Python code.

```
[3]  # Scatter Plot of the features SepalLength and PetalLength
     grid = sns.FacetGrid(iris_df, hue ="Class", height = 4)
     grid.map(plt.scatter, 'PetalLength', 'SepalLength').add_legend()
```



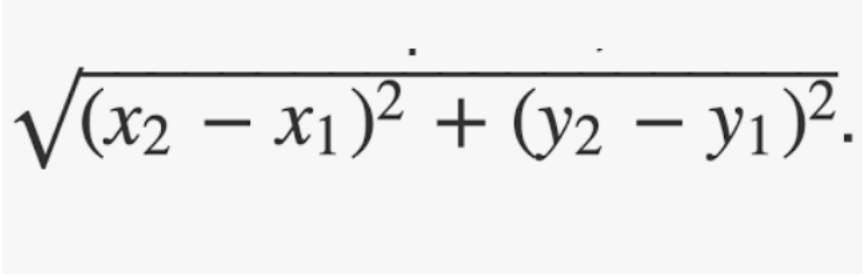
Create Random Clusters at Startup:

The k-means clustering technique is first implemented using random points as centroids or centers. The starting clusters are chosen at random by the Python code. We choose three randomly selected points within the dataset to serve as the first centroids because there are three clusters total.

```
def _init_clusters_(self):
    indices = np.linspace(0, len(self._X)-1, len(self._X), dtype=int)
    self._idx_centers = np.random.choice(indices, size=self.n_clusters)
    self.cluster_centers = self._X[self._idx_centers]
    self._init_data_to_clusters_()
```

Cluster data points:

Assigning each data point to the closest cluster comes next. A data point is considered to belong to a cluster when its Euclidean distance from that cluster is least. Figure 8 shows the following code, which calculates the Euclidean distance between each data point and each cluster and then assigns the data point to the closest cluster it may be. It is calculated by the formula

d = 

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Where d is the Euclidean distance between the points

```
def _init_data_to_clusters_(self):
    clusters_list = [[] for i in range(self.n_clusters)]
    for feature in self._X:
        curr_cluster_index = -1
        min_distance = np.inf
        for idx, center in enumerate(self.cluster_centers):
            curr_distance = calc_distance(feature, center)
            if curr_distance < min_distance:
                min_distance = curr_distance
                curr_cluster_index = idx
        if curr_cluster_index == -1:
            print(f"Cluster couldnt be identified for {feature}")
            return
        clusters_list[curr_cluster_index].append(feature)
    self.clusters = np.array(clusters_list, dtype=object)
    return self
```

Update the clusters:

The updated clusters will have either mean or centroid points of the data. The re-modified clusters will be created.

```
def _update_clusters_(self):
    if self.clusters is None:
        print("Data not loaded into clusters. Call _init_clusters_")
        return
    for i in range(self.n_clusters):
        self.cluster_centers[i] = np.mean(self.clusters[i], axis=0)
    self._init_data_to_clusters_()
```

K-Means:

Almost 100 iterations are performed. And the clusters will be formed from k-means pipeline.

Results:

The most frequent class data points assigned to each cluster are calculated and used to assign labels to each cluster. In this regard, we use information that is grounded in reality.

```
def _get_class_labels_(self):
    list_features = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
    dict_labels = dict(zip(list_features, range(3)))
    cluster_labels = [[] for i in range(self.n_clusters)]
    for feature, label in zip(self._X, self._y):
        min_dist = np.inf
        min_idx = -10
        for i, center in enumerate(self.cluster_centers):
            curr_dist = calc_distance(center, feature)
            if curr_dist < min_dist:
                min_dist = curr_dist
                min_idx = i
        cluster_labels[min_idx].append(dict_labels[label])
    estimated_labels = [np.argmax(np.bincount(i)) for i in cluster_labels]
    self.dict_labels = dict(zip(estimated_labels[::-1], list_features))
```

```
def _init_data_to_clusters_(self):
    clusters_list = [[] for i in range(self.n_clusters)]
    for feature in self._X:
        curr_cluster_index = -1
        min_distance = np.inf
        for idx, center in enumerate(self.cluster_centers):
            curr_distance = calc_distance(feature, center)
            if curr_distance < min_distance:
                min_distance = curr_distance
                curr_cluster_index = idx
        if curr_cluster_index == -1:
            print(f"Cluster couldnt be identified for {feature}")
            return
        clusters_list[curr_cluster_index].append(feature)
    self.clusters = np.array(clusters_list, dtype=object)
    return self
```

Using the label of the closest cluster center, predict the label of each data point.

```
def fit(self, X_data, y_data):
    self._X = X_data
    self._y = y_data
    self._init_clusters_()
    for i in range(self.__max_iter):
        old_cluster_centers = self.cluster_centers.copy()
        self._update_clusters_()
        distance = calc_distance(old_cluster_centers, self.cluster_centers, axis=1)
        if i == 0:
            prev_max_distance = np.max(distance)
            continue
        else:
            curr_max_distance = np.max(distance)
            change_diff = abs(((curr_max_distance - prev_max_distance))/ \
                              np.mean([prev_max_distance, curr_max_distance]))
            prev_max_distance = curr_max_distance
        if change_diff < self.__tol or np.isnan(change_diff):
            break
    self.__flag_fit = True

def predict(self, X_data):
    if not self.__flag_fit:
        print("Model not fit with data yet! run model.fit() before running predict")
    self._get_class_labels_()
    return np.apply_along_axis(self.__predict_data__, axis=1, arr=X_data)
```

Accuracy:

This code will give us the accuracy of the algorithm.

```
# predictions on test data
labels_predicted = model.predict(iris_features)
# Accuracy of KMeans model on testing dataset
accuracy = model.get_accuracy(labels_predicted)
print(f"Accuracy of classifications on testing data : {round(accuracy, 2)}%")
```

```
Accuracy of classifications on testing data : 66.67%
```

How can we improve the accuracy?

We should randomly select the first cluster and then the next cluster will be calculated by the Euclidean distance from the first cluster. This will provide the other next point that will belong to the another.

The data point with the greatest distance from both clusters will be chosen as the third cluster.

```
def plot_clusters(self, title=None):
    if not self.__flag_fit:
        print("Model not fit with data yet! run model.fit() before running plot")
    if title is None:
        title = "Cluster Plot"
    plt = self._init_plot(title)
    for i, class_name in enumerate(classname_to_label.keys()):
        plt.scatter(features[labels_count==i,2],features[labels_count==i,3],c=colors[i,None],alpha=1.0,s=20,lw=0,label=class_name)
```

Proximity Measure: Although Euclidean distance is used in this instance, it can be customized for the situation.

Conclusion: The aforementioned initialization strategy significantly enhances run-time convergence. However, the computational cost of the above initialization method is high.