CSCE847 Assignment 3

Introduction to Data Mining

Mohannad Alhanahnah, Jacob Shiohira

February 25, 2018

Introduction

K-means clustering is an unsupervised machine learning method that is popular for cluster analysis in data mining. The clustering aims to partition n observations into k clusters, where each instance of data belongs to the closest cluster. The distance definition used varies between Manhattan, Euclidean, Minkowski and more, but this implementation uses Euclidean distance. The problem is computationally difficult, namely it exists in the NP-hard category. This means that known algorithms are non-deterministic and do not run in polynomial-time.

K-Means Clustering Implementation

Requirements and Specifications

This implementation operates under 4 main assumptions, as specified in the assignment description:

- Assume that all the attributes are continuous variables.
    - In order to better match the clustering results from Weka's simpleKMeans algorithm, we attempt to encode the final class attribute as an integer when reading in data from the .arff file. It was found that this approach yields cluster results more closely related to Weka's simpleKMeans algorithm as opposed to ignoring the final column in each instance.
- The program must allow the number of clusters (k) to be specified as input.
- The program must allow the epsilon (change in the sum of the distances from the cluster centers) to be specified as input.
- The program must allow the number of iterations to be specified as input.

In general, k-means involves the following steps, which based on them we performed our implementation:

1. Selecting initial centroid based on the number of identified k-clusters.
2. Computing the membership of the instances, where Euclidean distance is used for computing the distance of an instance from the centroid of each cluster, and finally the instance is assigned to the closest cluster.
3. Computing the new centroids by taking the average of the assigned instances.
4. Repeating steps 2 and 3 until the algorithm terminates based on the aforementioned termination conditions.

Algorithm Design

Main Data Structure Used

The main data structure used was a dictionary. This implementation uses a dictionary named centroids that uses integers as the key and a list as the value. The integer key maps to a centroid's n-coordinates. For example, if the user specified that there should be 4 clusters, 1 maps to the coordinates of the 1st

centroid. Additionally, this implementation uses another dictionary named clusters (cluster membership/assignment) that maps the k-th integer to the instances that are closest to the k-th centroid. Even though there are two different dictionaries dealing with the coordinates of centroids and the instances closest to each centroid , it is easy to go back and forth between the dictionaries because both use the same integer to map to the values of the corresponding centroids and clusters.

Selecting Initial Centroids

The initial centroid coordinates were chosen randomly, per usual approach in k-means. However, this implementation attempts to choose a better set of initial coordinates by forcing each coordinate to be within a suitable range (min and max range of the values of a particular attribute). Each attribute in the .arff file is an explanatory variable, except the last attribute. By convention, the last attribute is a nominal attribute that corresponds to the classification of each instance. Since k-means is an unsupervised clustering algorithm, it can make use of the last column unlike any classification algorithms. Note that any non-numeric attribute is not considered, as the select_dtypes() function is used to select only numeric attributes from the following: ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']. Again, the only exception is if the last column contains nominal attributes. In that case, this implementation encodes the class data to integers and uses that in the clustering. This approach of using only numeric data allows us to find the minimum and maximum for each column and then set the centroid center somewhere between the minimum and maximum values.

There is no approach guarantees better coordinates for the initial centroids. However, we thought it to be better than just letting the initial k-centroid coordinates be in a truly random range.

Computing Cluster Membership

We used Euclidean distance to calculate the distance between instances in any given cluster and the centroids of each cluster. Distance was calculated using a function imported from scipy.spatial.distances. Cluster membership of each instance is assigned based on the minimum distance to each cluster's centroids. Membership for any given instance during an iteration is unique. Namely, an instance cannot belong to more than one cluster at any given point. The minimum distance requirement between an instance and cluster centroids help satisfy this requirement.

Computing New Centroid Clusters each Iteration

The coordinates of the new centroids are computed based on the average of each attribute of all instances that belong to a particular cluster. Each of the n coordinates of every cluster are computed this way. This process is done in step 2 of the k-means algorithm outlined in Requirements and Specifications.

Computing the termination

The k-means algorithm will terminate under either of the following two conditions: the number of iterations is surpasses the number of max_iterations specified by the user, or the change in the total sum of the squares of the distances (SSE) falls below epsilon. This is calculated as follows. The (numpy.linalg.norm) function is used to calculate the norm of the differences for each instance in a cluster and those distances from the cluster's centroid. Then, all distances are summed up for all clusters to yield

the SSE. Finally, the difference between the new SSE and the SSE from the previous iteration is computed. If the difference is lower than the identified epsilon value, then our implementation termination the algorithm, and generates the final clustering results.

Plots

This implementation was tested on a number of .arff data sets, which are all included in the Data folder in the repository. Results presented in figures 1,4,5,6, and 9 are based on the Data/iris.arff data set. Then, figures 7 and 8 are based on the Data/winewhitequality/train.arff data set. Results presented in figure 2 are based on a number of data sets, all of which are included in the Data/dimensions folder. Lastly, results presented in figure 3 are based on the Data/shuttle/train.arff data set.

Runtime of Algorithm as a function of Number of Clusters

Figure 1 shows the runtime in seconds of our implementation, where the range the number of clusters is 1-50. Obviously, it is expected the runtime will be increased when the number of clusters is increased. The Data/iris.arff data set was used for this testing.

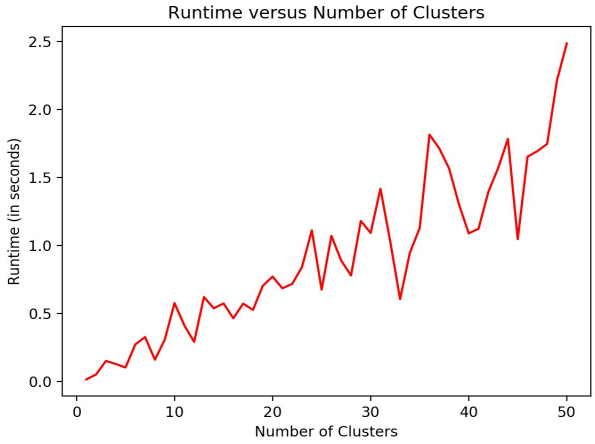

Fig. 1 Runtime of Algorithm as a function of number of clusters

Runtime of Algorithm as a function of Number of Dimensions

Figure 2 shows the runtime in seconds of our implementation, where the range the number of dimensions is 0 to 800. One can see that the runtime increases as the number of dimensions is increased, but it does not increase in a linear fashion. The Data/dimensions folder in the repository contains all the .arff files used for this testing.

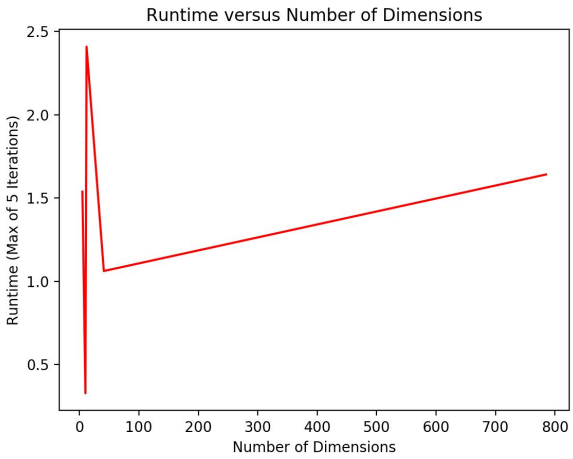

Fig. 2 Runtime of Algorithm as a function of number of dimensions

Runtime of Algorithm as a function of Size of Dataset

Figure 3 shows the runtime in seconds of our implementation, where the range the number of instances is 5,000 to 40,000 in increments of 5,000 instances. One can see that the runtime increases as the number of instances is increased in a near linear fashion. The Data/shuttle folder in the repository contains the train.arff files used for this testing.

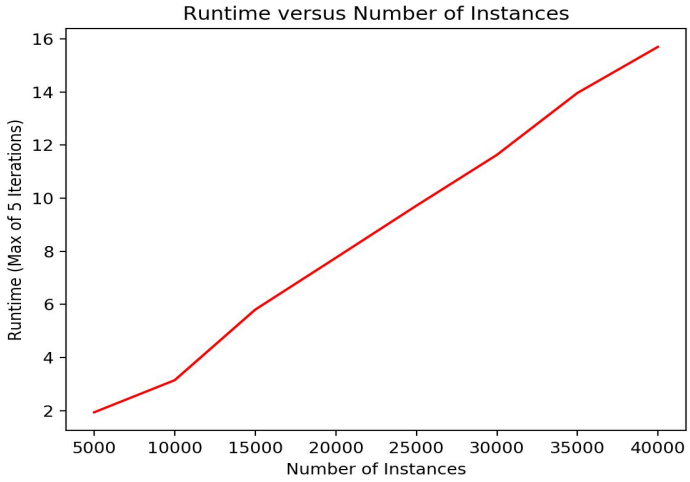

Fig. 3 Runtime of Algorithm as a function of number of instances

Goodness of clusters versus Number of Clusters

Figure 4 depicts the goodness of our k-means implementation. As expected, there is a much larger SSE when K is small. SSE is biased towards a larger value of K, as SSE goes to zero as K approaches the number of transactions. The optimal value of K in k-means clustering is generally determined by the elbow principle when plotting SSE versus number of clusters. At some point, the amount that SSE decreases for the next value of K drops off immensely, creating an elbow effect on the grap. The value of K where this dramatic change happens is often seen as the optimal value for K.

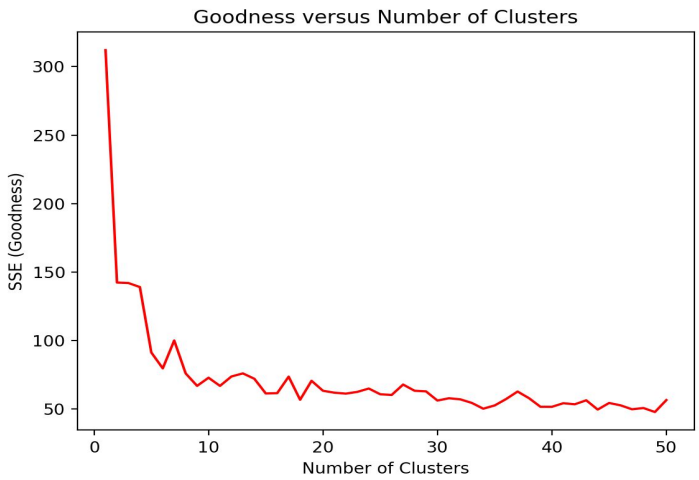

Fig. 4 Clustering Goodness as a Function of K

Comparison with Weka

Testing on the iris.arff dataset had a number of advantages. Namely, there are only 5 attributes for each instance of data. Thus, the algorithm was only ever dealing with 5-dimensions, making it much easier to comprehend what was happening. Further, the dataset is pretty easily separable on any 3-axis combination of the first 4 explanatory variables, which means that we could easily plot the visual plot to make sure that clusters were converging to where we would expect. Additionally, on average, the clusters converged to the locations one would expect even with totally random beginning centroid coordinates. Following is a brief discussion between Weka's results and the results of this implementation.

Performance

Weka's simpleKMeans is really fast regardless of number of clusters, K, specified or number of transactions. This varies differently from this implementation because as the number of transactions increased, the runtime of the algorithm also increased exponentially. This was especially seen when testing with the shuttle/train.arff dataset, which has 3,429 transactions. For a k value of 7, weka could converged after 45 iterations in 0.21 seconds whereas this implementation struggled to converge and took much longer to process. It ultimately stopped at the specified max_iteration value of 45 iterations in 14.77 seconds.

We were able to greatly improve the performance of the k-means clustering algorithm by using linear algebra functions to calculate the norm of the difference between two vectors. Initially, distance calculations were done using Euclidean Distance from the scipy.spacial.distance Python package. As the values of the attributes in the data set grew larger in magnitude, the Euclidean Distance function takes a longer time to do the computation. When we tried to use the norm function from the np.linalg package after taking the difference between two vectors, we found that the distance output was the same, but the time taken to do the calculation significantly decreased. For example, using the Euclidean Distance function mentioned above yielded an algorithm runtime of 33.5 seconds on the

Data/winewhitequality/train.arff data set. By using linear algebra instead, the runtime of the algorithm was decreased by over half to 14.6 seconds.

Normalization:

We observed that Weka normalizes the error in the simpleKMeans algorithm (this parameter is under Euclidean distance options). This implementation will perform normalization of the dataset if the corresponding flag is specified as a CLI argument (as an additional feature of our k-means implementation), which assisted us to compare our results to Weka's results. For more information on the command line parameters to run, see the README. When Weka performs normalization, it generates a much smaller SSE than our SSE value without normalization. However, Weka's output specifies all coordinates and clustering attributes in terms of the unscaled dataset. Figures 5 and 6 illustrate the implication of not normalizing the data. When the dataset is normalized, the overall runtime of the k-means algorithm decreases because it is not performing distance calculations on arbitrarily large numbers but rather on floating point numbers ranging from 0 to 1.

```
kMeans
======

Number of iterations: 5
Within cluster sum of squared errors: 87.32963569427773

Initial starting points (random):

Cluster 0: 6.1,2.9,4.7,1.4,Iris-versicolor
Cluster 1: 6.2,2.9,4.3,1.3,Iris-versicolor
Cluster 2: 6.9,3.1,5.1,2.3,Iris-virginica

Missing values globally replaced with mean/mode

Final cluster centroids:
                                        Cluster#
Attribute               Full Data             0               1               2
                          (150.0)          (51.0)          (50.0)          (49.0)
=============================================================================
sepallength                5.8433          5.9157           5.006          6.6224
sepalwidth                  3.054          2.7647           3.418          2.9837
petallength                3.7587          4.2647           1.464          5.5735
petalwidth                 1.1987          1.3333           0.244          2.0327
class                 Iris-setosa Iris-versicolor     Iris-setosa  Iris-virginica

Time taken to build model (full training data) : 0 seconds

=== Model and evaluation on training set ===

Clustered Instances

0       51 ( 34%)
1       50 ( 33%)
2       49 ( 33%)
```

Fig. 5 Weka's output for iris.arff with k=3 (normalization is not enabled)

```
kMeans
======

Number of iterations: 9
Within cluster sum of squared errors: 87.31

Initial starting points (random):
Cluster 1: 6.1,2.9,4.7,1.4,1
Cluster 2: 6.2,2.9,4.3,1.3,1
Cluster 3: 6.9,3.1,5.1,2.3,2

Final cluster centroids:
Attribute       Full Data       1       2       3
                (150)           (51)    (50)    (49)
============================================================
sepallength     5.843           5.916   5.006   6.622
sepalwidth      3.054           2.765   3.418   2.984
petallength     3.759           4.265   1.464   5.573
petalwidth      1.199           1.333   0.244   2.033
class           1.0             1.02    0.0     2.0

Time taken to build model (full training data) : 0.07633 seconds

Clustered Instances
1       51 (34.0 %)
2       50 (33.33 %)
3       49 (32.67 %)
```

Fig. 6 This implementation's output for iris.arff with k=3 (normalization is not enabled)

Another comparison was done between Weka (output shown in figure 7) and this implementation (output shown in figure 8) with the winequalitywhite/train.arff dataset using the same initial centroids. It was desirable to compare the output of Weka and this implementation. It can be seen that the clustering results in almost exactly the same clustering with very comparable sum of squared errors.

```
kMeans
======

Number of iterations: 49
Within cluster sum of squared errors: 702334.0654246034

Time taken to build model (full training data) : 0.17 seconds

=== Model and evaluation on training set ===

Clustered Instances

 0       247 (  7%)
 1       382 ( 11%)
 2       368 ( 11%)
 3       407 ( 12%)
 4       352 ( 10%)
 5       297 (  9%)
 6        61 (  2%)
 7        69 (  2%)
 8       349 ( 10%)
 9       229 (  7%)
10       248 (  7%)
11       420 ( 12%)
```

Fig. 7 Weka's output for winequalitywhite/train.arff with k=12 (normalization is not enabled)

```
kMeans
======

Number of iterations: 53
Within cluster sum of squared errors: 702920.499

Time taken to build model (full training data) : 30.34345 seconds

Clustered Instances
1       247 (7.2 %)
2       382 (11.14 %)
3       368 (10.73 %)
4       407 (11.87 %)
5       351 (10.24 %)
6       292 (8.52 %)
7       61 (1.78 %)
8       69 (2.01 %)
9       349 (10.18 %)
10      235 (6.85 %)
11      249 (7.26 %)
12      419 (12.22 %)
```

Fig. 8 Weka's output for winequalitywhite/train.arff with k=12 (normalization is not enabled)

Generating Initial Centroids

We observed that Weka uses a seed value in the simpleKMeans parameters. The seed value is used in generating a random number which, in turn, is used for making the initial assignment of instances to clusters. In general, k-means can be quite sensitive to how cluster centroids are initially assigned based on the distribution of the data. Thus, it is often necessary to try different initial centroid values and evaluate the results. Weka can reproduce consistent results because of the use of a seed variable, which produces pseudo-random functions to behave deterministically. This is helpful so that a user can replicate results when he or she encounters a good clustering. The Weka approach influenced this implementation because this implementation initially generated irreproducible centroids each time the k-mean clustering algorithm ran. However, it is of course helpful to be able to reproduce results, especially in development. So, this implementation includes the option to specify a seed value, which is used when generating the location of initial cluster centroids.

Figure 8 presents a 3-D plot for the iris.arff dataset after running the k-means clustering algorithm in this implementation over the dataset. The value of K was set to 3, which corresponds to the 3 different breeds of flowers in the dataset: Iris-setosa, Iris-versicolor, and Iris-virginica.

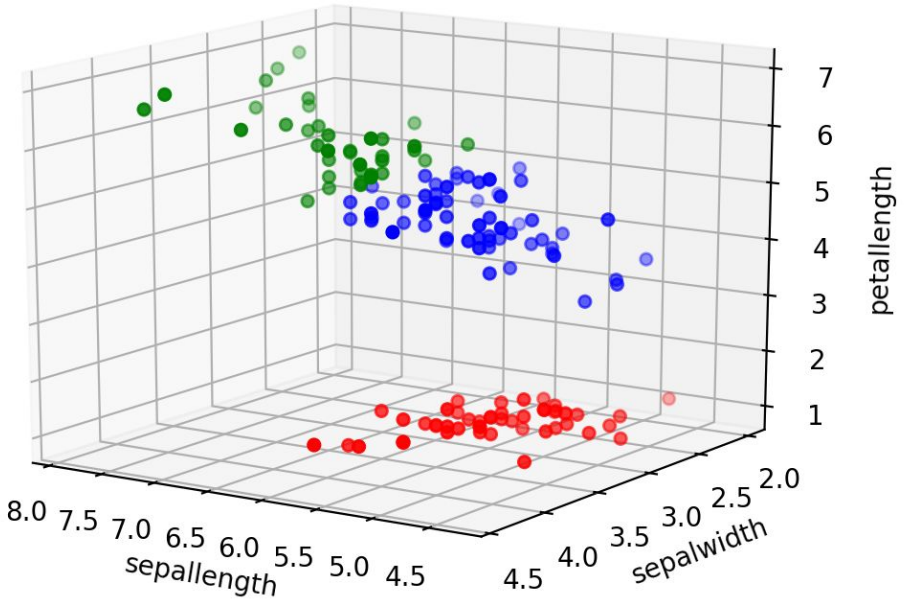

Fig. 9 Weka's output for winequalitywhite/train.arff with k=12 (normalization is not enabled)

Conclusion

This report describes our implementation for k-means algorithm, and highlights the techniques that have been used for developing the algorithm. It also makes a comparison between the results of our implementation and Weka's results, and shows that the results are very similar. Further investigation into performance of this implementation of the k-means clustering algorithm include the effect on clustering results by removing correlated columns in the dataset and normalizing the values in the dataset. By removing correlated attributes of the dataset might decrease overall runtime while still yielding "good" cluster results. As dimensions increase and values in the dataset increase in magnitude, computing the Euclidean distance become an increasingly costly operation. By normalizing the dataset, relationships between the data points in each instance are preserved while greatly decreasing the cost of computing distance.