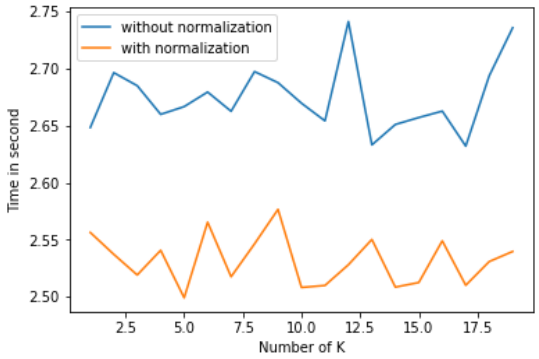
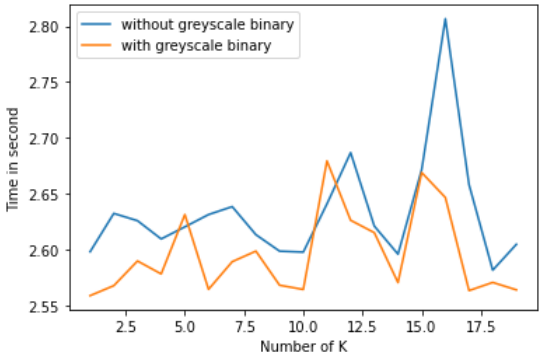
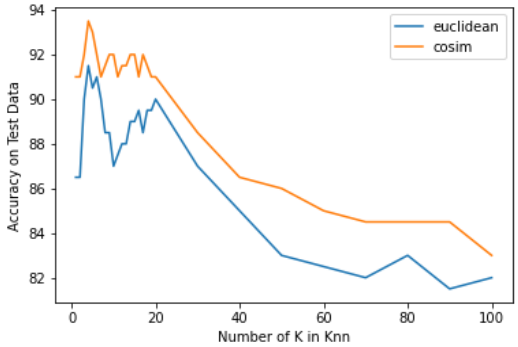
1. (1,0 points) Implement functions to compute Euclidean distance and Cosine Similarity between two input vectors a and b. These functions should return a scalar float value. To ensure that your functions are implemented correctly, you may want to construct test cases and compare against results packages like numpy or sklearn.

- We have tested our distance functions (*euclidean(a,b) and cosim(a,b)*) by using the euclidean_distances(a,b) and cosine_similarity(a,b) functions from sklearn.metrics.pairwise.
- We have built our cosine similarity function (*cosim(a,b)*) for a 1d array because it was sufficient for both the knn and k-means tasks. The arrays being tested in this function (a and b) are 2d arrays with one row. Hence the arguments of our cosim function are a[0] and b[0] to make them as 1-d arguments.
- We have rounded the results from both the functions to 5 decimals for the ease in comparison.
- The results from both the functions are successfully matching the results from the sklearn functions.
- We have included the function that does the comparison (*distance_test*) at the beginning of the main() function so it will print in the beginning whether the metrics are matching.

2. (4.0 points) Implement a k-nearest neighbors classifier for both Euclidean distance and Cosine Similarity using the signature provided in starter.py. This algorithm may be computationally intensive. To address this, you must transform your data in some manner (e.g., dimensionality reduction, mapping grayscale to binary, dimension scaling, etc.) -- the exact method is up to you. This is an opportunity to be creative with feature construction. Similarly, you are free to select your own hyper-parameters (e.g., K, the number of observations to use, default labels, etc.). Please describe all of your design choices and hyper-parameter selections in a paragraph. Once you are satisfied with performance on the validation set, run your classifier on the test set and summarize results in a 10x10 confusion matrix. Analyze your results in another paragraph.







For the design, we have used the following attributes and functions:

Attributes:

`distance_measure:` Store the way of calculating distance, 'euclidean' or 'Cosim'.

`n_neighbors:` the number of neighbors to use for prediction

`aggregator:` indicates how to aggregate label of n_neighbors, the default is 'mode'

`train_features:` storge all the training data

`train_targets:` storage all the corresponding label of the training data

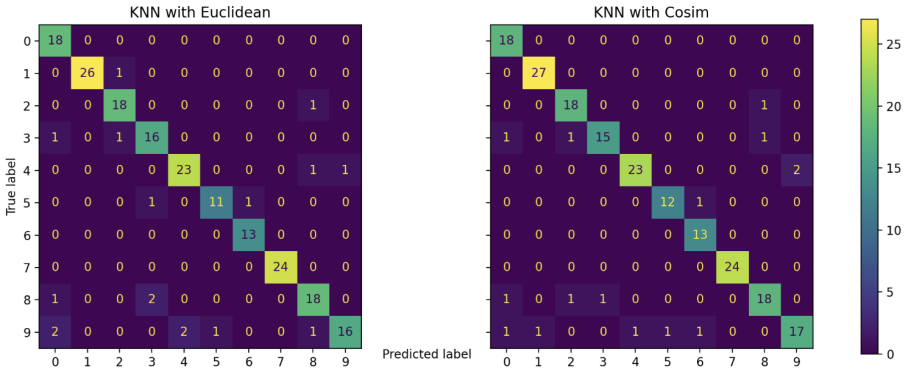
Functions:

`fit(feature, targets):` training the model by the given feature. (label only used for storage.)

`predict(features):` predict the label by the given feature. (will use the training label given by the fit().label)

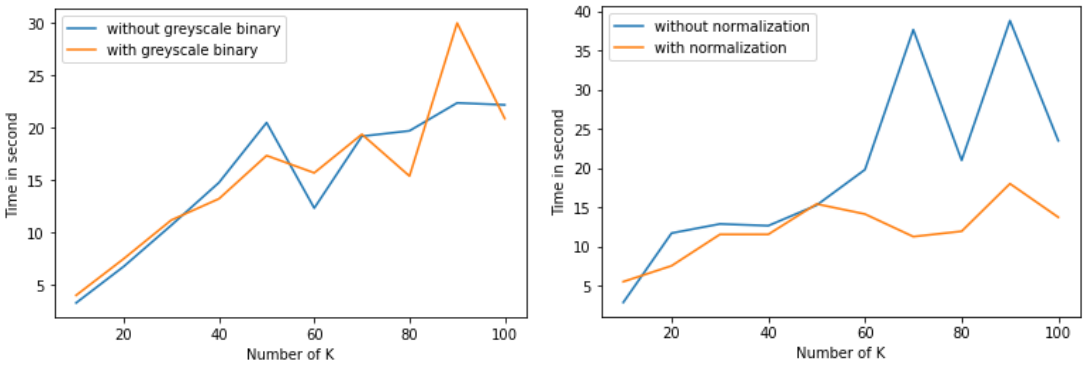
Since there is actually no training process in knn, we use `fit()` to just initialize two parameters. All the work is done in the `predict()` function. By comparing each test object with all the training data, we get a list of label with different distances (specified by the parameter `distance_measure`), and it sorts this list by distance, and extracts the top k (specified by the parameter `n_neighbors`) labels, and the mode of these k label will be assigned to this test object. It repeats until all the test objects are assigned and returns the list of these labels.

To address the problem of computational intensity, we have applied both mapping grayscale to binary as well as normalization. The results showed that both mapping grayscale to binary and normalization decrease the run time to some extent. In terms of hyper-parameters, we iterate K through 1 to 100. As we can observe from the above figure, when K is about 3 to 5, the accuracy is over 90 percent.



From the confusion matrix, we can see that KNN is doing almost equally well on identifying numbers 0-9. Also, calculating the distance using cosim seems slightly better than using euclidean.

3. (4.0 points) Implement a k-means classifier in the same manner as described above for the knearest neighbors classifier. The labels should be ignored when training your k-means classifier. Describe your design choices and analyze your results in about one paragraph each.



For the design, we have used the following attributes and functions:

Attributes:

`distance_measure:` Store the way of calculating distance, 'euclidean' or 'Cosim'.

`centers:` Store the current cluster center of this kmeans.

`nmeans:` Store the K of the Kmeans.

`prediction:` Store the prediction of each training data.

`train_label:` Store the true training label, used for the test data prediction.

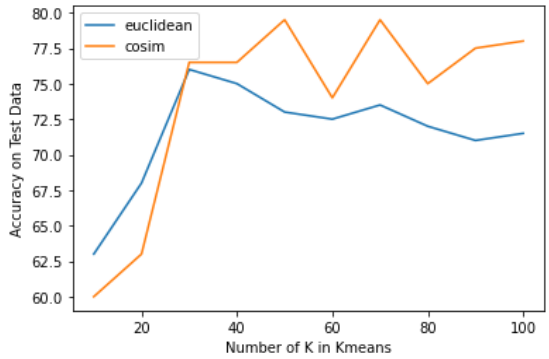
Functions:

`fit(feature, label):` training the model by the given feature. (label only used for storage.)

`predict(feature):` predict the label by the given feature. (will use the training label given by the fit().label)

To address the problem of computational intensity, we have applied both mapping grayscale to binary as well as normalization. The results showed that mapping grayscale to binary does not change the runtime much, but in case of normalization, it decreases the runtime only at very high k values. In terms of hyper-parameters, we iterate K through 10 to 100. As we can observe from the above figure, when K is about 50 and 70, the accuracy is about 79 percent.

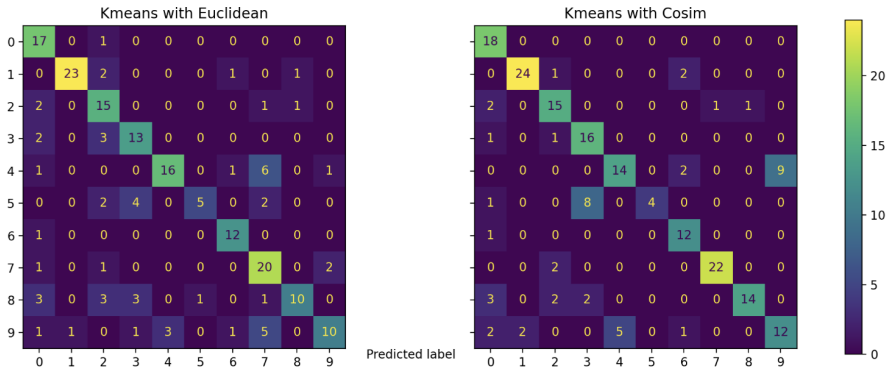
To be specific on the `predict()`. Since it is unsupervised training, there is no label of centers for us to test the accuracy. To solve this problem, we first label each center by its index. Then for each center, we find all data belonging to it, count their true label, add the most frequent true label and the label of center to a dictionary. Therefore, we got a map, mapping the center index to the most frequent label. This most frequent label is considered as the label of the cluster with that center. Once that is done, the accuracy can be the same function as for knn i.e count the percentage of test cases that are being rightly clustered into the clusters whose labels are same as that of the test cases' true labels.



For the result:

Here is the figure of how accuracy changed over different K. As the K increases in K-means, the accuracy increases. In the beginning, it seems weird to have a K not equal to 10 because we only have 10 digits. However, let's think in this way, for different people, the way they write digits may be totally different. So, when we add more centers, we are actually adding more centers for categorizing the way of writing the same digit.

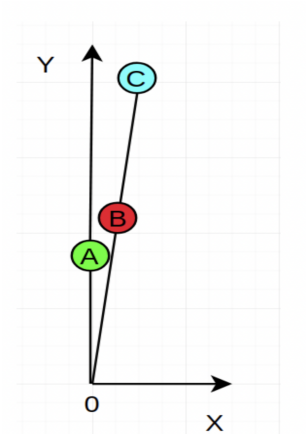
Also, with the increasing number of K, the cluster becomes smaller and smaller. When the cluster is small to some point, there may only be 3-4 points belonging to each center. In this case, it seems like K-means become a "KNN" with K = 3 or 4. In our way of making predictions, using the majority label for a cluster, it is the same as we get with the 3-4 nearest neighbors' labels.



From the confusion matrix, we can see that either using euclidean or cosim, 5 can be easily misclassified as 3, this situation is more serious in cosim than in euclidean, the same feature that 5 and 3 is sharing is they all twist three times. For euclidean, only 4 and 9 have a chance to be misclassified as 7, that might be something that is related to the dataset. While in cosim, 9 has a chance to be misclassified as 4 and vice versa. That may be caused by the fact that both 4 and 9 have a hole on the upper half and a tail on the lower half. It is interesting to see that the misclassifications between 4 and 9 only happened while using cosim and it is bidirectional. There might be some difference between the underlying feature extraction of euclidean and cosim that can be explored in the future.

4. (1.0 points) Collaborative filters are essentially how recommendation algorithms work on sites like Amazon ("people who bought blank also bought blank") and Netflix ("you watched blank, so you might also like blank"). They work by comparing distances between users. If two users are similar, then items that one user has seen and liked but the other hasn't seen are recommended to the other user. What distance metric should you use to compare user to each other? Given the k-nearest neighbors of a user, how can these k neighbors be used to estimate the rating for a movie that the user has not seen? In about one paragraph describe how you would implement a collaborative filter, or provide pseudo-code.

- Our idea is that we can use a combination of cosine similarity and euclidean distances to find similar users. Cosim does a good job of finding the differences in “tastes” of the users by calculating the angle between the vectors of movies they have watched (in the case of netflix). But if one user rates the movies high and the other rates them low, then cosim won't be able to capture that because it only sees the angle.
- In the image below, A,B and C are vectors of three users who have watched movies X and Y. The length of the vectors determine the respective ratings they have given to X and Y. The cosim between (A,C) will be the same as (A,B) but clearly C has given a much higher rating to Y than A. So B is a closer neighbor to A than C. Cosim is not able to capture that.



- So in that case euclidean might help us calculate the distance between how much the customers are rating.
- The method is to calculate the cosim between people first and for the people with very less cosim (below a certain threshold), we calculate the euclidean distance to see the difference in their ratings and sort the distances based on the euclidean. Once the k-neighbors are selected based on the above criteria, we can average (mean) the ratings of these neighbors and present it to the user.

5. (1.0 bonus points) Prepare a soft k-means classifier using the guideline provided for Question #3 above.