Machine Learning - MSAI 349

Homework 2

October 26th, 2021

Professor
Dr. David Demeter

Students
Aleksandr Simonyan
Dimitrios Mavrofridis
Donald Baracskay
Xingbang Liu
Ana Cheyre

Answer for Question 1:

The two functions have been implemented successfully in two different python files, each one named after its function. Thus, we have the euclidean function file and the cosine similarity file which are being used by the KNN and KMeans algorithms by simply calling the functions from each file.

Answer for Question 2:

Both KNN and KMeans ran for a prohibitively long period of time (especially KMeans). We attempted a number of strategies to reduce the data size, most of which were dimensionality reduction. Our simpler and more accurate strategy simply removes columns where all points have the value “0.” Because all values are the same, the columns effectively contain no useful information. This reduces dimensions down to around 600. Our other strategy takes the variances of each dimension and removes the columns which fall below a threshold (in the current case this threshold is determined by the standard deviation of the variances). This drops dimensions down to roughly 300, and thus allows KMeans and KNN to run much faster, although admittedly with an accuracy drop of around 4%. We have created multiple files in order to break down the project into smaller and easier to identify and debug parts. In order to change the dimensionality reduction methods, one has to visit the global_variables file and choose the preferred reduction method.

KNN was developed by following the given template where we receive the training dataset as an input and the testing set (query) in addition to the required distance or heuristic function to use. When a prediction is expected, the k-most similar elements for a given element of the training dataset are generated. A custom function is run to determine whether the Euclidean distance or the cosine similarity will be used to measure the similarity between the given elements. Furthermore, after generating the neighbors, based on the metric (Euclidean or Cosine) we pick the most common outcome. As a result, a list of the generated labels and correct predictions is returned in order to generate the 10x10 confusion matrix. A separate file with the name confusion_matrix contains a function that is responsible for taking the results of KNN and printing out a confusion matrix in an appropriate format. In order to identify the most optimal number of neighbors for each case, we tested the algorithm by supplying the training dataset in both the train set and the query. We then generated a for loop which tested all the possible numbers that k could receive. Finally, we analyzed the results and picked the number of neighbors equal to 5 which generated the highest accuracy. Eventually, we tested the algorithm for its accuracy by providing the test set into the query argument. As a result, KNN achieves very high accuracy. The results can be viewed in the table below. It properly identifies the digits at worst 86% of the time and this is with some data being removed to increase speed.

Reduction Function	Euclidean	Cosine Similarity
Remove Zeroes	90%	92.5%
Remove Low Variance	86%	92.5%

Figure 1. Accuracy of Approaches

17	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
0	**27**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**2**
1	**0**	**15**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
0	**0**	**0**	**17**	**0**	**3**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**23**	**0**	**0**	**0**	**0**	**1**
0	**0**	**0**	**0**	**0**	**8**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**1**	**1**	**12**	**0**	**2**	**2**
0	**0**	**2**	**0**	**0**	**1**	**0**	**23**	**1**	**1**
0	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**14**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**16**

Figure 2. Confusion Matrix Using the Euclidean Function

17	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**
0	**27**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**19**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
1	**0**	**0**	**16**	**0**	**1**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**22**	**0**	**0**	**0**	**0**	**2**
0	**0**	**0**	**0**	**0**	**11**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**1**	**13**	**0**	**0**	**1**
0	**0**	**0**	**0**	**0**	**0**	**0**	**23**	**0**	**0**
0	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**19**	**0**
0	**0**	**0**	**0**	**3**	**0**	**0**	**1**	**0**	**18**

Figure 3. Confusion Matrix Using the Cosine Similarity Function

Answer for Question 3:

The KMeans algorithm used is true to form. We label points based on either Euclidean distance or cosine similarity, and then find the average location of each centroid's associated points. We used two hyperparameters: k and max_iterations. Max_iterations is simply the number of times we loop through moving the centroids, and k is the number of clusters, which in turn makes it the number of centroids. Perhaps one unique thing about our algorithm is that we define centroids on top of random points that already existed in our dataset. This ensures that the centroids will always fall within the area where points are located even if the data is transformed into binary. *To avoid all kinds of bias, centroids are defined by random initialization. This avoids predefining a certain area for the point, but can also lead to a less optimal value.*

When compared to KNN, the results of KMeans are somewhat disappointing. This is, however, somewhat expected as KNN is a supervised learning algorithm and KMeans is a very similar but unsupervised algorithm without access to ground truth. That being said, the euclidean distance-based KMeans perform fairly well. It converges typically at around 30 - 40 iterations and is able to form somewhat "pure" clusters. For example, most of the "ones, " around 92%, were within the same cluster. However, digits like "fours" and "nines" are less pure but still appear in high frequency in two clusters. Cosine Similarity performs quite a bit worse and this can likely be explained by the sparseness of the data. Even with more than ⅔ of the dimensions removed, there are still 283 dimensions. This means that the points exist in a space that has 255 ^ 283 possible locations, and there are but a mere 2000 points. Cosine similarity performs poorly in sparse spaces. Our cosine similarity-based KMeans has not ever converged hitting the max_iteration threshold every time. It also produces fewer "pure" clusters, possibly

even having clusters with no associated points. However, it does seem to cluster together similar-looking digits, such as “sevens” and “ones” or “fours” and “nines.”

The kmeans.py is our maid code for K-means*. However,* In the k_means_alex code, we have reimplemented the k-means but in that case, we also added new iterations based on new initializations. Every time, we get the actual values of the clusters, and based on the accuracy function that we implemented we calculate the best clusters and return only them. Even though this approach directly contradicts the idea of unsupervised learning, we have implemented it in order to show how significantly initialization can impact final results. After many iterations with various random initializations, we can get a very high final accuracy score (accuracy is calculated as the ratio of the count of the number that occurs the most to the overall count of the number inside the clusters, for example, if cluster 1 has 10 points 8 of which are 1-s, the accuracy will be 80%). Our average accuracy exceeded after reinitialization and keeping the best values 70%, with many clusters having only pure numbers (for example cluster 1-has only 1-s, 2 has only 2-s, etc).

Answer for Question 4:

Collaborative filtering uses cosine similarity or Pearson correlations to compare users to each other. KNN can be used to label (label could be customer names) a new movie based on the labeled movies. The new movie will be compared with each labeled movie to calculate the distance. The distance will be used to vote for the most likely label that should be applied to this movie, and the likely rating could be estimated by calculating the average of the relevant distance.

To implement a collaborative filter, we would use the methodology outlined in the following pseudo-code:

```
collaborative_filtering(input -> user_vectors_list):
    “””
    This pseudocode is a user_based CF system that recommends similar users.
    Sample user_vectors_list:
    [
    [‘user_id’, ‘item_id’, ‘item_rating’, ‘other_features’]
    … other users
    ]
    ”””
    users_pair_distance = []
    for i in user_vetor_list:
        for j in user_vetor_list:
            numerator = sum ( [a * b for a, b in zip(i , j)])
            Sum_i = sqrt ( sum(item*item for item in i) )
            sum_ j = sqrt ( sum(item*item for item in j) )
            # We invert the sign of cosine distance
            Cosin_distance = - numerator / (sum_i * sum_j )
```

```
                heapq.heappush(users_pair_distance, [(i[0], j[0]), cosine_distance] )
    # Then we pop the first pair which corresponds to the highest cosine distance, that is the most similar
    Similar_users_pair = heapq.heappop(users_pair_distance)

    return similar_users_pair
```

Answer for Question 5:

To develop the model for the Soft K-Means classifier, we started by randomly initializing the centroids. Then each data point was assigned a probability value for each cluster. We built a “HiddenMatrix” that contains for each of the points the probability of belonging to each of the clusters. The same as before, we used two hyperparameters: k and max_iterations, k for the number of clusters, which equals the number of centroids, and max_iterations for the number of times the cycle is executed.

We see that the Soft K-Means Confusion Matrix is distributed a little more evenly between clusters compared to the K-Means matrix because when we assign the label to each point it is not binary based but probability-based. The average standard deviation between clusters in K-Means is 5.8, whereas for Soft K-Means it is 3.8, which proves to be a more homogeneous group. This shows that for binary classifications, where each element must belong to only one group, it is preferable to use K-Means, on the other hand, if the cluster is a fuzzy cluster, where variables have a high level of overlap, it is preferable to use Soft K-Means, which can be very good for Image Segmentation.

11	**0**	**0**	**7**	**0**	**7**	**0**	**0**	**0**	**0**
1	**10**	**1**	**1**	**1**	**0**	**0**	**5**	**0**	**0**
0	**0**	**17**	**0**	**1**	**0**	**0**	**9**	**0**	**0**
0	**0**	**1**	**19**	**0**	**4**	**0**	**0**	**0**	**0**
6	**1**	**0**	**0**	**5**	**0**	**1**	**0**	**0**	**0**
5	**0**	**1**	**6**	**0**	**10**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**3**	**0**	**15**	**0**	**0**	**0**
1	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**4**	**5**
0	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**13**	**2**
1	**1**	**0**	**1**	**2**	**0**	**1**	**2**	**1**	**12**

Figure 4. Soft K-Mean Confusion Matrix Using the Euclidean Function