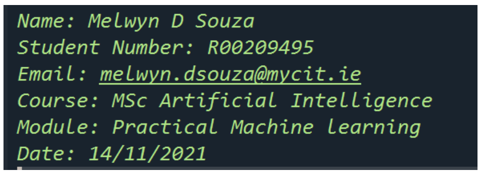
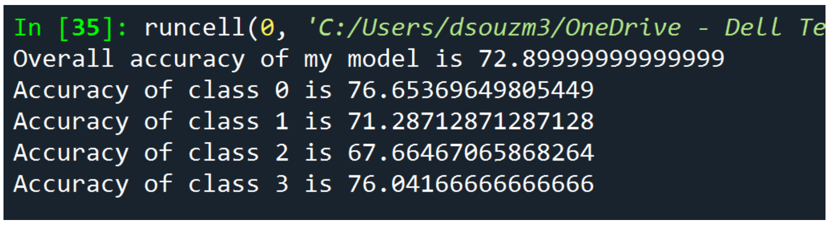
Practical Machine Learning



```
Name: Melwyn D Souza
Student Number: R00209495
Email: melwyn.dsouza@mycit.ie
Course: MSc Artificial Intelligence
Module: Practical Machine Learning
Date: 14/11/2021
```

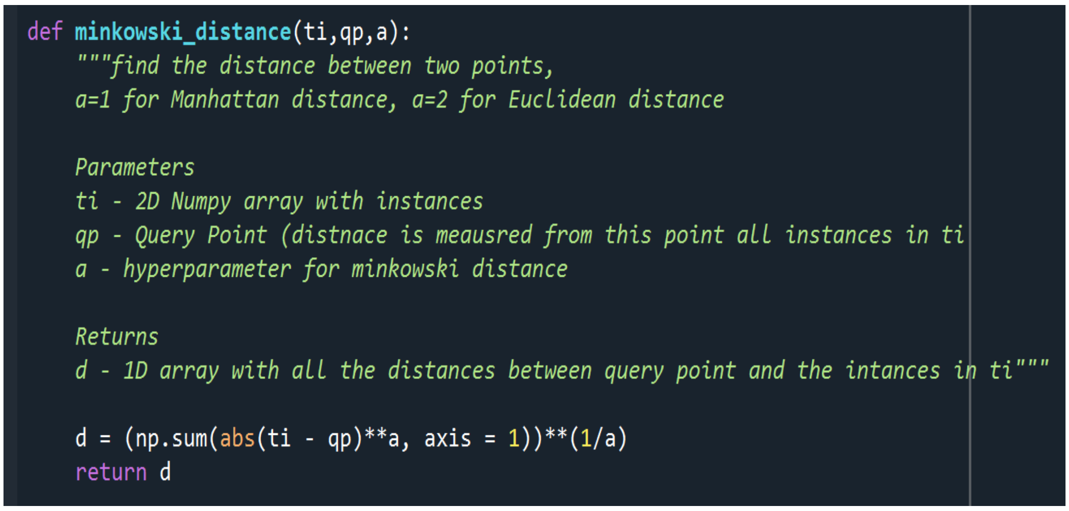
1. PART 1A

a. Accuracy of my model with a = 1 and k =3



```
In [35]: runcell(0, 'C:/Users/dsouzm3/OneDrive - Dell Te
Overall accuracy of my model is 72.89999999999999
Accuracy of class 0 is 76.65369649805449
Accuracy of class 1 is 71.28712871287128
Accuracy of class 2 is 67.66467065868264
Accuracy of class 3 is 76.04166666666666
```

b. Minkowski Distance



```python
def minkowski_distance(ti,qp,a):
    """find the distance between two points,
    a=1 for Manhattan distance, a=2 for Euclidean distance

    Parameters
    ti - 2D Numpy array with instances
    qp - Query Point (distnace is meausred from this point all instances in ti
    a - hyperparameter for minkowski distance

    Returns
    d - 1D array with all the distances between query point and the intances in ti"""

    d = (np.sum(abs(ti - qp)**a, axis = 1))**(1/a)
    return d
```

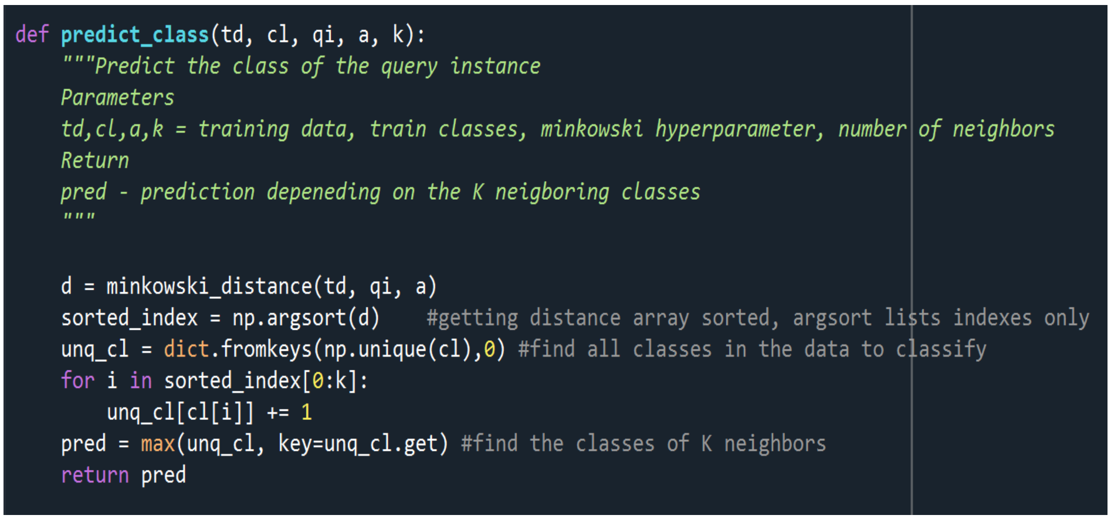
The function above is minkowski_distance(), it is used to calculate distance between two points using the equation shown below

$$d(p,\ q) = \ (\sum_{i=1}^{n} |p_i \ - \ q_i|^a)^{\frac{1}{a}}$$

The equation above is used to measure distance between point ‘p’ and ‘q’, when a=1, we are calculating Manhattan distance, and when a=2, Euclidean distance is calculated

The function above uses numpy library, broadcasting calculations across all elements

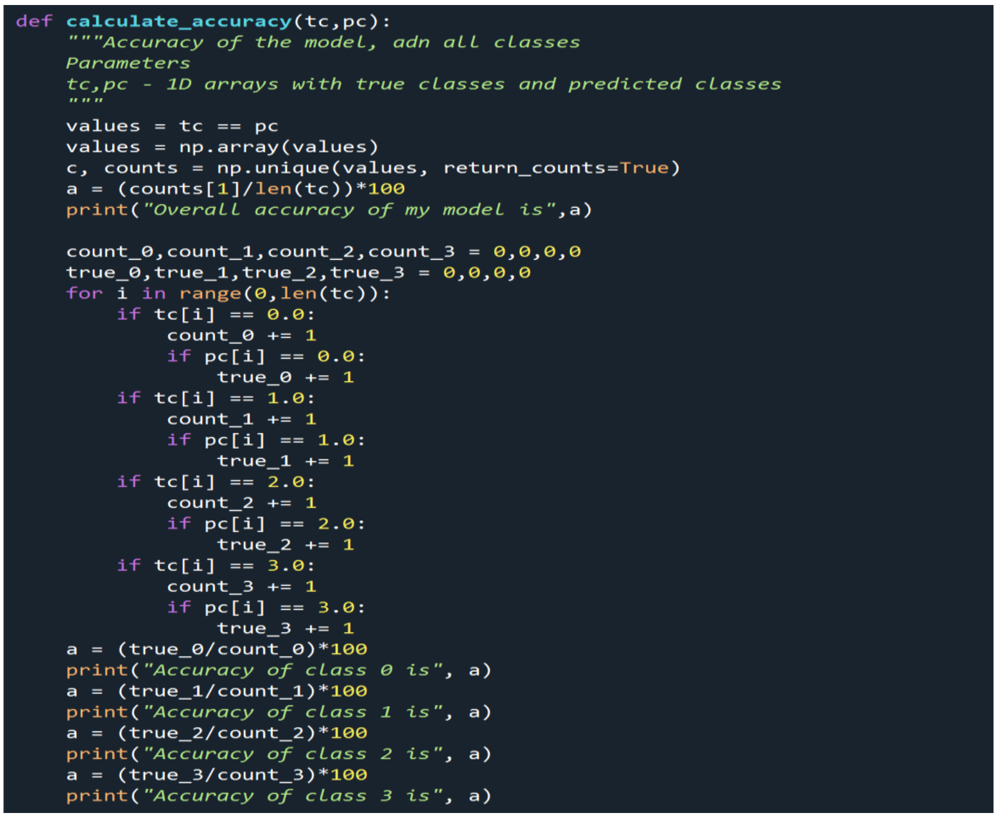
c. Predicting the class



```python
def predict_class(td, cl, qi, a, k):
    """Predict the class of the query instance
    Parameters
    td,cl,a,k = training data, train classes, minkowski hyperparameter, number of neighbors
    Return
    pred - prediction depeneding on the K neigboring classes
    """

    d = minkowski_distance(td, qi, a)
    sorted_index = np.argsort(d)    #getting distance array sorted, argsort lists indexes only
    unq_cl = dict.fromkeys(np.unique(cl),0) #find all classes in the data to classify
    for i in sorted_index[0:k]:
        unq_cl[cl[i]] += 1
    pred = max(unq_cl, key=unq_cl.get) #find the classes of K neighbors
    return pred
```

The predict class function has parameters as mentioned in the comments above,
Step1: Get the minkowski distance between the query point (qp) and training data (td)
Step2: Get the array sorted by the distances, in ascending order (argsort returns indexes which can be used in extracting feature instances etc)
Step3: Find all the classes in the dataset
Step4: Find the first k neighbours and get the classes they are assigned to
Step5: Assign the most prominent class amongst the k neighbors

d. Calculate accuracy



```python
def calculate_accuracy(tc,pc):
    """Accuracy of the model, adn all classes
    Parameters
    tc,pc - 1D arrays with true classes and predicted classes
    """
    values = tc == pc
    values = np.array(values)
    c, counts = np.unique(values, return_counts=True)
    a = (counts[1]/len(tc))*100
    print("Overall accuracy of my model is",a)

    count_0,count_1,count_2,count_3 = 0,0,0,0
    true_0,true_1,true_2,true_3 = 0,0,0,0
    for i in range(0,len(tc)):
        if tc[i] == 0.0:
            count_0 += 1
            if pc[i] == 0.0:
                true_0 += 1
        if tc[i] == 1.0:
            count_1 += 1
            if pc[i] == 1.0:
                true_1 += 1
        if tc[i] == 2.0:
            count_2 += 1
            if pc[i] == 2.0:
                true_2 += 1
        if tc[i] == 3.0:
            count_3 += 1
            if pc[i] == 3.0:
                true_3 += 1
    a = (true_0/count_0)*100
    print("Accuracy of class 0 is", a)
    a = (true_1/count_1)*100
    print("Accuracy of class 1 is", a)
    a = (true_2/count_2)*100
    print("Accuracy of class 2 is", a)
    a = (true_3/count_3)*100
    print("Accuracy of class 3 is", a)
```

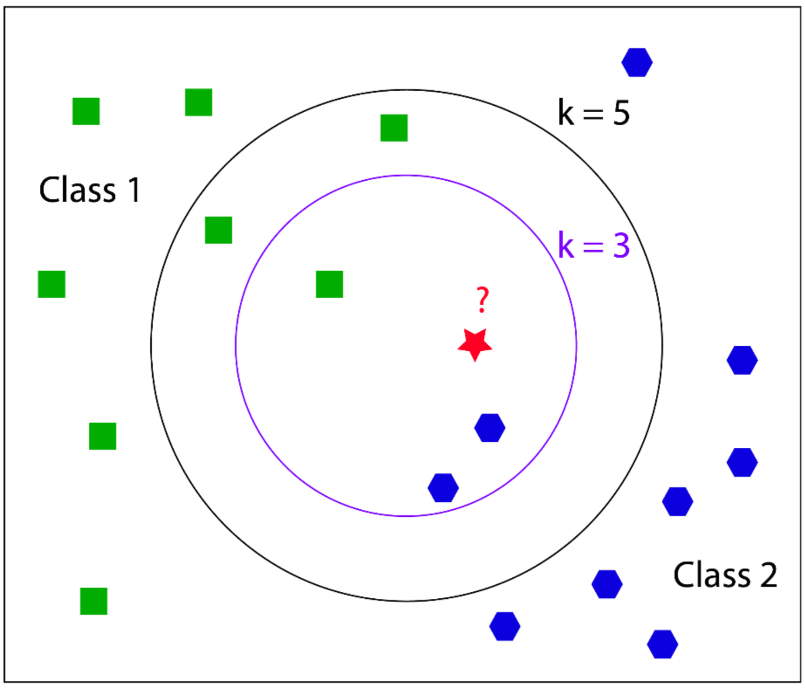
In the above function, the parameters 'tc' True Class 1D array and 'pc' Predicted Class 1D array are compared with each other by using "==" operator, which returns a Boolean 1D array of same length as pc or tc, this helps in calculating the accuracy

To get the accuracy of each class, a loop is used which finds the number of true class vs predicted class, it prints all accuracies in the end of the loop

2. PART 1B

a. *"By default, a k-NN algorithm will weigh the contribution of each feature equally when using standard Euclidean distance"*

Assigning the uniform weights do increase the accuracy of the spatial algorithms such as KNNs, but when there are noisy data points present, assigning uniform weights might drastically decrease the accuracy of the algorithm. One of the major drawbacks of KNN is, if value 'k' is too small, KNN will highly depend on the outliers and when the value of 'k' is too large it will then include too many feature instances from the other class, so the prediction will be incorrect. This is because the KNN algorithm by default uses standard Euclidean distance and the model considers all its neighbours equal, which means all data points have equal votes on the classification of the target data point class [3]

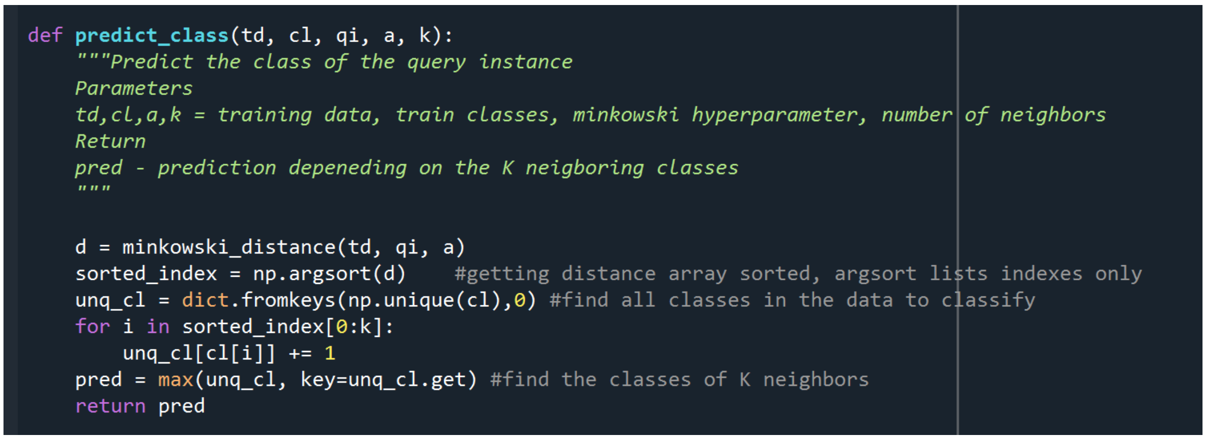


As an example, the picture above shows two circles, one is for k = 5 and other k = 3, when k = 3 is used (purple circle) to train the KNN model, the prediction for k=3 for the red star query point will be blue, the class assigned will be "class 2 / blue" since 2 nearest neighbours, blue hexagons are assigned to class 2 / blue

Now, if the same query point is used for prediction using 5 nearest neighbours (k = 5), the red star will be assigned class 1 / red since 3 of its nearest neighbours, green boxes are assigned to class1/red [1]

b. One technique which can be used to give more importance to the nearest feature points rather than the feature instances which are far away, the farther away the points are, they should have less impact on the prediction of the query point classes, there are several techniques to improve the performance, one of it is finding the inverse distance of all points from the query point and then using that as a voting system, so the nearest neighbours which are really close to the query point will have more value on the prediction even if they are less in number when compared to other total neighbours which are far away from the test query point

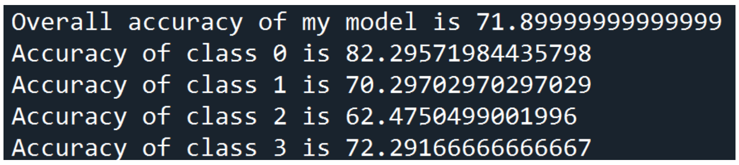
Below is the comparison of my model, the first function is the default KNN using standard Minkowski distance with a=1 (Manhattan distance) without using inverse distance



```python
def predict_class(td, cl, qi, a, k):
    """Predict the class of the query instance
    Parameters
    td,cl,a,k = training data, train classes, minkowski hyperparameter, number of neighbors
    Return
    pred - prediction depeneding on the K neigboring classes
    """

    d = minkowski_distance(td, qi, a)
    sorted_index = np.argsort(d)    #getting distance array sorted, argsort lists indexes only
    unq_cl = dict.fromkeys(np.unique(cl),0) #find all classes in the data to classify
    for i in sorted_index[0:k]:
        unq_cl[cl[i]] += 1
    pred = max(unq_cl, key=unq_cl.get) #find the classes of K neighbors
    return pred
```

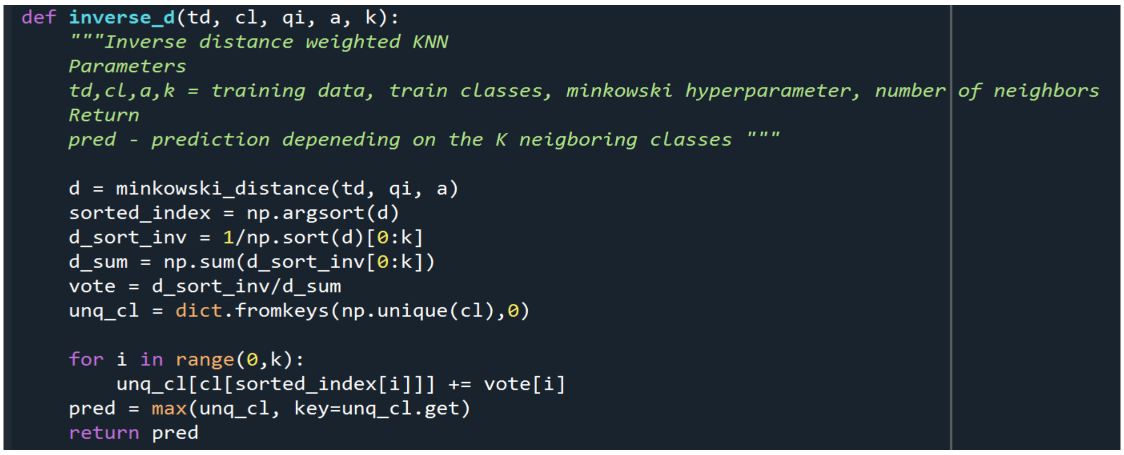
The prediction is solely depending on the distance from the query point to all other points, even when the points are far away and belong to different class

The model accuracy is shown below

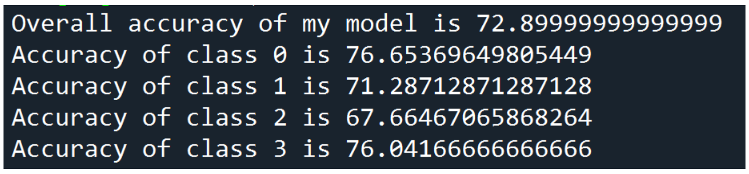


```
Overall accuracy of my model is 71.89999999999999
Accuracy of class 0 is 82.29571984435798
Accuracy of class 1 is 70.29702970297029
Accuracy of class 2 is 62.4750499001996
Accuracy of class 3 is 72.29166666666667
```

When inverse distance is used as a voting system to predict the class for our test instance, the closest neighbours will have higher probability compared to farther instances, the function below is implemented and the output accuracy improvement is as shown below [2]



```python
def inverse_d(td, cl, qi, a, k):
    """Inverse distance weighted KNN
    Parameters
    td,cl,a,k = training data, train classes, minkowski hyperparameter, number of neighbors
    Return
    pred - prediction depeneding on the K neigboring classes """

    d = minkowski_distance(td, qi, a)
    sorted_index = np.argsort(d)
    d_sort_inv = 1/np.sort(d)[0:k]
    d_sum = np.sum(d_sort_inv[0:k])
    vote = d_sort_inv/d_sum
    unq_cl = dict.fromkeys(np.unique(cl),0)

    for i in range(0,k):
        unq_cl[cl[sorted_index[i]]] += vote[i]
    pred = max(unq_cl, key=unq_cl.get)
    return pred
```



```
Overall accuracy of my model is 72.89999999999999
Accuracy of class 0 is 76.65369649805449
Accuracy of class 1 is 71.28712871287128
Accuracy of class 2 is 67.66467065868264
Accuracy of class 3 is 76.04166666666666
```

In the distance weighed KNN, the selection process for neighbours is done using different distance metrics such as Euclidean, Minkowski, Manhattan, Mahalanobis etc, most of the distance metrics are scale variant, they are highly influenced by the scale of the features in the vector, hence the value of distance changes drastically with the scale of each feature in the feature space [3], The D-KNNs are also impacted by imbalanced datasets, curse of dimensionality and irrelevant features

As an example, if we are trying to classify a target data point as A or B, and the training data points have 10,000 data points of class A and 2000 data points as class B, then the model is likely to predict the target data point class as A, this is the drawback of KNN by imbalanced datasets

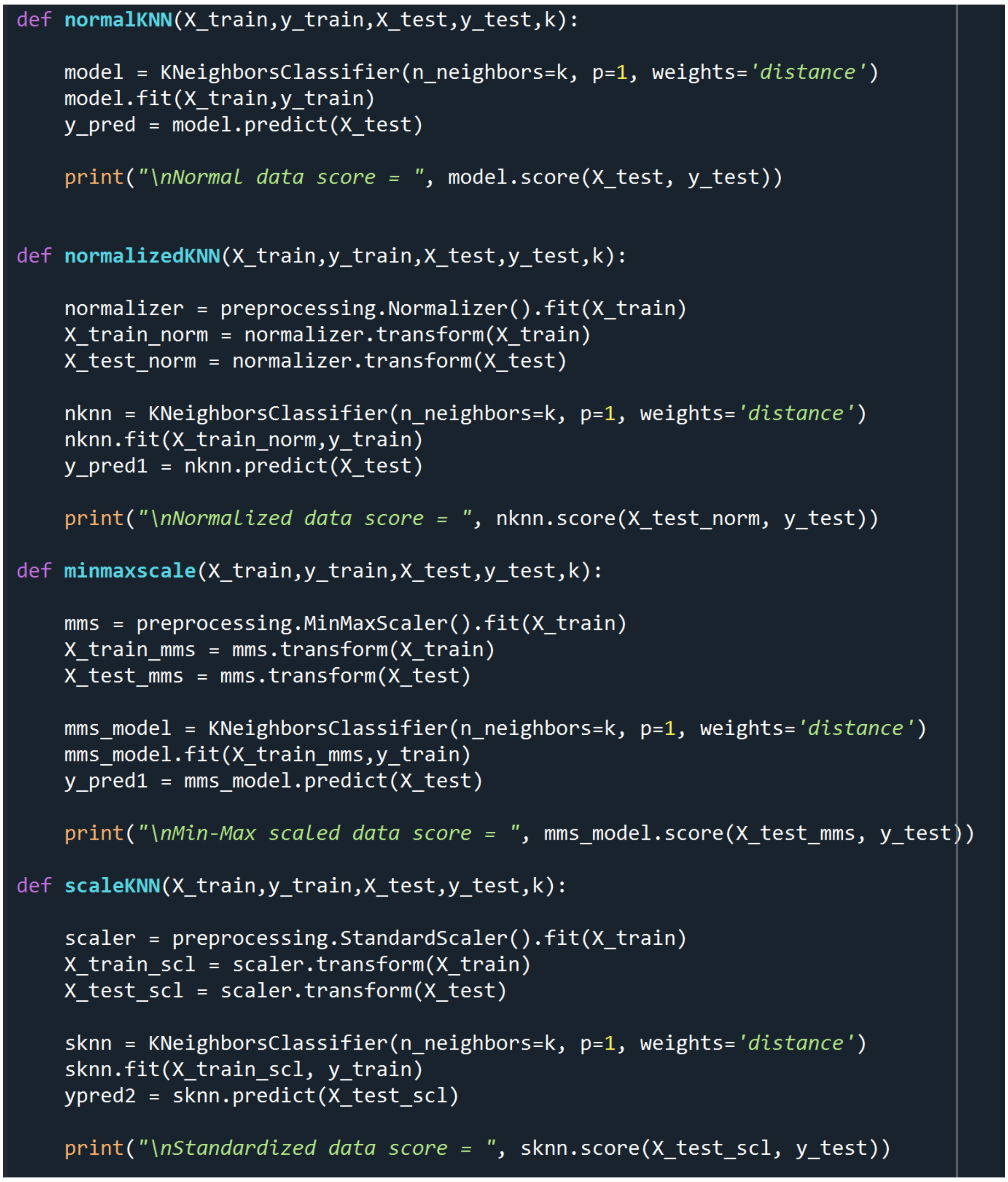
This issue can be tackled by finding the total number of unique classes present in the training data, and then making sure that all classes are balanced across the training data

Curse of dimensionality is when the KNN has to deal with high-dimensional data sets, the issue here is, as the number of dimensions increase, the features in the data set increase, which in turn increases the distance between the feature vectors. Hence, the density is decreased in the feature space and with low density, it is difficult to derive meaningful predictions using our instance based, lazy learner, ie KNN [4]

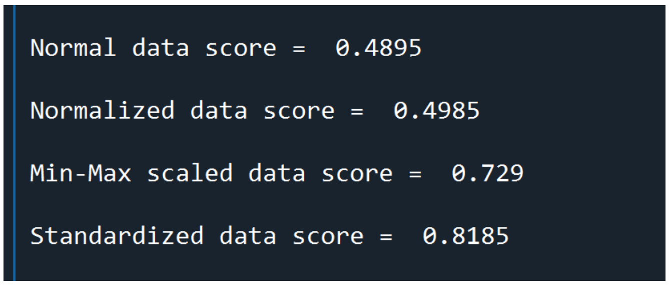
This issue can be tackled by adding more data sets in the feature space to increase the density of the space, this is not the most efficient way since the KNN is a lazy learner, the computationally time will drastically increase with denser feature space. The other option is to remove some dimensions which have minimal impact on the performance of the KNN, this is called dimensionality reduction which is achieved by feature selection

Different types of feature selections are univariate feature selection, low variance features extract, recursive feature elimination

Now we will see how different scaling techniques impact the overall model performance, python libraries are used to develop, test and generate the results



```python
def normalKNN(X_train,y_train,X_test,y_test,k):

    model = KNeighborsClassifier(n_neighbors=k, p=1, weights='distance')
    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)

    print("\nNormal data score = ", model.score(X_test, y_test))


def normalizedKNN(X_train,y_train,X_test,y_test,k):

    normalizer = preprocessing.Normalizer().fit(X_train)
    X_train_norm = normalizer.transform(X_train)
    X_test_norm = normalizer.transform(X_test)

    nknn = KNeighborsClassifier(n_neighbors=k, p=1, weights='distance')
    nknn.fit(X_train_norm,y_train)
    y_pred1 = nknn.predict(X_test)

    print("\nNormalized data score = ", nknn.score(X_test_norm, y_test))

def minmaxscale(X_train,y_train,X_test,y_test,k):

    mms = preprocessing.MinMaxScaler().fit(X_train)
    X_train_mms = mms.transform(X_train)
    X_test_mms = mms.transform(X_test)

    mms_model = KNeighborsClassifier(n_neighbors=k, p=1, weights='distance')
    mms_model.fit(X_train_mms,y_train)
    y_pred1 = mms_model.predict(X_test)

    print("\nMin-Max scaled data score = ", mms_model.score(X_test_mms, y_test))

def scaleKNN(X_train,y_train,X_test,y_test,k):

    scaler = preprocessing.StandardScaler().fit(X_train)
    X_train_scl = scaler.transform(X_train)
    X_test_scl = scaler.transform(X_test)

    sknn = KNeighborsClassifier(n_neighbors=k, p=1, weights='distance')
    sknn.fit(X_train_scl, y_train)
    ypred2 = sknn.predict(X_test_scl)

    print("\nStandardized data score = ", sknn.score(X_test_scl, y_test))
```



```
Normal data score =  0.4895

Normalized data score =  0.4985

Min-Max scaled data score =  0.729

Standardized data score =  0.8185
```

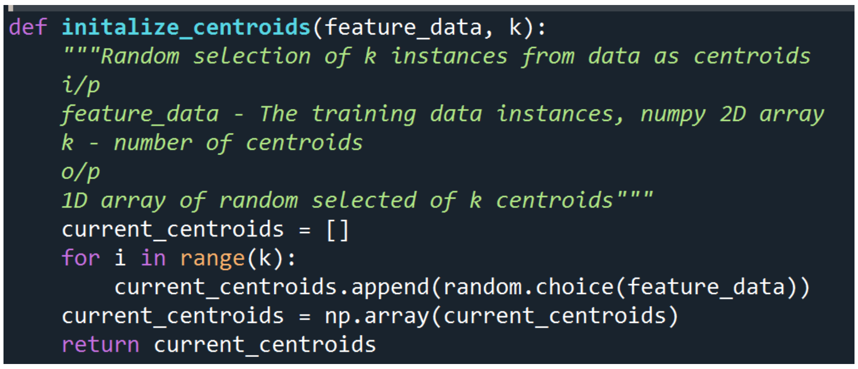
The code for this can be found in the zip namely “PART1B.py”

3. PART 2A

K means is one of the oldest and widely used learning algorithm, it can be achieved by following the steps below

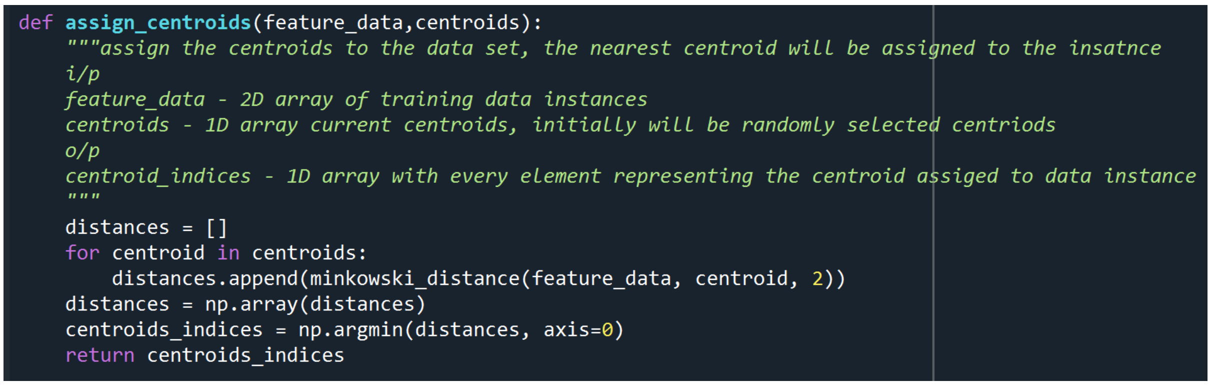
- Pick random ‘k’ number of centroids from the data sets
- Find the distance from each centroid to every point in the dataset
- Assign each instance to the centroid closest to the instance
- Find the mean of all data points that are associated with centroids, the mean is the new centroid
- Repeat the above 2 steps until convergence or for specified iterations

1. The function below is used to randomly select ‘k’ centroids initially (kmeans++ uses a different strategy to assign initial centroids)



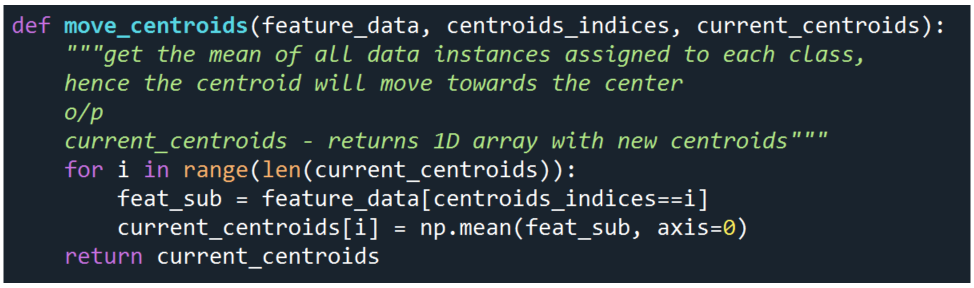
```python
def initalize_centroids(feature_data, k):
    """Random selection of k instances from data as centroids
    i/p
    feature_data - The training data instances, numpy 2D array
    k - number of centroids
    o/p
    1D array of random selected of k centroids"""
    current_centroids = []
    for i in range(k):
        current_centroids.append(random.choice(feature_data))
    current_centroids = np.array(current_centroids)
    return current_centroids
```

2. The function below is used to assign the centroid which is closest to the instance point



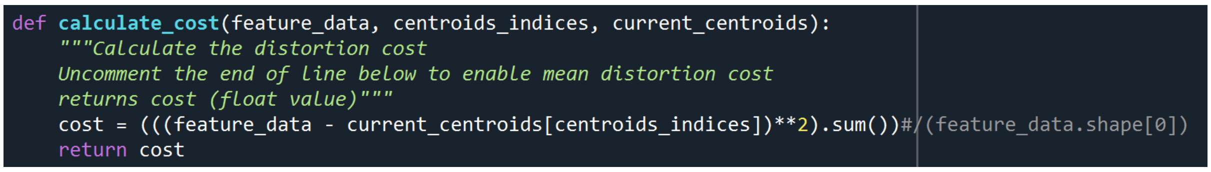
```python
def assign_centroids(feature_data,centroids):
    """assign the centroids to the data set, the nearest centroid will be assigned to the insatnce
    i/p
    feature_data - 2D array of training data instances
    centroids - 1D array current centroids, initially will be randomly selected centriods
    o/p
    centroid_indices - 1D array with every element representing the centroid assiged to data instance
    """
    distances = []
    for centroid in centroids:
        distances.append(minkowski_distance(feature_data, centroid, 2))
    distances = np.array(distances)
    centroids_indices = np.argmin(distances, axis=0)
    return centroids_indices
```

3. Finding the mean of all the instances associated to each centroid, this will be the new centroid



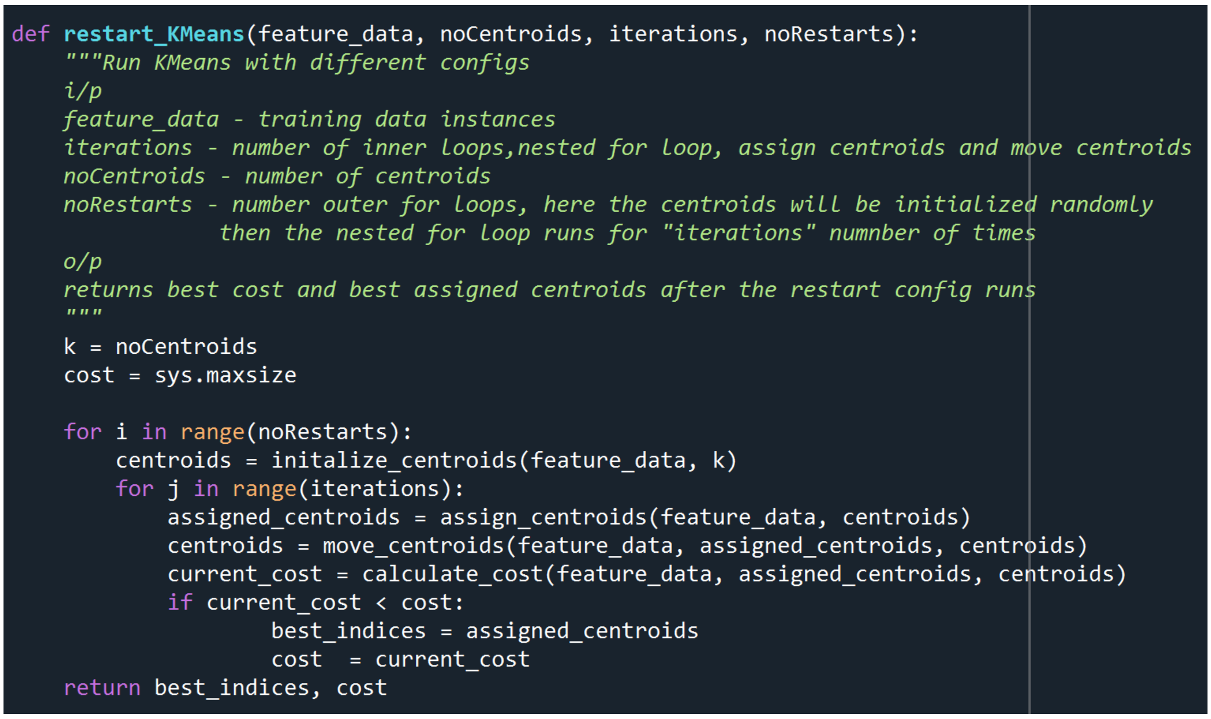
```python
def move_centroids(feature_data, centroids_indices, current_centroids):
    """get the mean of all data instances assigned to each class,
    hence the centroid will move towards the center
    o/p
    current_centroids - returns 1D array with new centroids"""
    for i in range(len(current_centroids)):
        feat_sub = feature_data[centroids_indices==i]
        current_centroids[i] = np.mean(feat_sub, axis=0)
    return current_centroids
```

4. Calculate the distortion cost after moving the centroid inorder to evaluate the performance of our model



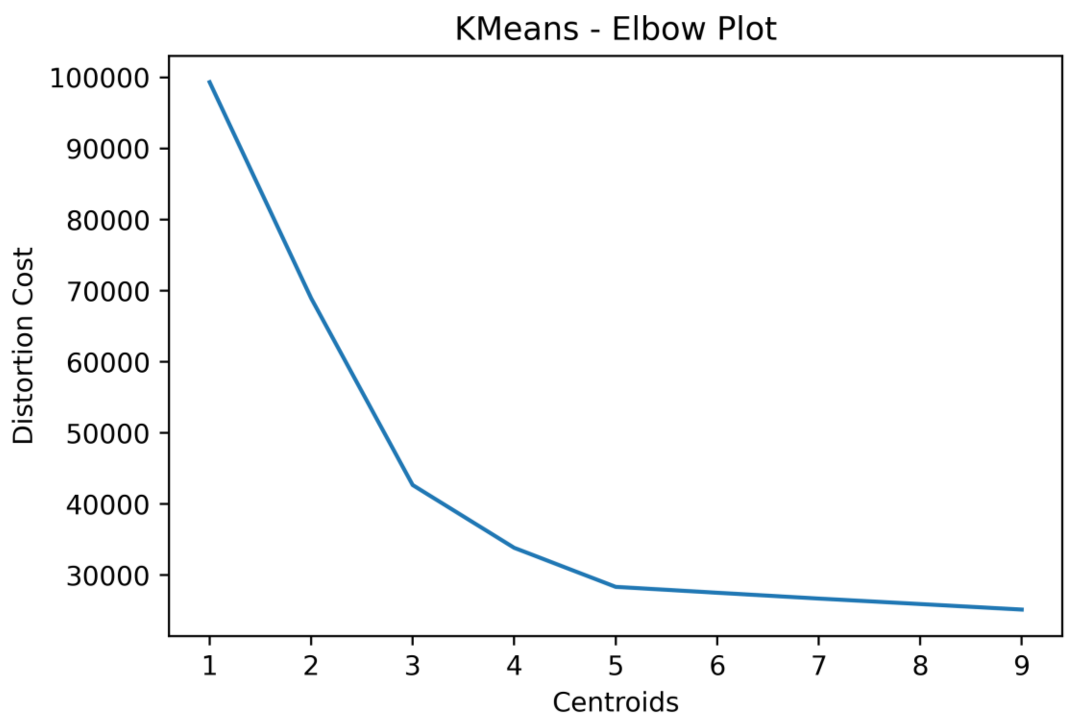
```python
def calculate_cost(feature_data, centroids_indices, current_centroids):
    """Calculate the distortion cost
    Uncomment the end of line below to enable mean distortion cost
    returns cost (float value)"""
    cost = (((feature_data - current_centroids[centroids_indices])**2).sum())#/(feature_data.shape[0])
    return cost
```

5. Restart K Means is a function which runs K means algorithm with the config as specified by the user



```python
def restart_KMeans(feature_data, noCentroids, iterations, noRestarts):
    """Run KMeans with different configs
    i/p
    feature_data - training data instances
    iterations - number of inner loops,nested for loop, assign centroids and move centroids
    noCentroids - number of centroids
    noRestarts - number outer for loops, here the centroids will be initialized randomly
                 then the nested for loop runs for "iterations" numnber of times
    o/p
    returns best cost and best assigned centroids after the restart config runs
    """
    k = noCentroids
    cost = sys.maxsize

    for i in range(noRestarts):
        centroids = initalize_centroids(feature_data, k)
        for j in range(iterations):
            assigned_centroids = assign_centroids(feature_data, centroids)
            centroids = move_centroids(feature_data, assigned_centroids, centroids)
            current_cost = calculate_cost(feature_data, assigned_centroids, centroids)
            if current_cost < cost:
                    best_indices = assigned_centroids
                    cost  = current_cost
    return best_indices, cost
```

6. From the elbow plot below, the cost drops drastically from ~90,000 to ~30,000 at k=5 and continues to decrease very slowly, therefore the optimal value of 'k' from the plot will be k=5
   The cost decreases with higher values 'k', but it is not optimal as the computational cost increases as k increases


KMeans - Elbow Plot
Distortion Cost
100000
90000
80000
70000
60000
50000
40000
30000
1
2
3
4
5
6
7
8
9
Centroids

4. PART 2B

One of the major drawbacks of KMeans algorithm is, it is computationally very expensive when the dataset is large, also it needs large memory space. Computational time for KMeans is O(kns), where n is number of instances, s is the number of non-zero elements in the instance, k is the number of cluster centres

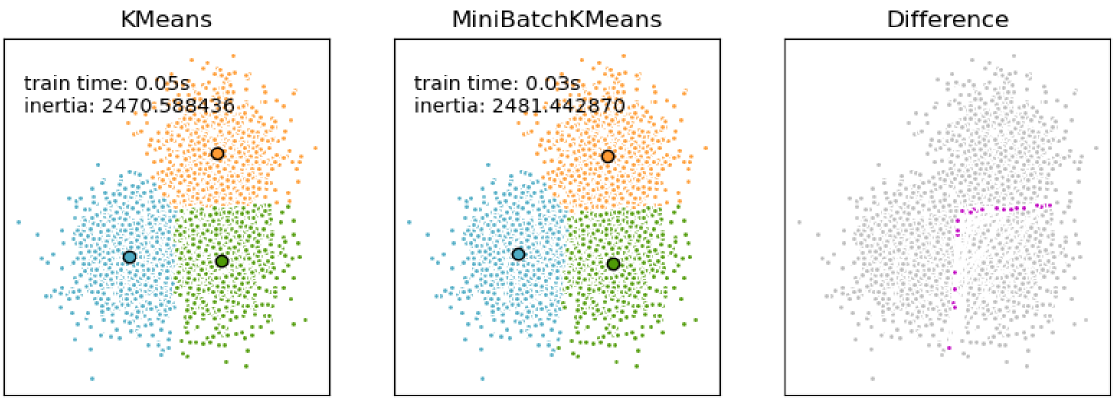
To reduce the temporal and space complexity, another approach known as mini-batch KMeans is used. The idea is to get 'b' random samples from the main dataset, over each iteration, a new random subset is obtained, and this is continued until convergence, a learning rate is applied which decreases with number of iterations

There will be loss in the cluster quality since we do not use all the data from the main dataset, but when comparing the computational cost, loss of cluster quality seems fine [6]

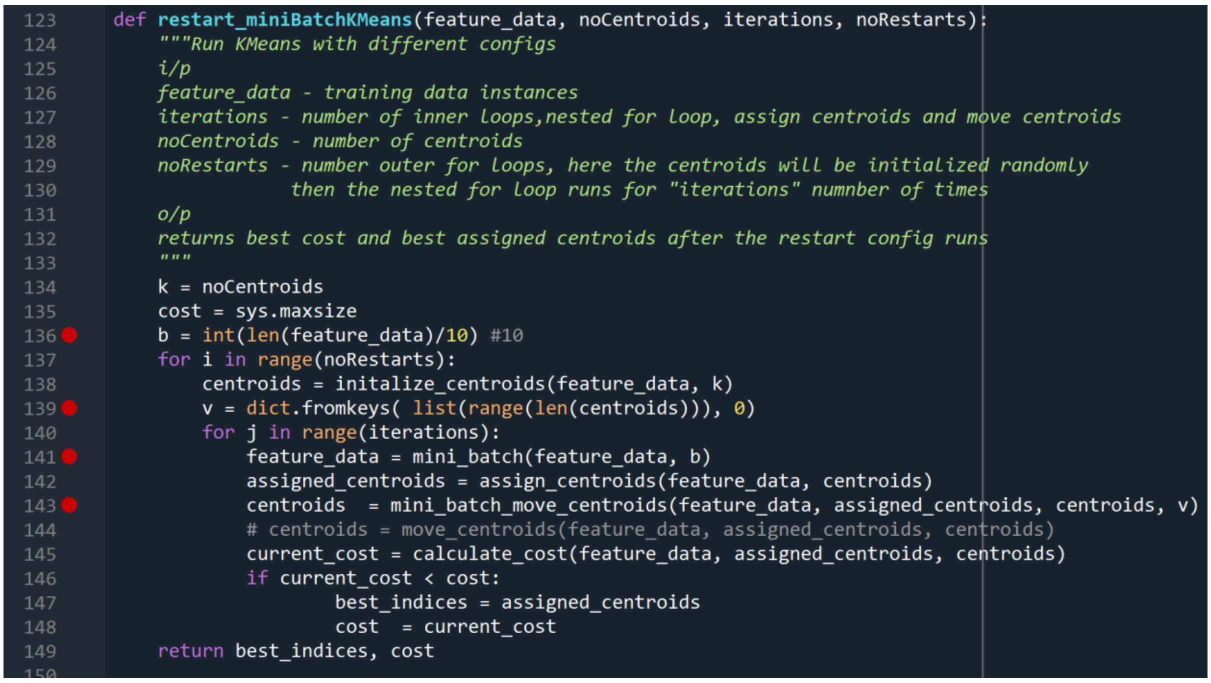
The algorithm for mini-batch K means is as shown in the picture below [6]

```
Given: k, mini-batch size b, iterations t, data set X
Initialize each c ∈ C with an x picked randomly from X
v ← 0
for $i$ ← $1$ to $t$ do
    M ← b examples picked randomly from X
    for $x \in M$ do
        d[x] ← f(C,x)
    end
    for $x \in M$ do
        c ← d[x]
        v[c] ← v[c] + 1
        $\eta \leftarrow \frac{1}{v[c]}$
        c ← $(1-\eta)c+\eta x$
    end
end
```

As discussed earlier, the major drawback of minibatch KMeans is the performance drop since we do not use the complete data set, the picture below shows the difference between standard K Means and minibatch KMeans outputs [7], we can notice the computational train time decrease by 40%, but the 'difference' picture shows few points wrongly clustered in minibatch which is a drawback

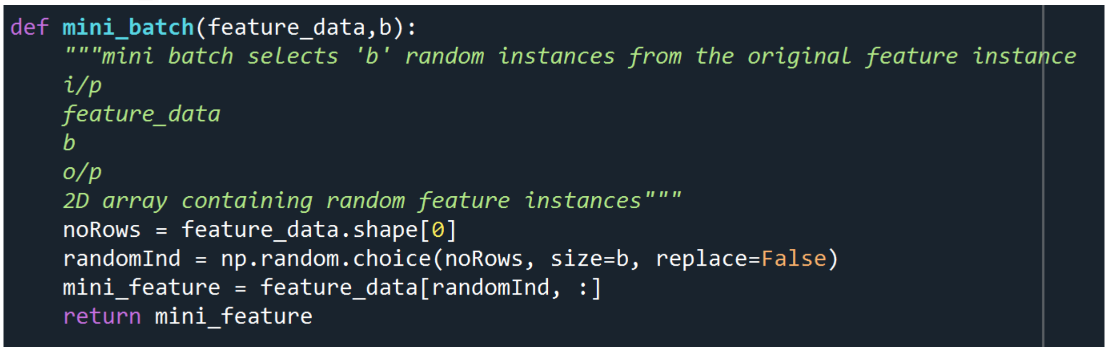


The changes I did to my main KMeans algorithm in part A is shown in snapshots below, the red dots in the picture below are major changes, the variable ‘b’ is for batch size which I have selected as 1/10 of total length of data, ‘v’ is a dictionary to count the centroid occurrences, the restart function calls mini batch and move centroid functions



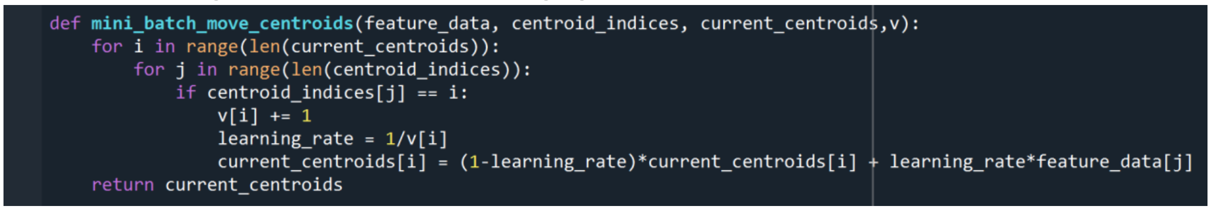
```
def restart_miniBatchKMeans(feature_data, noCentroids, iterations, noRestarts):
    """Run KMeans with different configs
    i/p
    feature_data - training data instances
    iterations - number of inner loops,nested for loop, assign centroids and move centroids
    noCentroids - number of centroids
    noRestarts - number outer for loops, here the centroids will be initialized randomly
                then the nested for loop runs for "iterations" numnber of times
    o/p
    returns best cost and best assigned centroids after the restart config runs
    """
    k = noCentroids
    cost = sys.maxsize
    b = int(len(feature_data)/10) #10
    for i in range(noRestarts):
        centroids = initalize_centroids(feature_data, k)
        v = dict.fromkeys( list(range(len(centroids))), 0)
        for j in range(iterations):
            feature_data = mini_batch(feature_data, b)
            assigned_centroids = assign_centroids(feature_data, centroids)
            centroids  = mini_batch_move_centroids(feature_data, assigned_centroids, centroids, v)
            # centroids = move_centroids(feature_data, assigned_centroids, centroids)
            current_cost = calculate_cost(feature_data, assigned_centroids, centroids)
            if current_cost < cost:
                    best_indices = assigned_centroids
                    cost  = current_cost
    return best_indices, cost
```

The function below is used to initialize or to select ‘b’ number of instances from the main dataset or the feature_data



```
def mini_batch(feature_data,b):
    """mini batch selects 'b' random instances from the original feature instance
    i/p
    feature_data
    b
    o/p
    2D array containing random feature instances"""
    noRows = feature_data.shape[0]
    randomInd = np.random.choice(noRows, size=b, replace=False)
    mini_feature = feature_data[randomInd, :]
    return mini_feature
```

This function follows the algorithm shows in the first picture under part2A, it calculates learning rate and then follows gradient descent for converging to the best center



```
def mini_batch_move_centroids(feature_data, centroid_indices, current_centroids,v):
    for i in range(len(current_centroids)):
        for j in range(len(centroid_indices)):
            if centroid_indices[j] == i:
                v[i] += 1
                learning_rate = 1/v[i]
                current_centroids[i] = (1-learning_rate)*current_centroids[i] + learning_rate*feature_data[j]
    return current_centroids
```

Bibliography

[1]http://www.coxdocs.org/doku.php?id=perseus:user:activities:matrixprocessing:learning:classificationparameteroptimization

[2]https://visualstudiomagazine.com/articles/2019/04/01/weighted-k-nn-classification.aspx

[3] Dynamic Feature Scaling for K-Nearest Neighbor Algorithm Chandrasekaran Anirudh Bhardwaj1 , Megha Mishra 1 , Kalyani Desikan2 https://arxiv.org/ftp/arxiv/papers/1811/1811.05062.pdf

[4] http://scikit-learn.sourceforge.net/dev/modules/feature_selection.html

[5] https://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf

[6]https://upcommons.upc.edu/bitstream/handle/2117/23414/R13-8.pdf?sequence=3&isAllowed=y

[7] https://scikit-learn.org/stable/auto_examples/cluster/plot_mini_batch_kmeans.html#example-cluster-plot-mini-batch-kmeans-py