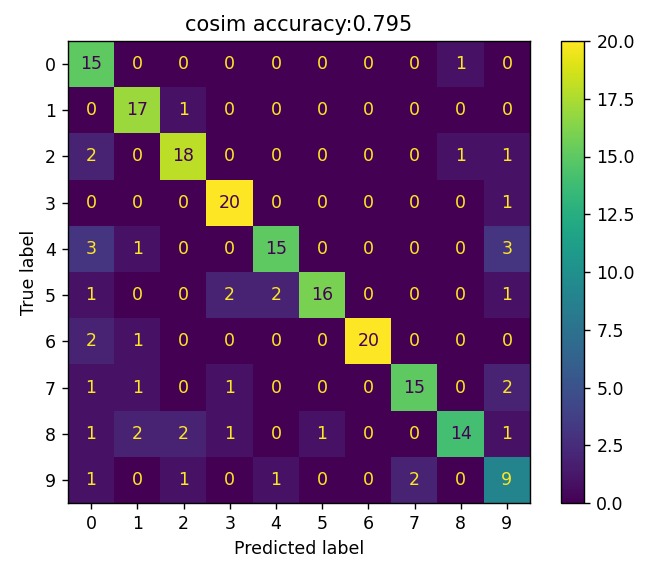
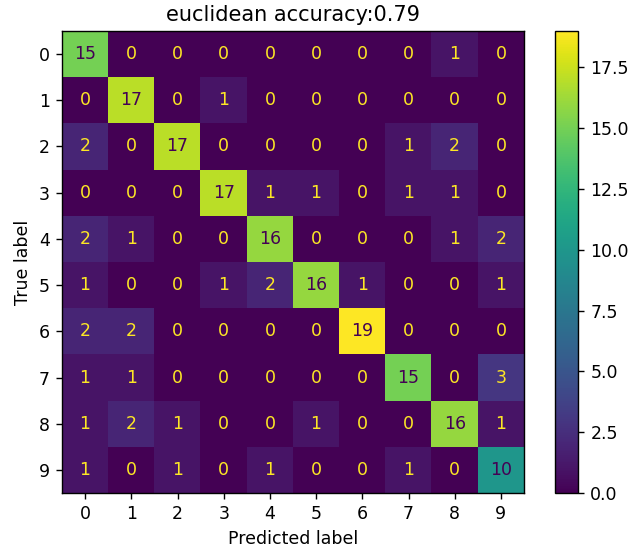
Homework 2 Writeup

CS 349 - Machine Learning

Spencer Rothfleisch, Louie Shapiro, Max Ward

**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 use 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 our KNN classifier, we chose to use Principal component analysis (PCA), a statistical form of dimensionality reduction, for transforming our data. PCA reduces the data to the dimensions that explain the majority of the variance of the dataset, greatly improving speed while only slightly decreasing accuracy. For our hyperparameters, we chose K = 10 and used all of the training dataset (which we determined through a trial and error process).

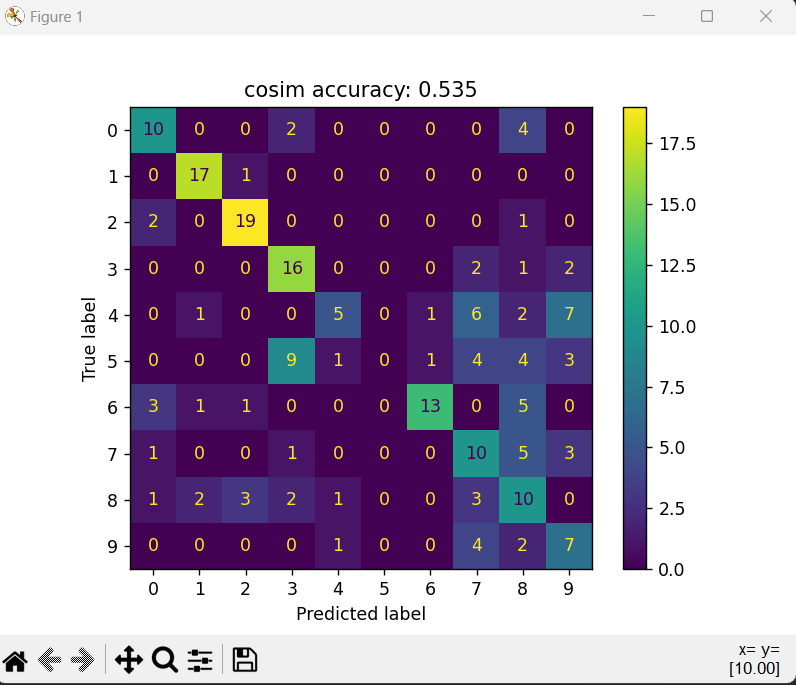


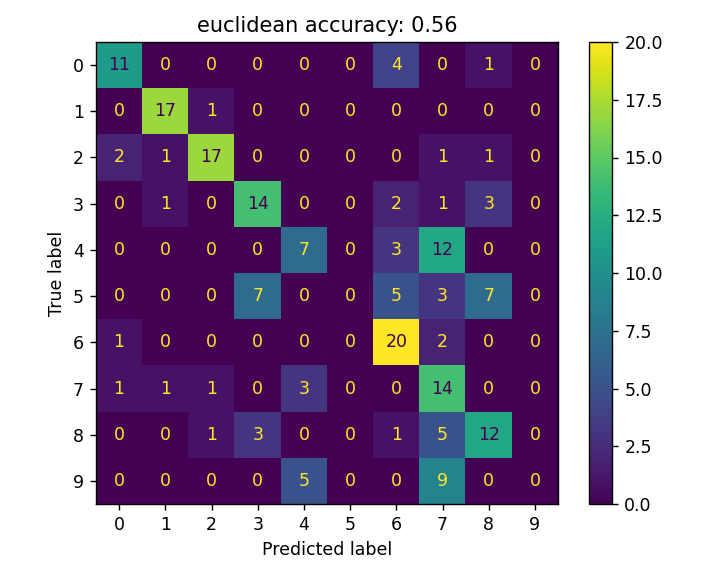
Our KNN confusion matrices for euclidean and cosine similarity distance metrics

Our results for KNN were near 80% for both the cosim and euclidean distance metrics. The confusion matrix displays some errors that our model made. For example, our KNN incorrectly called 4s and 6s zeros. This could be due to the similar shape (including a closed circle) if the 4s and 6s and 0s within the test/training data. Overall, the model performed with high accuracy. The model performed at a higher accuracy (around 85-90%) when PCA dimension reduction was removed. We decided it was acceptable to trade increased performance for a minor drop in accuracy.

**3. (4.0 points) Implement a k-means classifier in the same manner as described above for the k-nearest 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.**

The design choice we made for transforming the data was converting the data to binary, where 0 represented white and 1 represented black. This facilitates an improved performance because it reduces the noise of the data and enhances contrast between the pixels as there are no longer gradual changes in shade color in parts of the image. The cutoff we chose for whether it should be a 0 or 1 was 120, slightly below the middle pixel value of 127.5. We chose this value because in our extensive testing it yielded the best results. Our K value was set to 10, as we wanted to have 1 cluster for each number. To create the initial clusters, we started out with complete randomization. However, this caused certain clusters to be too close to each other to begin and thus the convergence would take a significant amount of time. We solved this issue with another major design choice, a random selection of the initial mean and then placing the other means semi-randomly. This means that the means are a certain distance apart from each other to ensure that the clusters are not going to be initialized too close to each other, but the randomization still comes into play after the initial distance apart is determined.





Our K means confusion matrices for euclidean and cosine similarity distance metrics

Our results were in the low to mid 50% range for both euclidean and cosine similarity, with euclidean typically being a few percentage points higher. This can be attributed to difficulties K-Means Clustering had in deducing certain numbers, such as 3, 5, and 8. Before transforming the data, the percentages were even lower, as the classifier being run with both distance metrics would result in around 40%. We are overall satisfied with our results here, but we have come to understand that KNN is a better choice of classifying algorithm for this task.

**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.**

Typically, the cosine similarity distance metric is used for collaborative filtering. Cosine similarity is a commonly used metric in this context since it calculates the cosine of the angle between two vectors, reflecting the orientation rather than the magnitude, which is useful when comparing user preferences. By looking at the orientation instead of the magnitude one can tell how similar the entirety of a user’s watch information and profile are no matter how many movies or TV shows they have watched, rated, etc.This is valuable because two users could have interacted with a different number of items, or rated items by different scales (thumbs up/thumbs down versus star ratings), but still share similar preferences in terms of the pattern of interactions or ratings. Therefore, cosine similarity provides a measure of similarity that is more reflective of shared preferences or behaviors, rather than being influenced by a sheer number of watched programs/ratings.

As for implementing a collaborative filter, the approach we would use (and will use for our final project!) is the User-User Collaborative Filtering where the similarity between users is calculated using cosine similarity, and recommendations are made based on the preferences of similar users. Specifically, to estimate how much a user would like a movie or TV show that they have not seen, we can find the k-nearest neighbors to the user and then calculate the weighted average of the ratings these neighbors have given to that movie or TV show, with the weights being the similarity scores between the users.This average would represent how much the algorithm predicts the user would like the movie if they had rated it themselves.