Practical Machine Learning
Assignment 1 Report

Rajbir Bhattacharjee
R00195734

Table of Contents

Part 1A: Distance Weighted KNN Regression 3
Checklist 3
Final r^2 Value: 3
Implementation of calculate_distances 4
Implementation of *predict* 5
Implementation of *calculate_r2* 8
Part 1B: Parameters and Techniques for Model Performance 9
Feature Selection 9
Feature Extraction 10
Principal Component Analysis (PCA) 11
Fischer Linear Discriminant Analysis 11
PCA with Kernel 12
Using a different Distance Metric 13
Mahalnobis Distance 13
Learning Feature Weights 16
Implementation 17
Mahalanobis Distance 17
PCA 17
Forward/Backward Feature Selection 19
Part 2A: K-means Clustering 22
Checklist 22
Generate_centroids 22
assign_centroids 22
move_centroids 22
calculate_cost 23
restart_KMeans 24
Results 25
Part 2B: Mini-Batch K-Means 27

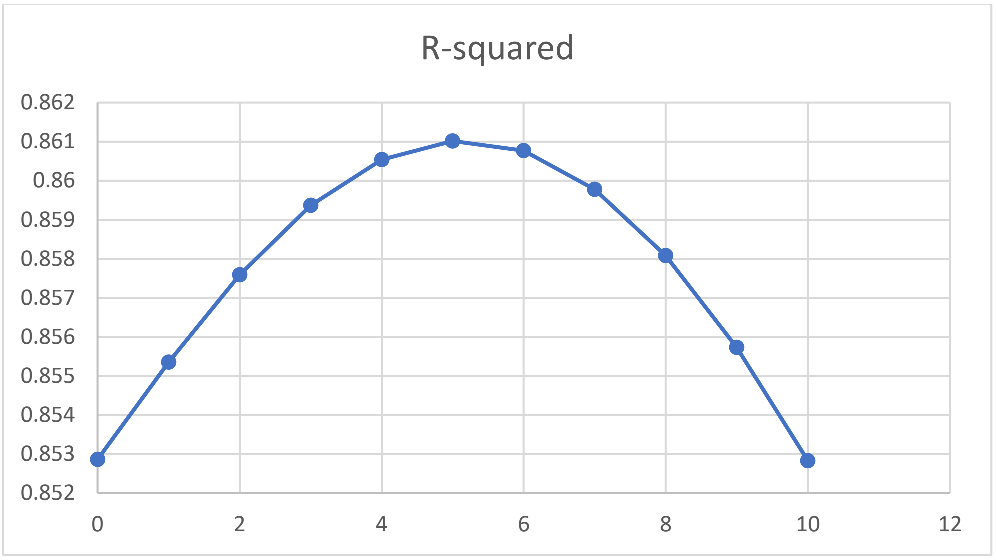
Part 1A: Distance Weighted KNN Regression

Checklist

Final r^2 Value:

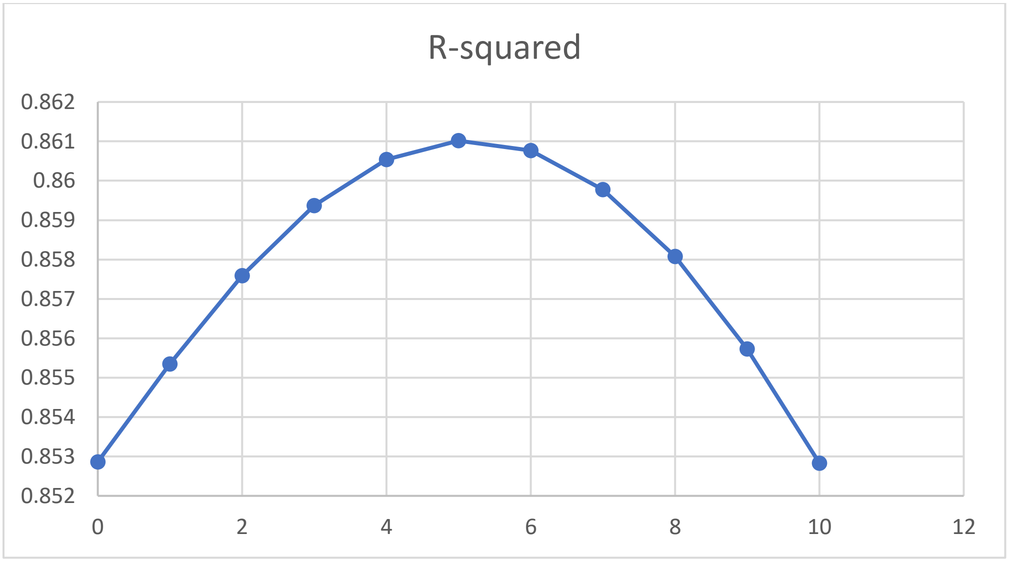
For k=3, the final r^2 value is: 0.8287402327561264
This was achieved, when the weight used was $(1/\text{distance})^3$.

The plot below illustrated how the r^2 error varies as the power in the weight is changed for k=3:



The global best value was achieved for k=10 and power=5. The r2 value was 0.8610154123824508.

For k=10, this is how the r^2 values varies with the power.



We shall see that feature selection led to much better r^2 values. Spefically, when forward selection was employed to select the 5 best features, and Mahalanobis distance was used, the model was able to achieve an r^2 value of 0.9691290819198476.

Implementation of calculate_distances

```python
def calculate_distances(allvalues:np.ndarray, row:np.ndarray)->np.ndarray:
    """
    Calculate the distance between all the training samples, and one
    point.

    If there are m training examples, and N features, then the shapes are as follows
    allvalues       m x N
    row             N x 1
    diff2           m x N
    np.sum(diff2)   m x 1
    Return          m x 1
    """
    diff2 = np.square(allvalues - row)
    """
    diff2 is m x N, summing it along axis 1 will give an m x 1 array
    """
    return np.sqrt(np.sum(diff2, axis=1))
```

Implementation of *predict*

Predict() was implemented as follows. The numpy function argsort() was used to find the smallest distances. The algorithm can be as follows:

1. Create a heap using the distances as the sort index in O(n) complexity
2. Remove k elements from the heap, each in log(n) complexity

The total complexity of the above is O(n + k*log(n)), while argsort has the best case complexity of O(n log(n))

However, the same was not implemented. For a small data of this size, this algorithm might actually perform worse than a sort because of the overheads involved. This would disappear as n grows larger.

```
def predict(train_features:np.ndarray, train_values:np.ndarray, test_features:np.ndarray, k:int, 
p:int)->np.ndarray:
    global g_normalize_to_zero_mean_and_unit_variance, g_scale_between_zero_and_one

    if g_normalize_to_zero_mean_and_unit_variance:
        """
        If normalization is required, normalize to zero mean and unit
        standard deviation.
        """
        stddvarr = np.std(train_features, axis=0)
        meanarr = np.mean(train_features, axis=0)
        train_norm = normalize(train_features, stddvarr, meanarr) # Normalize
    else:
        """
        If normalization is not required do nothing
        """
        train_norm = train_features

    if g_scale_between_zero_and_one:
        """
        If scaling is required scale to a range in [0, 1]
        """
        amin, amax = get_min_max(train_norm)
        train_norm = scale(train_norm, amin, amax)                # Scale

    if g_normalize_to_zero_mean_and_unit_variance:
        """
        Normalize the test values by the same amount we had normalized the
        training features.
        It is important to use the values of stddvarr and meanarr exactly
        as we had used in the training normalization
        """
        test_norm = normalize(test_features, stddvarr, meanarr)
    else:
        test_norm = test_features

    if g_scale_between_zero_and_one:
        """
        Scale the test features exactly as we had scaled the train features
        Again, it is important to use the same values of amin and amax that we
        had used in the training scaling
        """
        test_norm = scale(test_norm, amin, amax)        # Normalize


    all_predictions = []

    """
    This version uses np.argsort(), and then chooses k values out of it
    The best case complexity of np.argsort() is O(n log(n))

    A better complexity can be achieved by using heapify.
    Heapify operation can run in O(n), and removing something from a heap is
    O(log(n))

    The algorithm can be as follows:
    1. Create a heap using the distances as the sort index in O(n) complexity
    2. Remove k elements from the heap, each in log(n) complexity
    The total complexity of the above is O(n + k*log(n))

    However, this scheme has not been implemented. For this size of data,
    the overheads might actually be higher than the speedup achieved.

    For larger data sets, this might be more efficient.
    """
    for i in test_norm:
        """
        For each data point in the test data, categorize it
        """

        """
        Calculate the idstance between this point, and all other points
        """
        distances = calculate_distances(train_norm, i)

        """
        Sort based on the distance, and get the k smallest indices
        """
        sorted_indices = np.argsort(distances)[0:k]

        """
        Get the nearest distances, this will be used in the weighted average
        """
        nearest_distances = distances[sorted_indices]

        """
        Avoid division by zero
        """
        nearest_distances = nearest_distances + 0.0000000001

        """
        Get the y for the nearest k points
        """
        nearest_values = train_values[sorted_indices]

        """
        Weight is inverse of distance
        """
        nearest_weights = 1 /  nearest_distances

        """
        The power used in calculating the weight is a parameter
        weight = (1/distance) ^ p
        """
        nearest_weights = nearest_weights ** p

        """
        Multiply each of the nearest weights with the nearest y values
        and sum it up
        """
        prediction = np.sum(np.multiply(nearest_weights, nearest_values))

        """
        Divide by the sum of the weights so that everything adds up and the
        weighted average is proper
        """
        prediction = prediction / np.sum(nearest_weights)

        """
        Add this prediction to an array, so that we can get all the predictions
        for all the points in one place and return it
        """
        all_predictions.append(prediction)

    return all_predictions
```

Implementation of *calculate_r2*

```
def calculate_r2(predicted:np.ndarray, actual:np.ndarray)->float:
    """
    Calculate the r2 error and return a float
    If there are m examples
    predicted       mx1
    actual          mx1
    """

    """
    Find the square residuals. This is mx1
    SUM((predicted_i - y_i) ^ 2)
    """
    sum_square_residuals = np.sum(np.square(actual - predicted))

    """
    Find the mean of the actual values
    """
    mean_actual = np.mean(actual)

    """
    Find the sum of squares, this is mx1:
    SUM((y_mean - y_i) ^ 2)
    """
    sum_squares = np.sum(np.square(actual - mean_actual))

    """
    Return the r2 value
    """
    return 1 - (sum_square_residuals / sum_squares)
```

Part 1B: Parameters and Techniques for Model Performance

By default, KNN will weigh the contribution of each feature equally when using standard Euclidean distance.

Euclidean distance is given by:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Here, p_i and q_i are features in the vectors **p, q**. As we can see, here no feature in this metric is given a greater weight than any feature.

Scaling and normalization or standardization must be first performed on each feature so that all the features have values on a similar scale. If that is not done, then one feature might affect the same result too much at the expense of other features – and that might lead to sub-optimal results. One common way to standardize is to ensure that all the features have zero mean and unit standard deviation. Standardization ensures that all the features have a similar range of values. However, standardization does not resolve all problems. This is because:

1. There might be irrelevant features, and those will also be given equal weight, thereby affecting the result.
2. Some features might be correlated, and as such, those features will be counted twice. Again, this might have an effect on the result.
3. Even if all the features are on the same scale, and are all independent of each other, it could just be that some of them are more important than others, and a standard Euclidean distance does not take care of such differences. For example, if age and industry/profession were used to predict the salary of a person, both age and industry/profession may be independent of each other, and may have an effect on the salary, but industry/profession may affect the salary much more than a person's age. The standard KNN implementation doesn't not give a higher weightage to a person's industry/profession after normalization has been performed on it.

With a distance weighted KNN, the above problems still remain the same.

There are several ways to deal with this problem, and they can broadly be categorized as[1] [2]:

1. Feature Selection
2. Feature Extraction and/or changing the dimensionality
3. Using a different distance measure (generic)
4. Using different weights for each feature with a custom distance measure. Learning of weights can be employed, and there is a possibility of using a different model to learn the weights.
5. Using different weights for different neighbours

Feature Selection

In feature selection, one or more features are discarded, and then the KNN is run.

Feature Selection can broadly be categorized into three types:

1. Optimum Methods. This is an exhaustive search where all permutations and combinations of features. However, this quickly becomes intractable as the number of permutations and combinations grows exponentially with the number of features.
2. Heuristic

[1] Sudeshna Sarkar, https://www.youtube.com/watch?v=KTzXVnRlnw4

[2] He et. al., Attribute Value Weighting in K-Modes Clustering, https://arxiv.org/ftp/cs/papers/0701/0701013.pdf

3. Randomized[3]

The goal of features selection is to do the following:

1. Find uncorrelated features
2. Eliminate irrelevant features

Evaluation of features for feature selection can be grouped into two categories

1. Unsupervised or filter methods
   In this, training and testing is performed using the training data only. The cross-validation set is never looked at.
2. Supervised or wrapper methods
   In this method, the features are selected by running over a learning algorithm. Training is done using the training examples, and validation is done over the test data set. A secondary model may be used to predict the best features

Some common methods for feature selection are

1. Forward Search
   In this method, initially the set of features is empty, and features are added one by one. While adding a feature, all features are evaluated, and the classification error is obtained if the feature is incorporated. The feature that gives the best gain in optimality is chosen. The process is stopped when the improvement in optimality falls below a cutoff.
2. Backward Selection
   This method begins with all the features in the selected set. Features whose performance that impact the error the smallest adversely are repeatedly removed.

Features selection can further be categorized as

- Univariate Features Selection
  In univariate feature selection, each feature is considered independently, and a decision is taken whether to include it for evaluation or not. Common measures used are
  a. Pearson Correlation Coefficient
  b. F-Score
  c. Chi-Square
  d. SNR
  e. Mutual Information

  These methods measure correlation between a feature and the target variable.
- Multivariate Feature Selection

In our case, we implemented forward and backward search, and the model was able to find a set of five features that affect the output the most and produced the best results.

Feature Extraction

In feature extraction, the current set of features is used to derive a new set of features. This is different from feature selection where features are simply selected, but here, features are combined with each other to form a completely new set of features.

This usually involves projection of a higher dimensional feature space to a lower dimensional feature space. The new features are uncorrelated and cannot be reduced further.

[3] Boutsidis, et.al., Unsupervised Feature Selection for the k-Means Clustering Problem, https://www.stat.berkeley.edu/~mmahoney/pubs/NIPS09.pdf

Some common techniques are

1. PCA
2. SVD
3. Fischer Linear Discriminant Analysis
4. PCA with different kernels
   a. Cosine Similarity
   b. Radial Basis Function (RBF)
   c. Polynomial Kernel
   d. Sigmoid Kernel
   e. Laplacian Kernel
   f. Chi-Squared Kernel

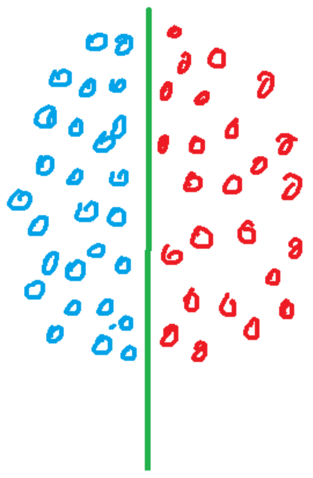
Principal Component Analysis (PCA)

In PCA, the existing set of features is projected on a new space with fewer dimensions. The axes of the new space are chosen in such a way that it maximizes the variance in the data. Furthermore, the axes are ranked in the order of the most variance (therefore impact), and only the features that contribute most information are chosen.

Apart from the problem with weights of features, PCA is also helpful in dimensionality reduction when there are few samples and a lot of dimensions and helps us mitigate the curse of dimensionality.

PCA was implemented as a step before KNN, however, the results were not encouraging, and the performance was sub-optimal. The results are Presented later.

One problem that PCA runs into is that the new axes chosen in PCA are chosen to maximize variation. However, that might not lead to good class separation. Consider the situation below:



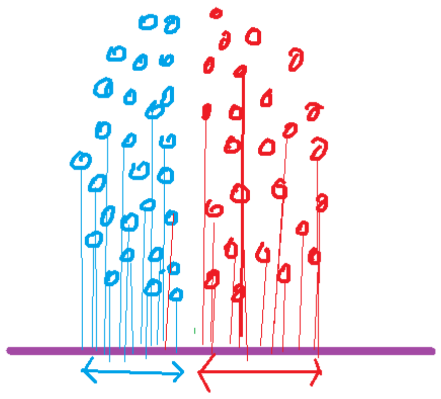
In the diagram above, there are two classes, and the principal axis chosen is shown along the green line. The principal axis is chosen such that the variance in the data is maximized. This does a very good job of describing the data, however, in this case, this principal is not very helpful in helping us categorize the data as both the red and blue categories overlap along the axis.

Fischer Linear Discriminant Analysis

In Fischer Linear Discriminant Analysis, the new axis are chosen such that:

1. It maximizes inter-class distance
2. Minimizes intra-class distance

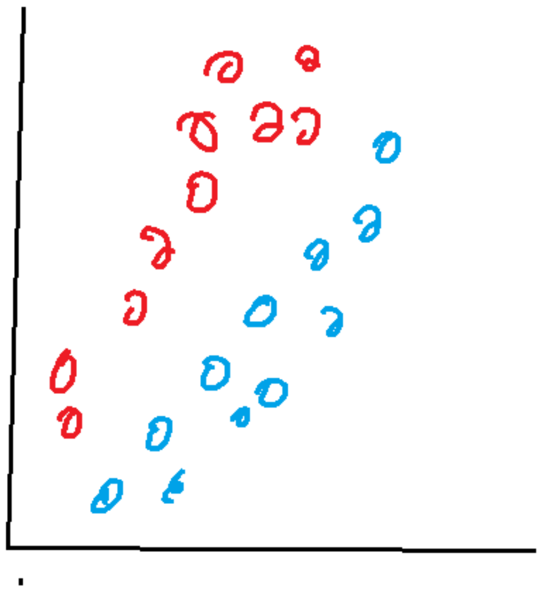
In the previous example, using LDA, the following axis would be chosen, and we will be able to separate the classes easily as they do not overlap over the axis



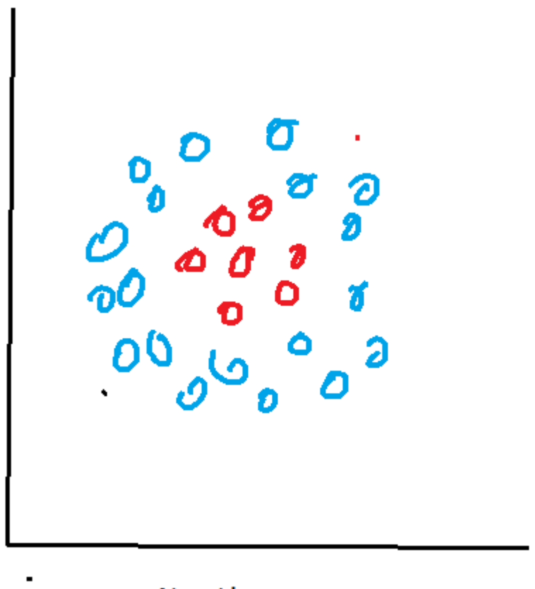
Both PCA and LDA have their uses. PCA does not need class information to work, and can be used to reduce the number of features. LDA needs class information, but can be used to get better class separation.

PCA with Kernel

Generally, PCA works very well when the data is linearly separable. However, for non-linear separation, PCA does not perform very well. [4]



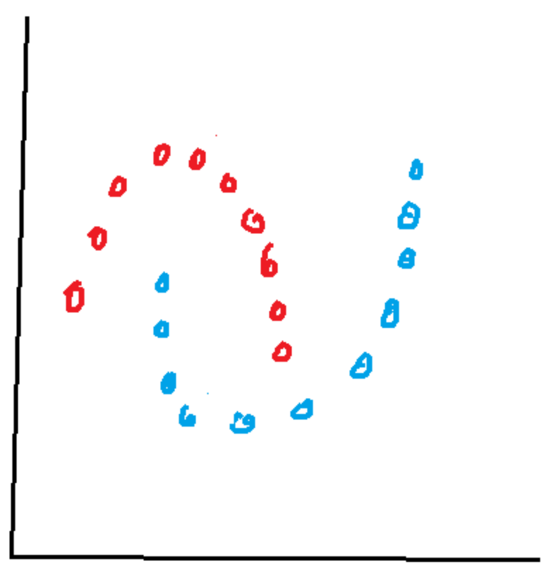
Linear

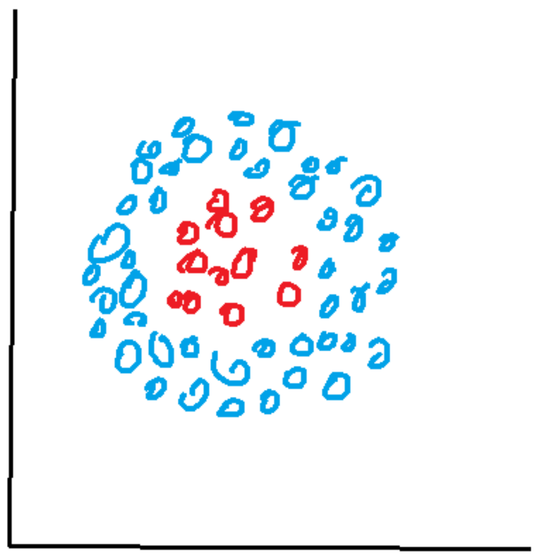


Non-Linear

The RBF Kernel is one such kernel which gives good results for radially separated classes. For example, RBF kernel would give better separation for the following distributions, and regular PCA will give sub-optimal results for these shapes.

[4] http://rasbt.github.io/mlxtend/user_guide/feature_extraction/RBFKernelPCA/





Using a different Distance Metric

A host of other distance metrics can be used to ameliorate some of the problems with Euclidean distance. Minkowski's distance is a popular choice, but it also assigns equal weight to all features.

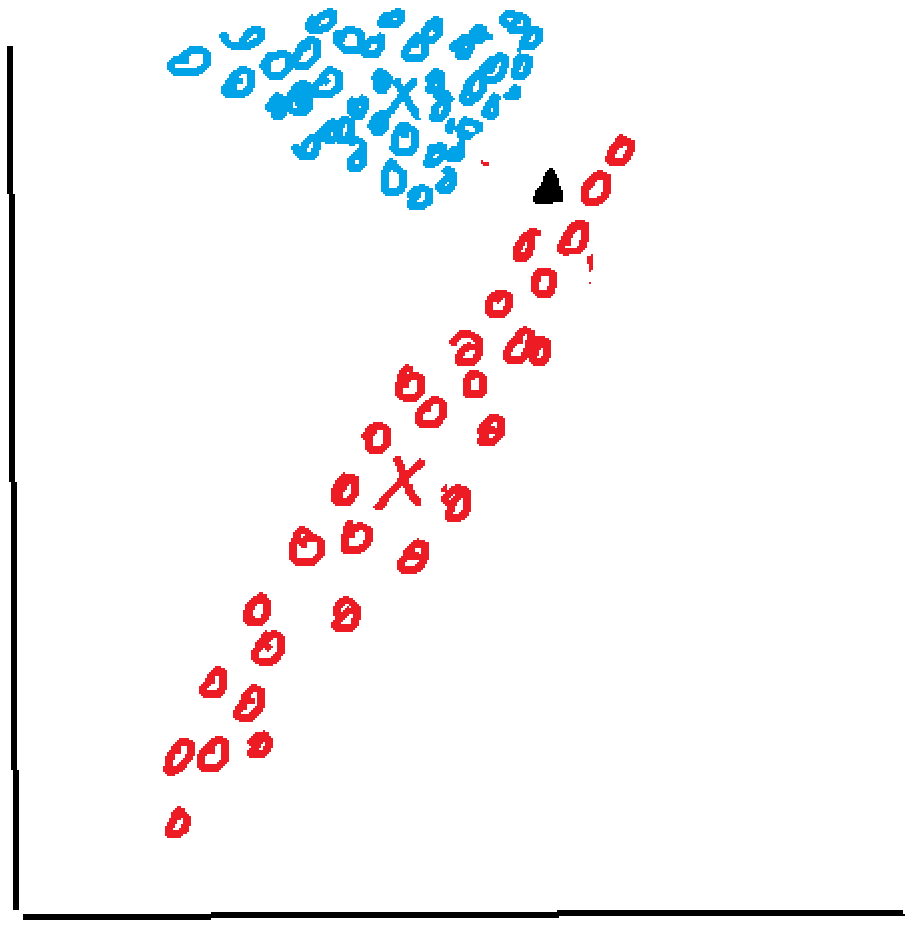
Some other choices for distance functions can be categorized as suitable for[5]

1. Real Valued Vector Spaces – Euclidean, Mahalnobis, Chebyshev, Minkowski etc.
2. Latitude and Longitude form – havershine
3. Integer Valued Vector – Hamming, Canberra, Bray Curtis
4. Boolean – Jaccard, Matching, Dice, Kulinski, Rogers-Tanimoto, Russel-Rao, Sokal-Michener, etc.
5. Strings – Cosine distance is a popular choice

Mahalnobis Distance

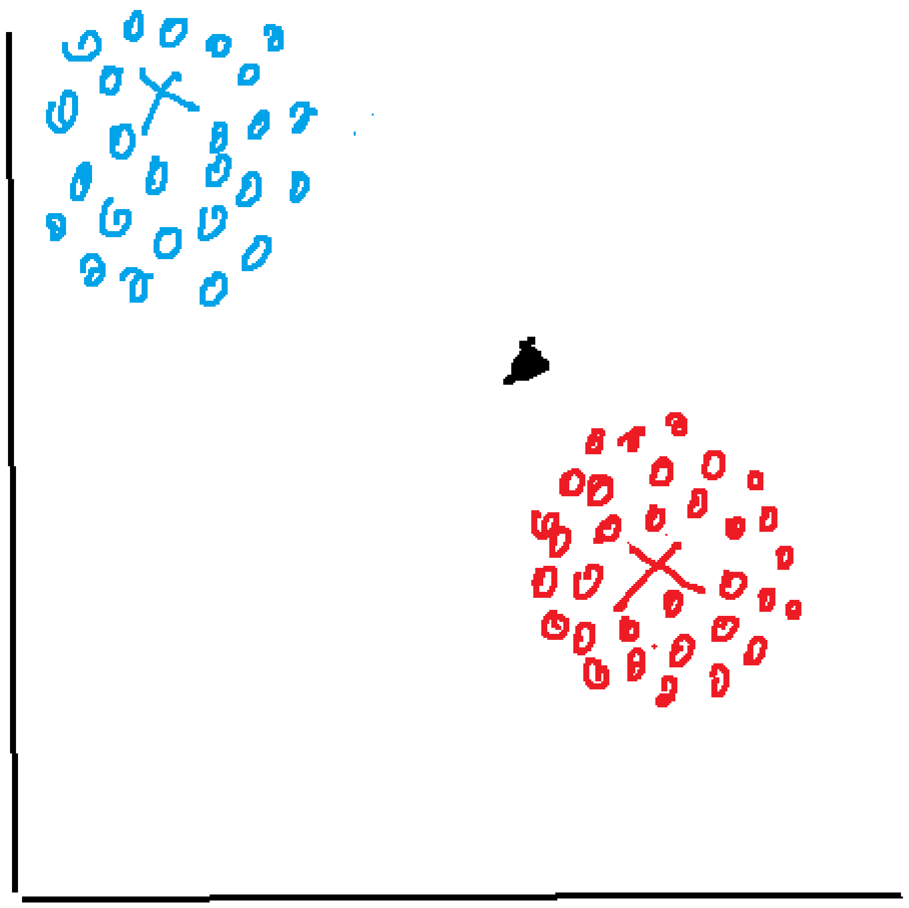
Consider a distribution that looks like this.

[5] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html

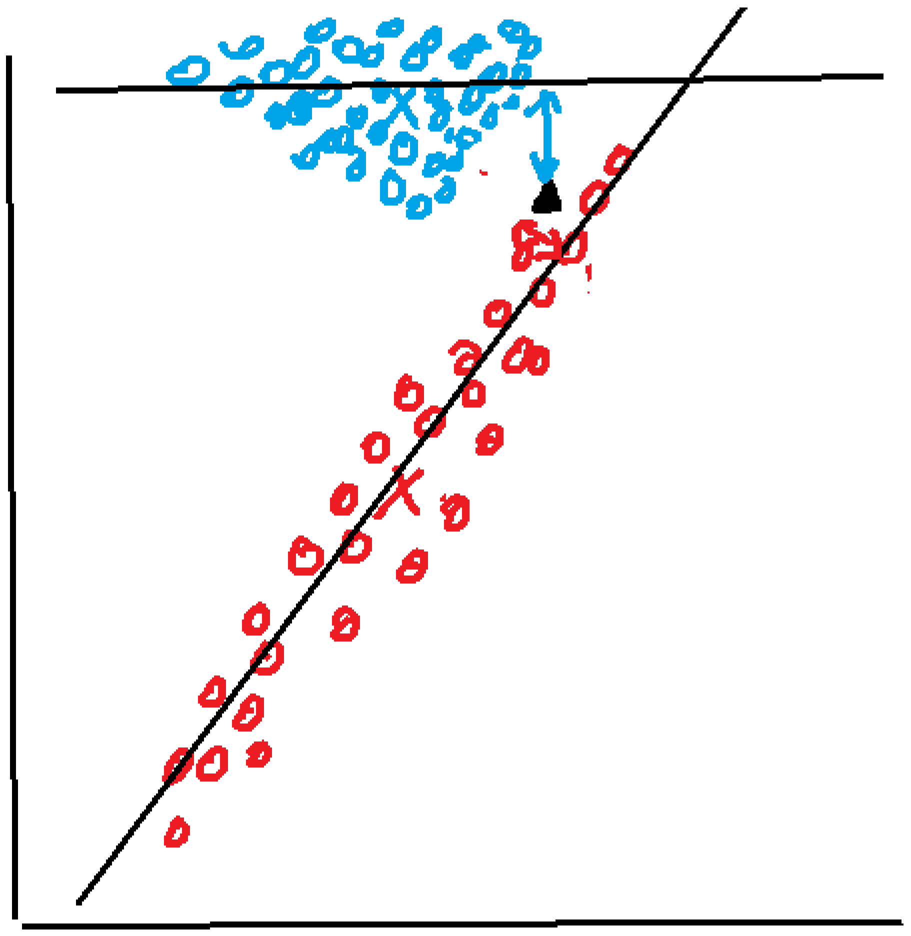


Here there are two classes, red, and blue, and the centroids are marked with ‘x’. If we are to evaluate the point marked by the black triangle, and its distance from the centroid is measured, then it is closer to the blue centroid than the red centroid, and might be categorized as blue. However, the distribution of red items is not a sphere, and elongated.

In general, k-NN works well when the distribution of the classes is spherical like below:



Mahalanobis distance is one which takes care of correlations between the data, and the shapes of the distributions. Mahalanobis distances uses the correlation covariance matrix to calculate the distance from an axis that describes the similarity rather than the Euclidean distance.



Mahalanobis distance measure was implemented and gave slightly better results than standard Euclidean distance. The results are presented later.

Learning Feature Weights

Another technique widely used is weighted k-means with weights per feature. These weights can be learned by employing another model, or by experimentation. One approach is given by Chan et al. by using dissimilarity metrics to learn weights. [6] Another approach is given by Chen and, Rege and Dong give another approach to use domain knowledge to achieve better results with semi-supervised clustering[7]. Huang et.al. propose yet another way to learn weights in an automated manner[8]. De Amorim builds on the above methods to provide yet another method of learning weights[9].

[6] https://www.sciencedirect.com/science/article/abs/pii/S0031320303004035

[7] https://link.springer.com/article/10.1007%2Fs10115-008-0134-6

[8] https://www.taylorfrancis.com/books/e/9780429150418/chapters/10.1201/9781584888796-19

[9] https://www.dcs.bbk.ac.uk/site/assets/files/1025/amorim.pdf

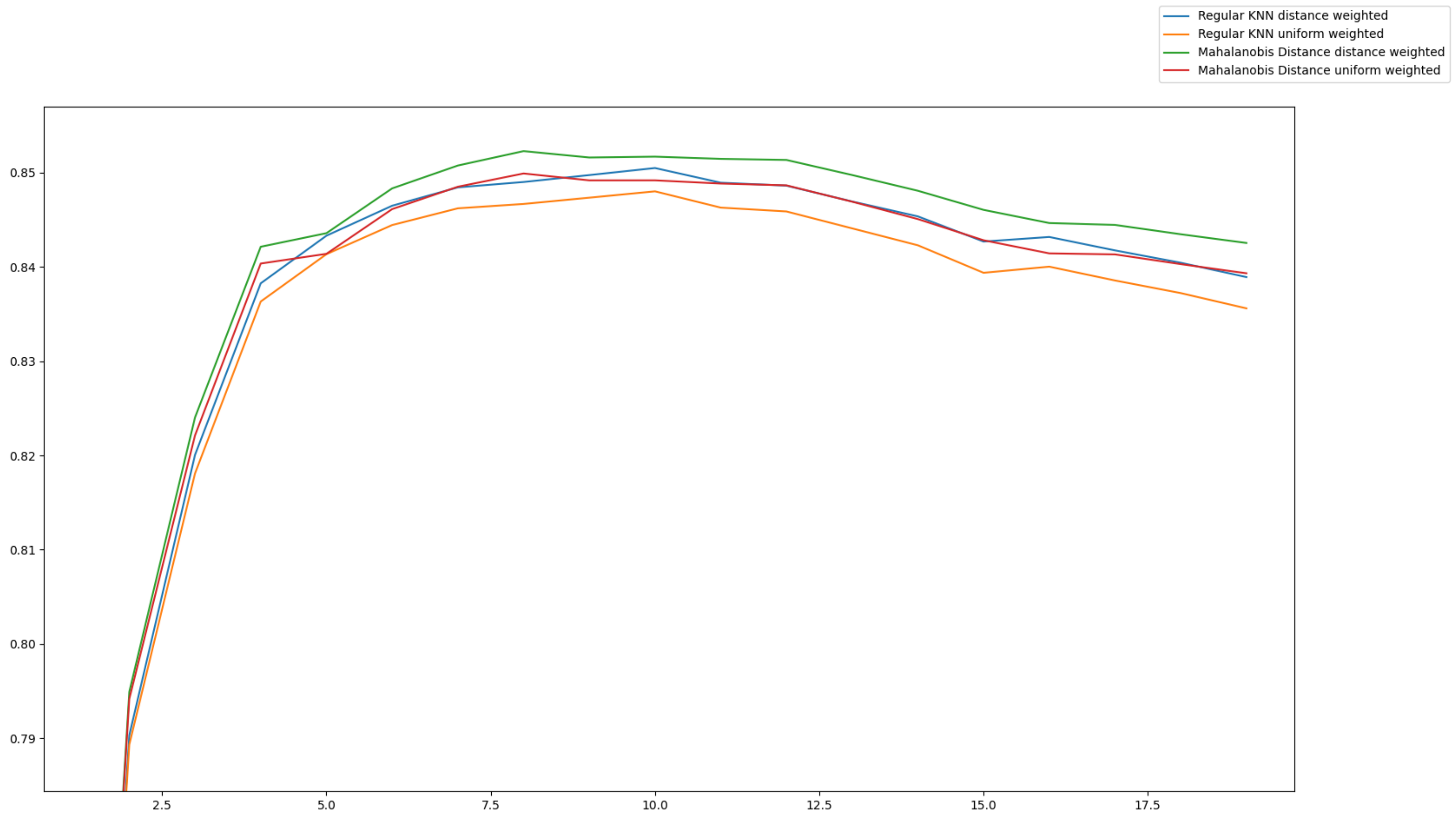
Implementation

Mahalanobis Distance

Mahalanobis distance was implemented using the mlxtend library

```
def knn_mahalanobis(x_train, y_train, x_test, y_test, fig, ax, description, wt='distance'):
    r2scores = []
    indices = []
    for i in range(1, 20):
        neigh = KNeighborsRegressor(n_neighbors=i, metric='mahalanobis', metric_params={'V':
np.cov(x_train.T)}, weights=wt)
        neigh.fit(x_train, y_train)
        predicted = neigh.predict(x_test)
        r2 = r2_score(y_test, predicted)
        r2scores.append(r2)
        indices.append(i)
        print(f"{description} - n={i} - r2 = {r2}")
    ax.plot(indices, r2scores, label=description)
```

The results obtained were slightly better than euclidean distance, and the regression value the model was able to achieve was 0.85228 at k=8. The graph below shows the performance of regular KNN over Mahalanobis KNN with varying k.

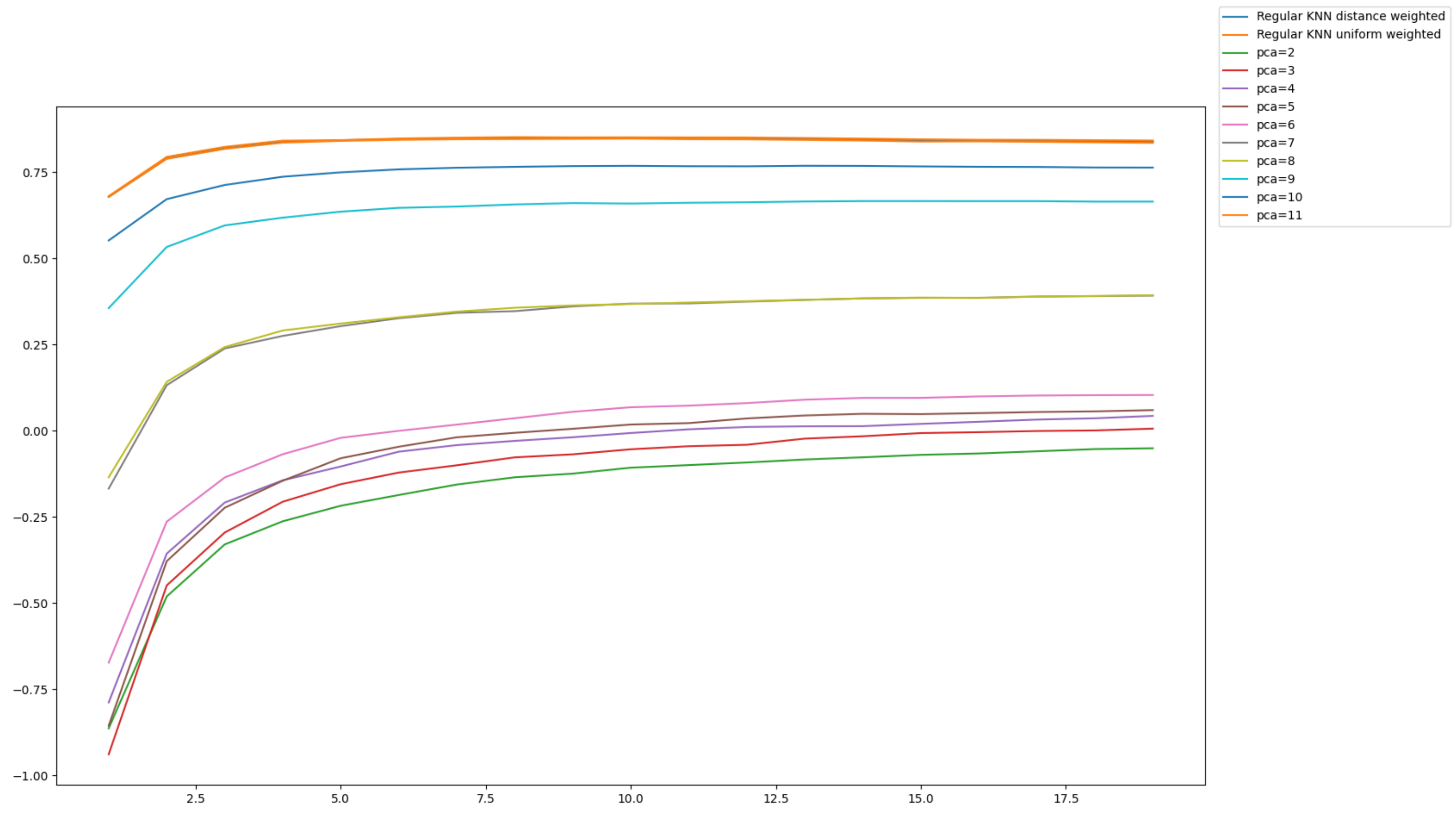


PCA

PCA with different weights were implemented using the following code:

```
def knn_pca(x_train, y_train, x_test, y_test, fig, ax, description, wt='distance'):
    for j in range(2, x_train.shape[1]):
        if j != (x_train.shape[1] -1):
            pca = PCA(n_components=j)
            pca.fit(x_train)
            x_train_pca = pca.transform(x_train)
            x_test_pca = pca.transform(x_test)
        else:
            x_train_pca = x_train
            x_test_pca = x_test
        r2scores = []
        indices = []
        for i in range(1, 20):
            neigh = KNeighborsRegressor(n_neighbors=i, metric="mahalanobis", metric_params={'V':
np.cov(x_train_pca.T)}, weights=wt, n_jobs=6)
            neigh.fit(x_train_pca, y_train)
            predicted = neigh.predict(x_test_pca)
            r2 = r2_score(y_test, predicted)
            r2scores.append(r2)
            indices.append(i)
            print(f"{description} - pca={j} - n={i} - r2 = {r2}")
        the_description = f"{description}={j}"
        ax.plot(indices, r2scores, label=the_description)
```

The plot below shows r^2 value plotted against varying k. The different lines represent how many axes were selected after PCA. The results were not promising, and in lower dimensions, PCA produced a negative r^2 value.



Regular KNN performed better than PCA. One explanation for this may be that there are a lot of features that adversely impact the accuracy in this dataset (as we shall see later in feature selection). When PCA reduced dimensionality, it introduced noise from noisy features which impact the accuracy adversely into other features, leading to worse performance.

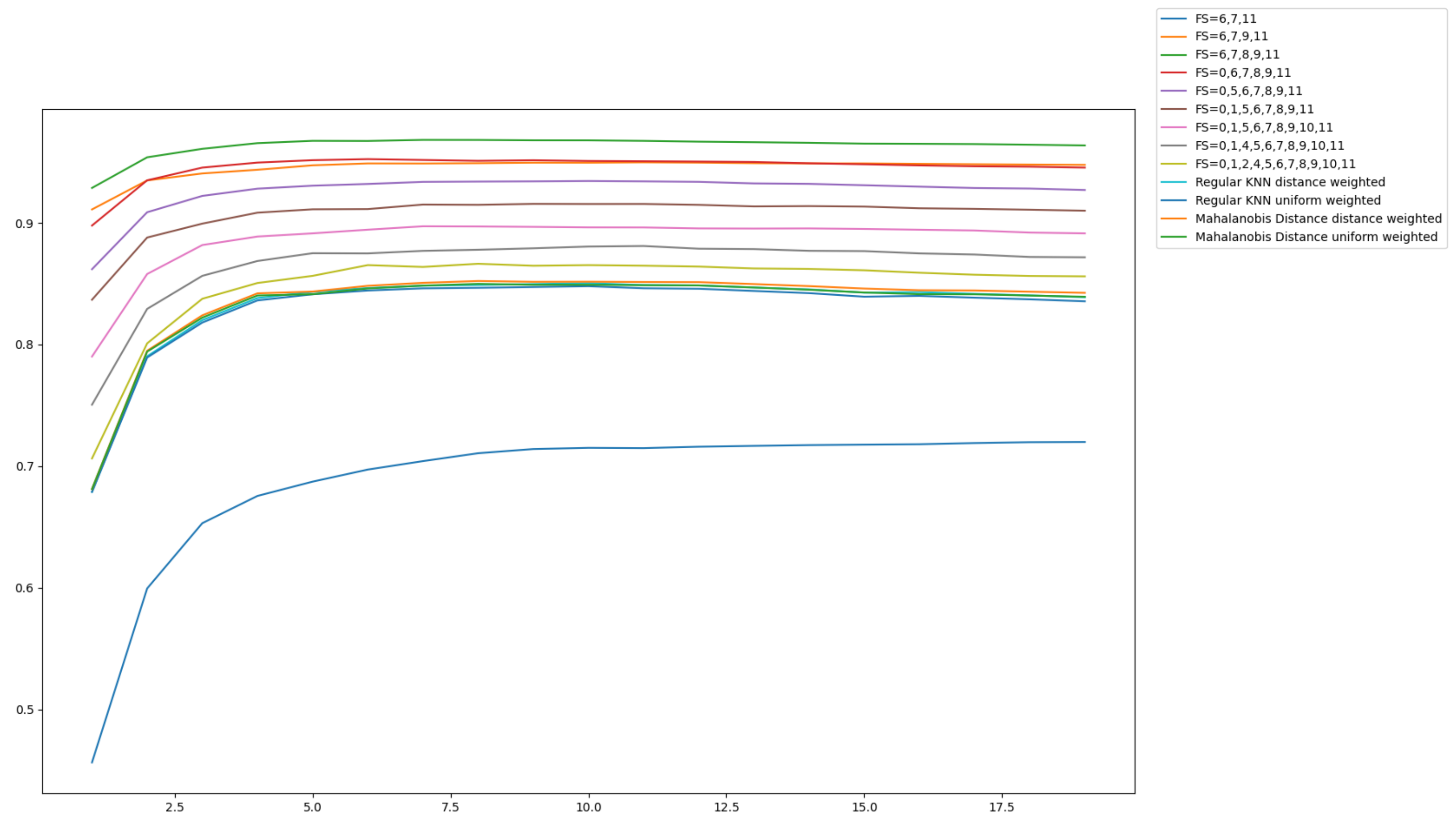
Forward/Backward Feature Selection

Forward and backward feature selection was implemented using the mlxtend library. Both these selection models select features by running the KNN as an evaluation metric multiple times. Both forward and backward selection produced similar results, but forward selection was faster than backward selection. That is because forward selection trains on fewer features on an average (since it starts with 0 features and keeps building till it has reached the optimal, while backward selection starts with all features and continually drops features).

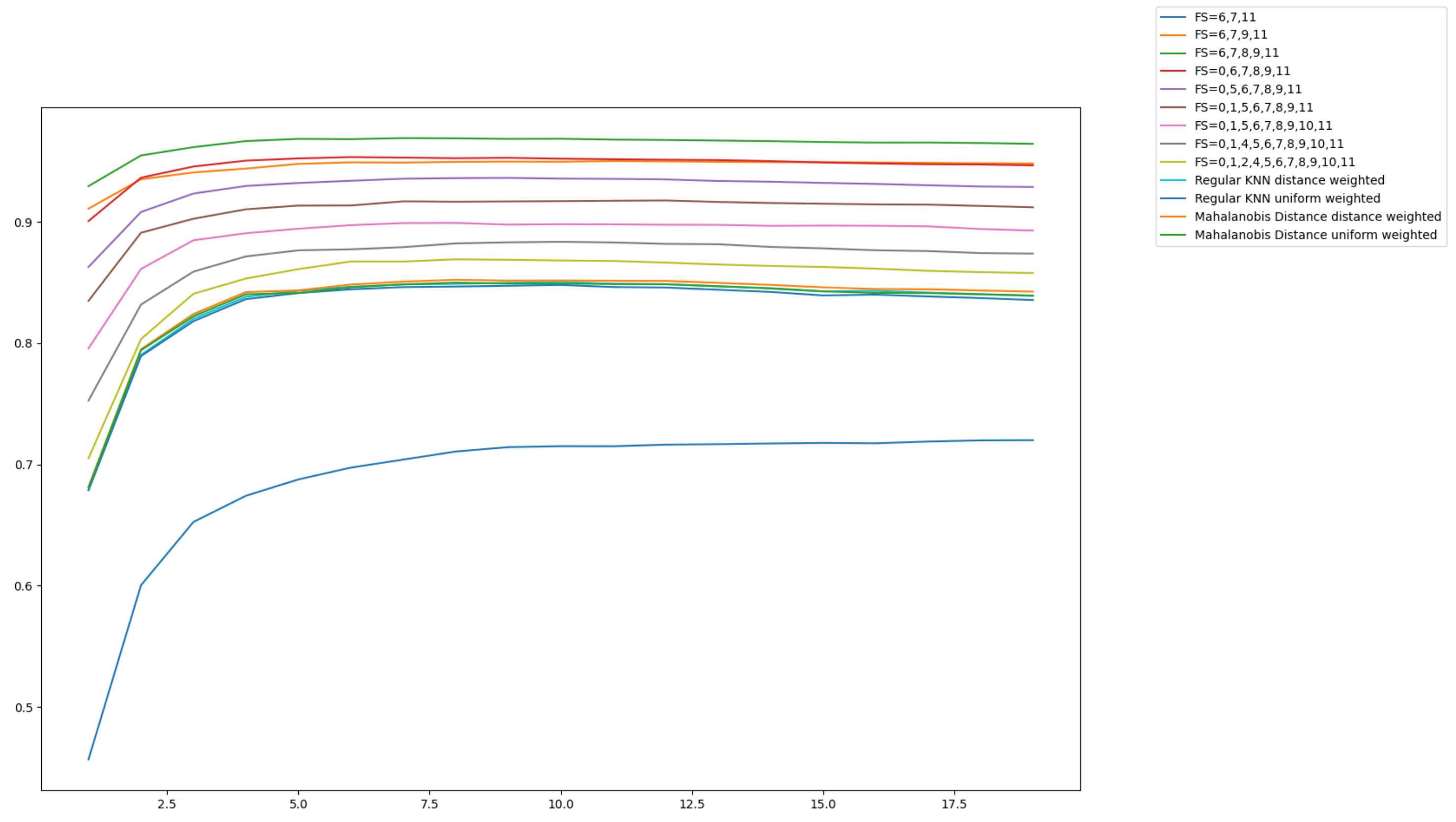
```python
def forward_selection(x_train, y_train, x_test, y_test, fig, ax, description, wt='distance'):
    for j in range(3, x_train.shape[1]):
        knn = KNeighborsRegressor(n_neighbors=10, weights='distance', n_jobs=6)
        sfs = SequentialFeatureSelector(
                knn,
                k_features=j,
                forward=True,
                floating=False,
                scoring='r2',
                verbose=0,
                n_jobs=6)
        kk = sfs.fit(x_train, y_train)
        selected = [int(i) for i in sfs.k_feature_names_]
        print(selected)
        new_train_x = x_train[:,selected]
        new_test_x = x_test[:,selected]
        print(new_train_x.shape, new_test_x.shape)
        knn_regular(new_train_x, y_train, new_test_x, y_test, fig, ax, f"FS={','.join(sfs.k_feature_names_)}", wt)
        print('-' * 80)


def backward_selection(x_train, y_train, x_test, y_test, fig, ax, description, wt='distance'):
    for j in range(3, x_train.shape[1]):
        knn = KNeighborsRegressor(n_neighbors=10, weights='distance', n_jobs=6)
        sfs = SequentialFeatureSelector(
                knn,
                k_features=j,
                forward=False,
                floating=True,
                scoring='r2',
                verbose=0,
                n_jobs=6)
        kk = sfs.fit(x_train, y_train)
        selected = [int(i) for i in sfs.k_feature_names_]
        print(selected)
        new_train_x = x_train[:,selected]
        new_test_x = x_test[:,selected]
        print(new_train_x.shape, new_test_x.shape)
        knn_regular(new_train_x, y_train, new_test_x, y_test, fig, ax, f"FS={','.join(sfs.k_feature_names_)}", wt)
        print('-' * 80)
```

The model was able to achieve an r^2 0.9683441146104497 for n=7, and the subset of features 6, 7, 8, 9, 11.

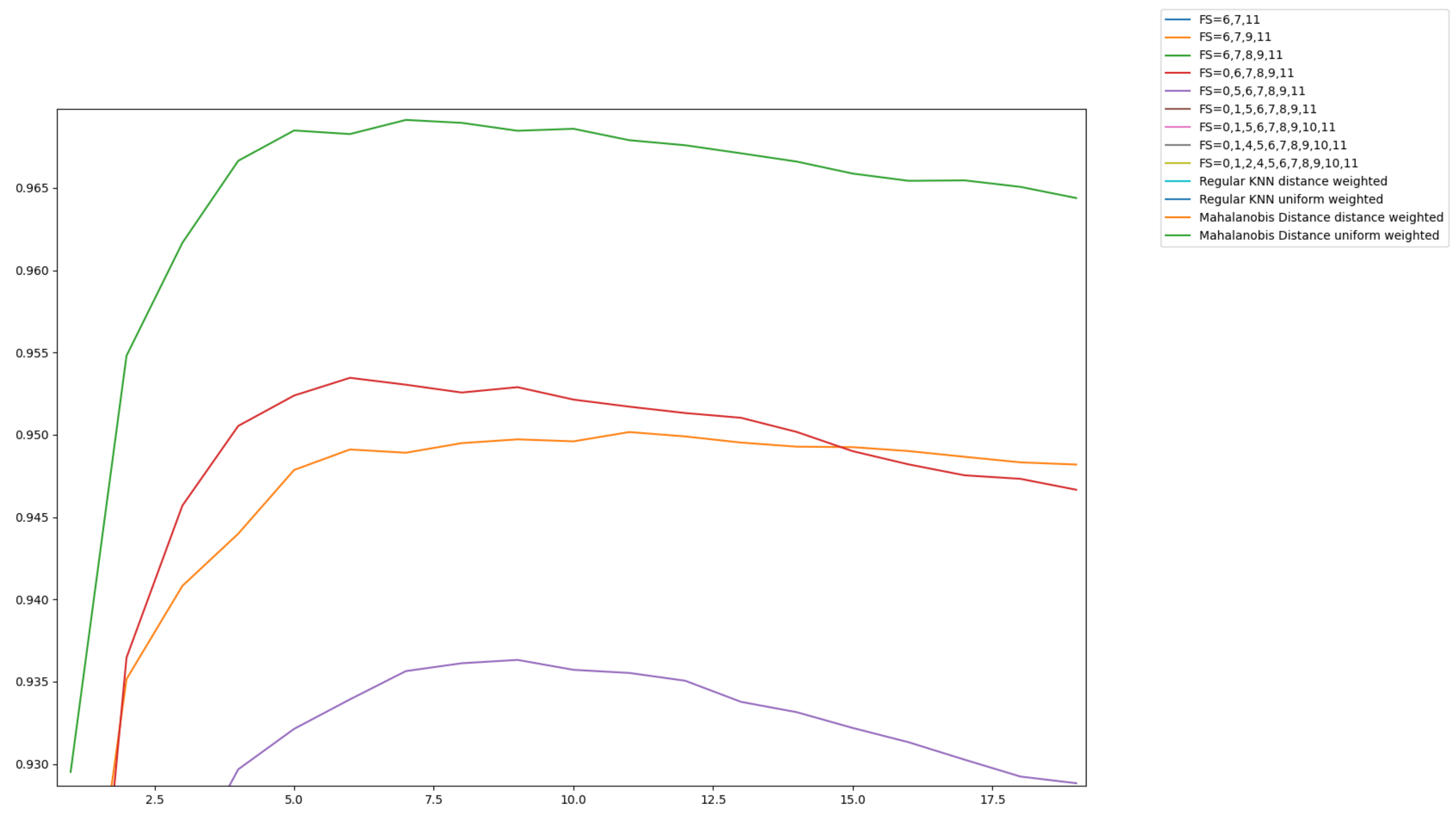
The performance of various features selected is presented in the following graph.



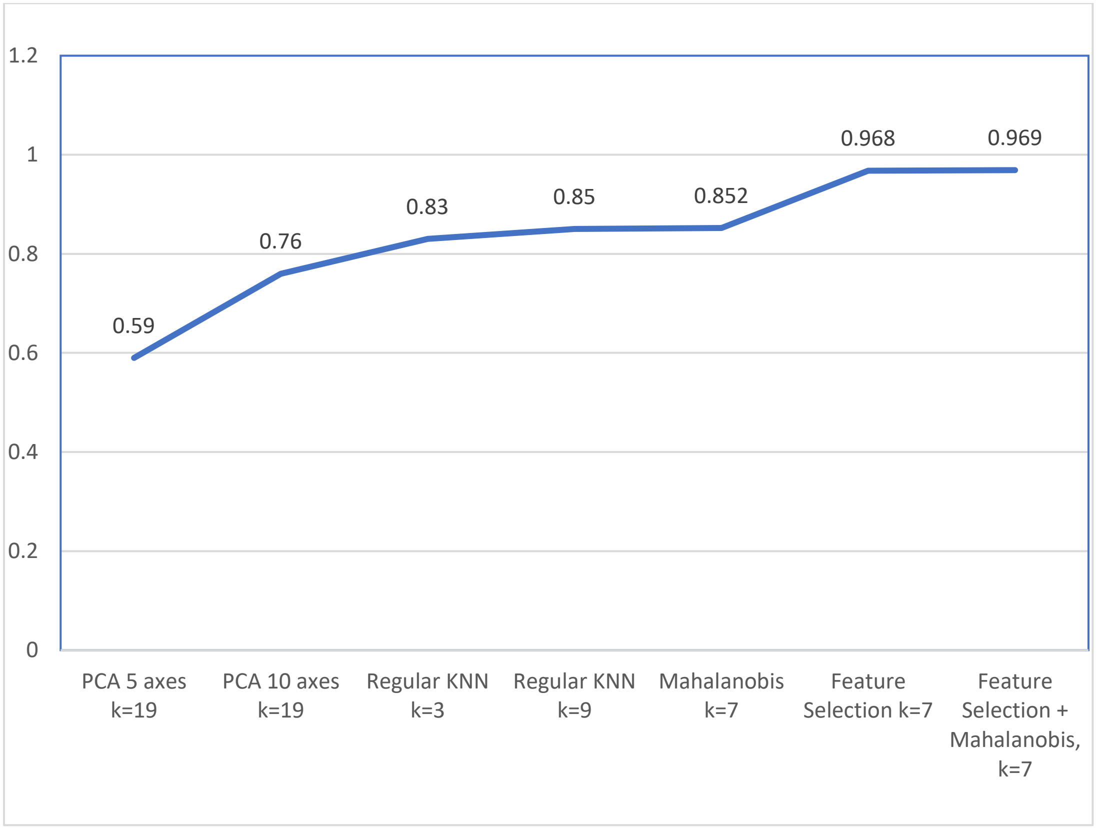
Feature selection was paired with Mahalanobis Distance to produce a slightly more accurate prediction, and was able give an r^2 value of 0.9691290819198476 at k=7, and the same set of features mentioned above.



In the above case, the optimal k was 8. The same trend was seen where increasing k led to better results in the beginning, and then plateaued and started falling.



The overall comparison of the various methods used is presented below:



Part 2A: K-means Clustering

Checklist

Generate_centroids

Generate_centroids was implemented using the following code

```
def generate_centroids(data:np.ndarray, k:int) -> np.ndarray:
    """
    Randomly choose k points as initial centroids
    """
    indices = np.random.choice(data.shape[0], k, replace=False)
    return data[indices,:]
```

assign_centroids

assign_centroids was implemented as follows

```
def assign_centroids(data:np.ndarray, centroids:np.ndarray)->list:
    arr = []
    """
    For each centroid
    """
    for centroid in centroids:
        """
        Calculate the data between this centroid and each of the data points
        This will give a kx1000 matrix
        """
        distances = calculate_distances(data, centroid)
        arr.append(distances)
    """
    Append k 1x1000 matrices to make a kx1000 matrix
    """
    arr = np.array(arr)

    """
    Find the index of the smallest centroid. If there are k centroids, then
    the distance of the array is kx1000.
    For each 1000 columns, find the row i that has the lowest value
    and that is the index of the smallest centroid, and that is what we assign
    to this point

    if there are k centroids, then all elements in this array lie between [0, k01]
    """
    indices = np.argmin(arr, axis=0)

    """
    Return a 1000x1 array
    """
    return indices.T
```

move_centroids

move_centroids was implemented as follows

```
def move_centroids(data:np.ndarray, assignments:np.ndarray, num_centroids:int)->np.ndarray:
    """
    If there are m features, and 1000 instances
    data = 1000 x m
    assignments = 1000x1
    where assignment is an array where each element a the index i
    is the number of the centroid that is closest to the element
    It takes a value between [0, k-1] if there are k centroids
    """
    new_centroids = []
    """
    For each centroid
    """
    for i in range(num_centroids):
        """
        Find the data points that are assigned to this centroid
        """
        pts_for_centroid = data[assignments == i]
        """
        Find the new centroid for all those data points
        by taking mean on each axis, this is 1xm if there are m axes, and
        append it to a list
        """
        new_centroids.append(np.mean(pts_for_centroid, axis=0))
    """
    Return all the new centroids, this is k x m
    """
    return np.array(new_centroids)
```

calculate_cost

Two versions of calculate_cost were implemented, one takes in the centroids if they are already available, and the other calculates the centroids. The first one is implemented for efficiency.

```
def calculate_cost(data:np.ndarray, centroids:np.ndarray, assignments:np.ndarray)->float:
    """
    Assignments are the indices of the centroids that are closest to each point
    It is an array 1000x1
    We need to convert this to an array containing the actual centroids.
    If there are m features, then this would be a 1000xm array
    This is easily done by indexing
    """
    closest_centroids = centroids[assignments, :]

    """
    Subtract each axis of the closest centroid from each point
    """
    square_distances = np.square(np.subtract(data, closest_centroids))
    """
    return the mean of the square of the distances
    We can do some optimization here
    Distance of each point = SQRT(SUM_OVER_i((xi - yi)^2)), i = number of features
    Mean distance = [(Distance of each point) ^ 2] / m
    We can get rid of the square root and just add all the individual differences,
    for all the axes for all the points, and return it
    """
    return np.sum(square_distances) / data.shape[0]


def calculate_cost2(data:np.ndarray, assignments:np.ndarray)->float:
    """
    This version of calculate_cost calculates the centroids in case they are
    not already calculated
    """
    centroid_nums = np.unique(assignments)
    centroids = []
    for i in centroid_nums:
        pts_for_centroid = data[assignments == i]
        thecentroid = np.mean(pts_for_centroid, axis=0)
        centroids.append(thecentroid)
    return calculate_cost(data, np.array(centroids), assignments)
    for i in centroid_nums:
        pts_for_centroid = data[assignments == i]
        thecentroid = np.mean(pts_for_centroid, axis=0)
        centroids.append(thecentroid)
    return calculate_error(data, np.array(centroids), assignments)
```

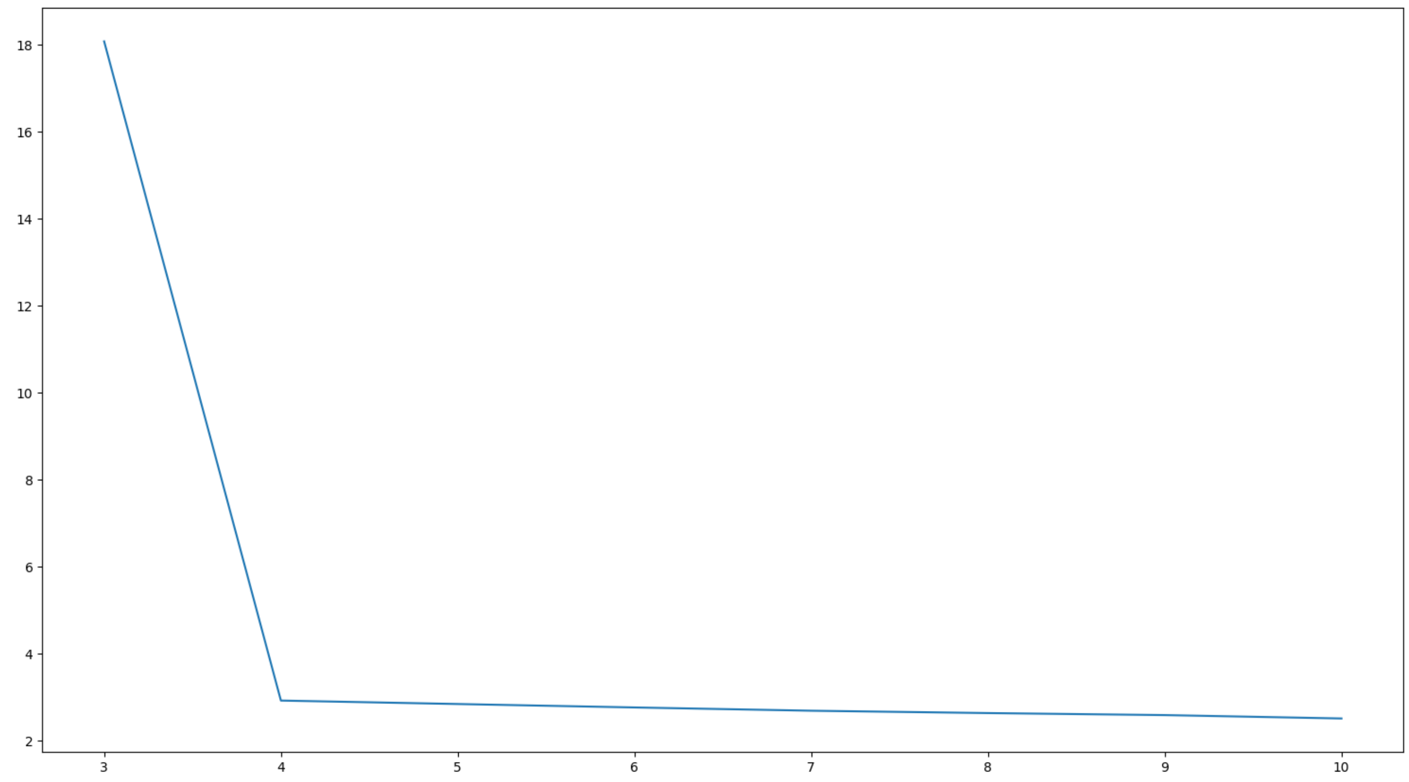
restart_KMeans

restart_Kmeans was implemented as follows

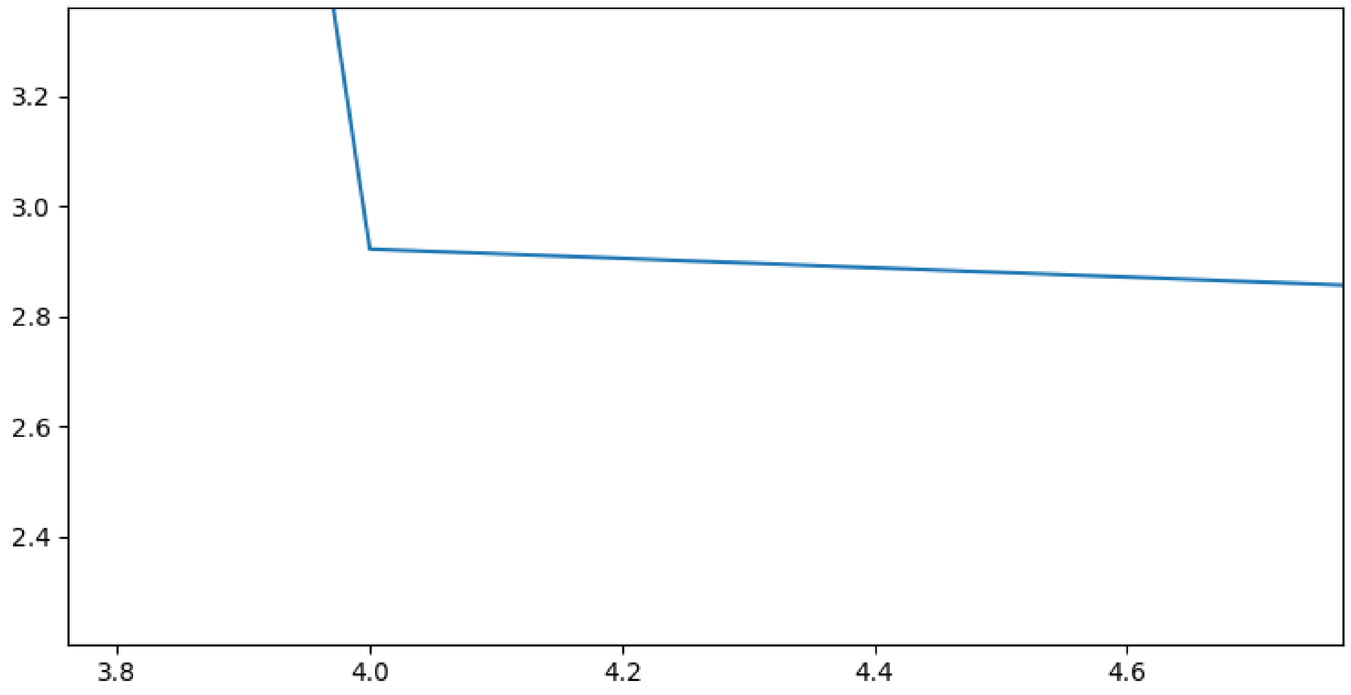
```
def should_stop(error_history:list)->bool:
    global g_stop_early
    if not g_stop_early or len(error_history) < 6:
        return False
    retVal = True
    last_errors = error_history[-5:]
    error = last_errors[0]
    for err in last_errors:
        if err != error:
            retVal = False
    return retVal



def iterate_knn(data:np.ndarray, num_centroids:int, iterations:int)->tuple:
    """
    Generate Random centroids
    """
    error_history = []
    centroids = generate_centroids(data, num_centroids)
    for kk in range(iterations):
        """
        Get the centroid each data point is assigned to, this is an integer
        """
        assignments = assign_centroids(data, centroids)
        """
        Calculate the error, we can
        use this to stop early if the error is not changing anymore
        """
        error = calculate_cost(data, centroids, assignments)
        error_history.append(error)
        """
        Move Centroids, this will give us the new set of centroids
        if there are k centroids, and m features, this is k x m
        """
        centroids = move_centroids(data, assignments, num_centroids)

        """
        If the value hasn't changed in the last n iterations, stop early
        """
        if should_stop(error_history):
            break
    """
    Calculate the final error, return this and along with the assigned
    centroids
    """
    error = calculate_cost(data, centroids, assignments)
    return error, assignments




def restart_KMeans(filename:str, num_centroids:int, iterations:int, restarts:int,
no_normalize:bool):
    data = read_file(filename)

    """
    Normalize the data
    """
    if not no_normalize:
        themean = np.mean(data, axis=0)
        stddev = np.std(data, axis=0)
        norm_data = (data - themean) / stddev

    best_error = None
    best_assignment = None
    """
    Run for N restarts
    """
    for i in range(restarts):
        if not no_normalize:
            error, assignments = iterate_knn(np.copy(norm_data), num_centroids, iterations)
            """
            The error is calculated based on the normalized data, so we must recalculate
            it based on the non-normalized data, otherwise the scale of the error
            will not match up with the non-normalized version
            """
            error = calculate_cost2(data, assignments)
        else:
            error, assignments = iterate_knn(np.copy(data), num_centroids, iterations)
        if None == best_error or error < best_error:
            best_error = error
            best_assignment = assignments
    return best_error, best_assignment
```

Results

KMeans was run for 200 iterations and 10 restarts. An elbow plot was plotted for k between 3 and 10. The elbow plot showed a distinct elbow at k=4

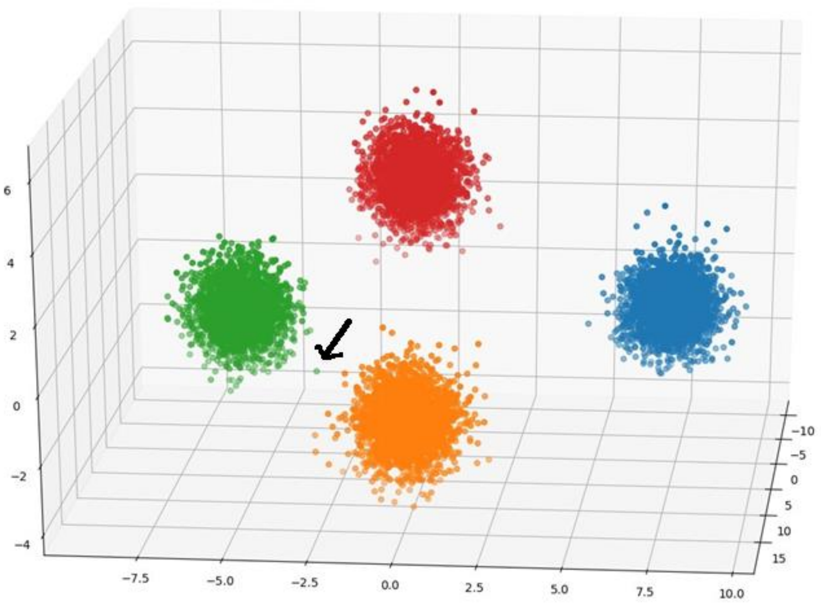


The error kept falling after k=4 as well, but the rate of improvement was very low:



At k=4, the error was 2.9221505660845826.

K=4 was selected as this gave the best output. Anything higher than k would have overfitted the data. To get an intuitive idea of the clustering, PCA on 3 axes was applied after clustering at k=4, and 4 distinct clusters were seen. While most of the points are tightly clustered, there are a few points that straddle between clusters.



A higher value of k would have assigned a different cluster to them. This would not have been desirable. One such point is marked by an arrow.

Part 2B: Mini-Batch K-Means

Regular K-Means proceeds with the following steps:

1. Generate random centroids
2. Repeat
    a. Assign-centroids: Assign all the points to the closest centroid
    b. Move-Centroids: Compute the new centroids by taking mean of all the points in a cluster
    c. Compute-distances: compute distance between each centroid and each point

In every iteration, it moves the centroid by a small amount. The problem here is that every iteration is very expensive on resources, both CPU and memory. All the points must be kept in memory, and all distances must be calculated. This is OK for small data sizes, but if the records are millions of records, this approach will not scale:

1. It will hit memory thresholds
2. Each step of moving the centroid involves calculating a lot of distances. Some of this can be mitigated by caching distances, but things may still not scale even after this.

The idea of Mini-Batch k-means was proposed by D. Sculley[10]. The idea of mini-batch k-means descent is analogous to mini-batch gradient descent with similar objectives [11]. In mini-batch gradient descent, similar problems occur – to take

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

[11] https://www.youtube.com/watch?v=l4lSUAcvHFs

one step of the gradient descent, one must process all data items. Processing all points before making a single iteration is referred to as batch mode.

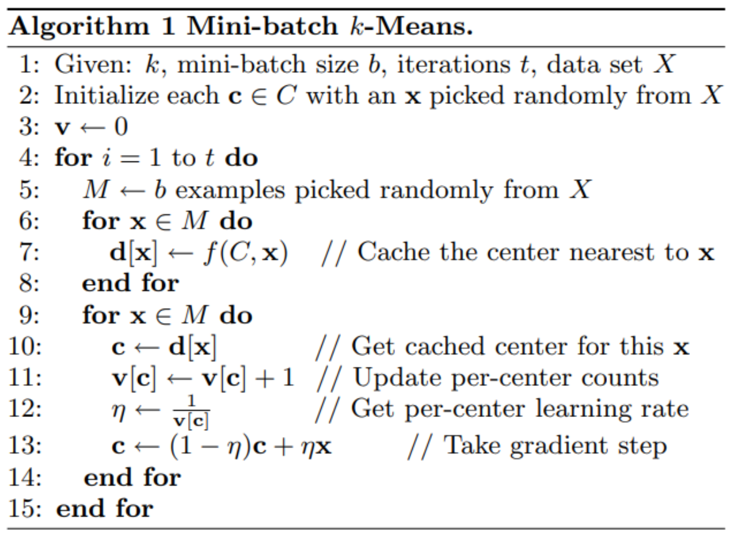
The idea behind mini-batches is to break up all the data into a number of batches, and to process each batch independently. Usually the batch size is a small power of 2, and values of 64, 128, 256 etc. are common.

The benefit of mini-batch k-means is that we can start moving the centroids after processing each mini-batch, instead of having to process each data point in the complete batch first. The result is that convergence is faster.

A second advantage of mini-batch gradient descent is that there are significant memory improvements, as we need to only hold a mini-batch in memory, instead of the full batch which could be millions of data points.

A third advantage of mini-batch gradient descent is that in each iteration, mathematical operations are working on a smaller batch, and hence the operations of sorting, summing, finding the mean, finding all distances etc. are faster.

Below is the algorithm for mini-batch k-means algorithm from D. Sculley's paper cited earlier:



Algorithm 1 Mini-batch k-Means.

```
1: Given: $k$, mini-batch size $b$, iterations $t$, data set $X$
2: Initialize each $\mathbf{c} \in C$ with an $\mathbf{x}$ picked randomly from $X$
3: $\mathbf{v} \leftarrow 0$
4: for $i = 1$ to $t$ do
5:     $M \leftarrow b$ examples picked randomly from $X$
6:     for $\mathbf{x} \in M$ do
7:         $\mathbf{d}[\mathbf{x}] \leftarrow f(C, \mathbf{x})$   // Cache the center nearest to $\mathbf{x}$
8:     end for
9:     for $\mathbf{x} \in M$ do
10:        $\mathbf{c} \leftarrow \mathbf{d}[\mathbf{x}]$   // Get cached center for this $\mathbf{x}$
11:        $\mathbf{v}[\mathbf{c}] \leftarrow \mathbf{v}[\mathbf{c}] + 1$   // Update per-center counts
12:        $\eta \leftarrow \frac{1}{\mathbf{v}[\mathbf{c}]}$   // Get per-center learning rate
13:        $\mathbf{c} \leftarrow (1 - \eta)\mathbf{c} + \eta\mathbf{x}$   // Take gradient step
14:    end for
15: end for
```

As we can see in the algorithm, in each iteration the algorithm does the following:

1. Randomly pick a mini-batch M of b examples from the data, b is the mini-batch size

2. For each data x point in M
   a. Find the centroid nearest to x

3. For each data point x in M
   a. Increment the per-centre count
   b. Get the updated learning rate for the centre (inverse of per-centre count)
   c. Move the centroid slightly towards the x as per the learning rate η. The learning rate η keeps getting smaller as more points are assigned to a centroid.

As we can see the function in line 13 is like the function used in stochastic gradient-descent with momentum which is given by:

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$

This function is key to mini-batch k-means working and converging. A good discussion of this function can be found in Andrew Ng's tutorial [12]. η in Algorithm 1 continuously keeps getting smaller as more and more points are assigned to a centroid. The effect of this is that initially, when few points are assigned to a centroid, the centroid will take large steps towards the direction of every new point. But as more and more points are assigned to a centroid, its count increases and the value of η will start becoming small. Hence if already a lot of points have been assigned to a centroid, any new point assigned to the centroid will only cause a small movement towards the centroid. This function is very important because if the movement didn't become smaller as more points were assigned to a centroid, the function would have never converged.

[12] https://www.youtube.com/watch?v=l4lSUAcvHFs

Another property of using this equation is that it *approximates a streaming average* or *exponentially weighted moving average*. The net result of this function is that it approximates calculating the mean of all the points to arrive at a new centroid.

The batch size b itself can be parametrized. In one extreme, if b = n(X), then this is the same as batch k-Means. In another extreme, if b=1, this allows us to implement streaming k-means. The Spark M-Lib library uses this method to give a routine for streaming k-means[13].

The nature of the moving average is such that it only approximates the true average. Hence mini-batch k-means will produce worse results than batch k-means. However, the loss of accuracy is outweighed by the fact that this allows us to scale k-means to millions of data points, and also allows us to implement streaming k-means.

For small data samples, however, using batch k-means is preferred because if the data sets are still manageable, then it will give us better results.

Another thing to keep in mind while implementing mini-batch k-means is that the batch size should be chosen so that it can fit within GPU memory and system memory. Also, it is common to choose a power of 2 for mini-batch size because CPUs and GPUs usually work in powers of 2.

[13] https://spark.apache.org/docs/1.5.0/mllib-clustering.html