1. (1.0 points)
Implement functions to compute Euclidean Distance, Cosine Similarity, Pearson Correlation and Hamming Distance between two input vectors a and b. These functions should return a scalar float value. To ensure your functions are implemented correctly, you may want to construct test cases and compare them against results packages like numpy or sklearn.

See starter.euclidean(), starter.cosim(), starter.pearson_correlation(), and starter.hammering_distance()

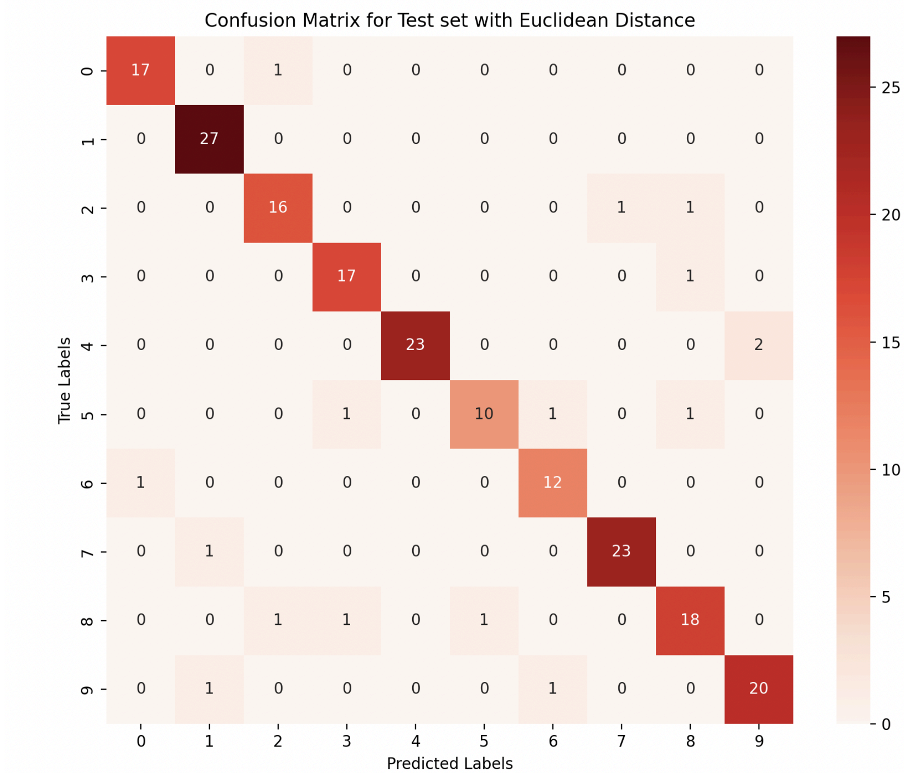
2. (3.0 points)
(1) Implement a k-nearest neighbors classifier for Euclidean distance and Cosine Similarity using the signature provided in starter.py. This algorithm may be computationally intensive. To address this, you must transform your data in some manner (e.g., dimensionality reduction, mapping 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 can select your hyper-parameters (e.g., K, the number of observations to use, default labels, etc.).

See starter.knn()

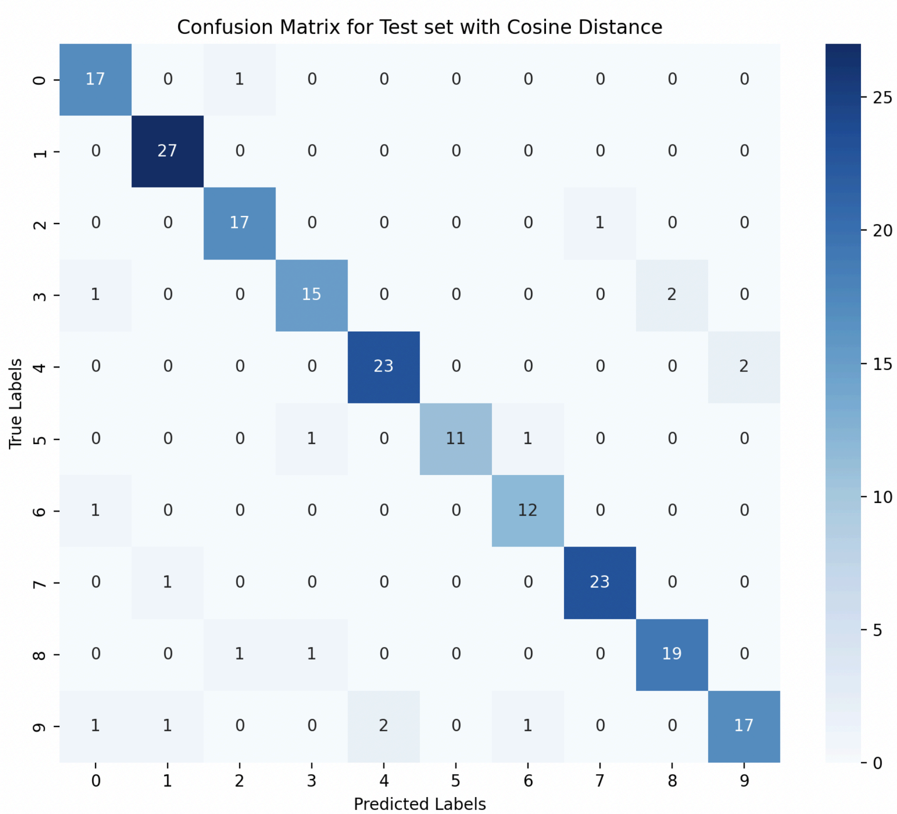
(2) Please describe all of your design choices and hyper-parameter selections in a paragraph.

2 distance metrics —Euclidean and cosine similarity—are used to provide versatility for various applications, adapting to different kinds of data provided by the users. K, the number of nearest neighbors, is a hyperparameter to enable users to control the trade-off between bias and variance in predictions. The n_comp parameter for PCA helps in dimensionality reduction, which is important for improving efficiency and performance while working with high-dimensional data. It ensures that the KNN method does not exceed the number of features available.The function structure also promotes easy extensions for additional distance metrics in the future. Overall, these design choices are aimed at providing a solid foundation for effective KNN classification while maintaining a user-friendly environment.

(3) Once you are satisfied with the performance on the validation set, run your classifier on the test set and summarize results in a 10x10 confusion matrix for each distance metric.



Confusion Matrix for Test set with Euclidean Distance
True Labels
Predicted Labels



Confusion Matrix for Test set with Cosine Distance
True Labels
Predicted Labels

(3) Analyze your results in another paragraph.

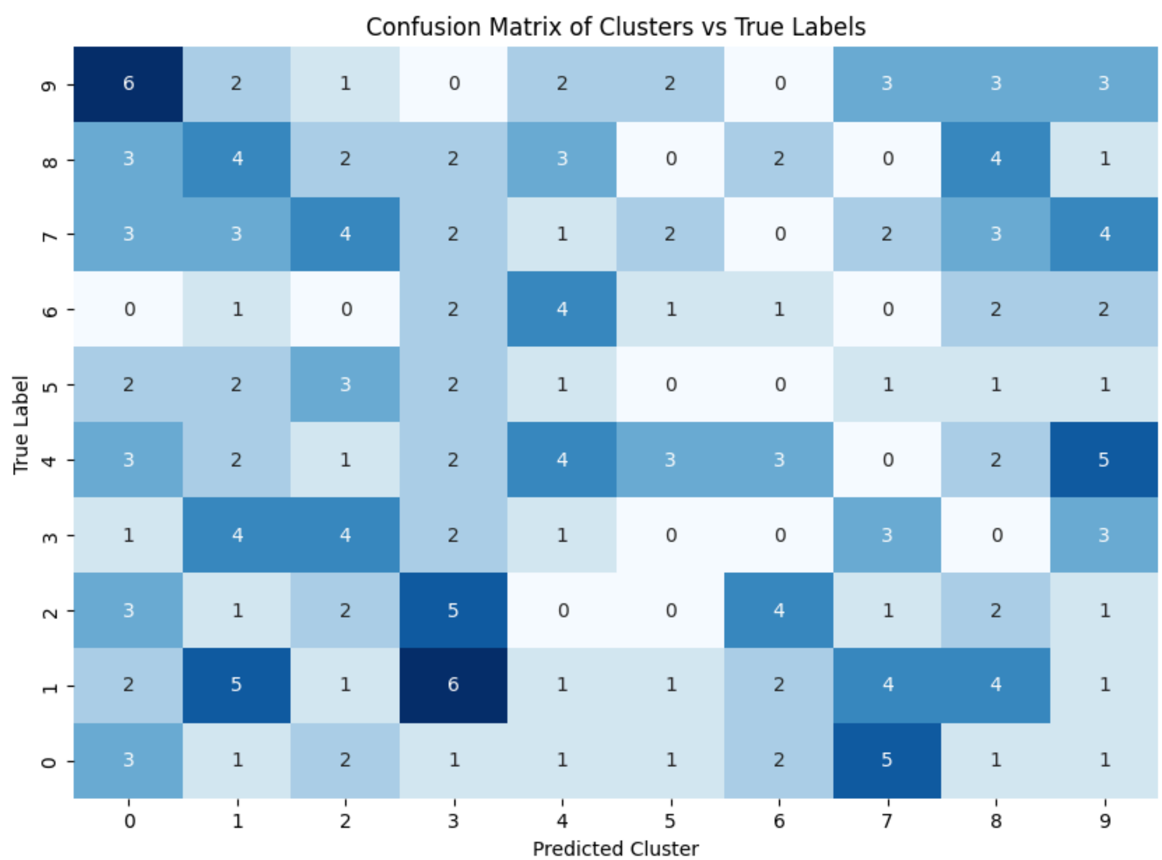
KNN using euclidean and cosine distance metrics has very similar accuracy based on the confusion matrix. Both models do very well in the evaluation process based on the confusion matrix. With most of the values landing on the diagonal, meaning the predicted label matched the actual label, and only a few 1s off the diagonal (missed prediction), it suggests this is a fairly accurate and satisfying KNN model.

3. (3.0 points)

(1) 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.

See starter.kmeans()

(2) Present a quantitative metric to measure how well your clusters align with the labels in mnist_test.csv.



Accuracy: 0.1300
Precision: 0.1216
Recall: 0.1215
F1 Score: 0.1202

(3) Describe your design choices and analyze your results in about one paragraph each.

Design Choices

For this k-means implementation, several design decisions were made to enhance clustering on the MNIST dataset. First, dimensionality reduction using PCA was applied to reduce high-dimensional pixel data to a lower-dimensional space, capturing essential features while discarding noise. This was followed by normalization to ensure all features contributed equally, which helps especially when using cosine similarity as a distance metric. Initialization of centroids was handled by selecting random points from the training data, with a mechanism to reassign centroids of empty clusters to points farthest from existing centroids, ensuring each cluster captures a unique subset of data. A convergence threshold was introduced to avoid early stopping due to minor fluctuations in centroid positions.

Analysis of Results

The results, with an accuracy of 13% and relatively low precision, recall, and F1 score (all around 12%), indicate that the k-means clusters only weakly align with the true labels in the MNIST dataset. This performance is above random guessing but suggests that the clusters do not fully capture the unique characteristics of each digit class. The confusion matrix reveals that the model frequently misclassifies digits across multiple clusters, likely due to overlap in visual features between certain digits (e.g., 1 and 7, 3 and 8). While dimensionality reduction and normalization helped improve clustering somewhat, these results highlight the limitations of k-means on complex, high-dimensional image data, where alternative clustering methods or additional feature engineering might yield better results.

4. (3.0 points)

(1) Using one (or more) of the distance metrics implemented in Question #1 above to build a collaborative filter using movie ratings only on movielens.txt to recommend movies for users a, b and c using the train_{a,b,c}.txt. The number of users to consider K and the number of movies to recommend M are hyper-parameters, among others, that you can tune on valid_{a,b,c}.txt.

See starter.get_top_k_similar_users()

(2) Please describe how your collaborative filter works, list the hyper-parameters and describe their role.

The following algorithm plays the role of the collaborative filter by generating a list of movies for user A. The respective roles of the hyper-parameters can be understood by going through the steps. Let u_x represent a user, where x is a user ID:

u_x = (x, {$movie\_name\_0$: user x's rating for that movie, $movie\_name\_1$: user x's rating for that movie, ..., $movie\_name\_n$: user x's rating for that movie})

(i) Extract {u_m | m is a user_id of movielens.txt}

(ii) Extract u_a from train_a.txt, where a is the user ID of user A.

(iii) Find the K most similar users of u_a among {u_m}, {u_m1, u_m2, ..., u_mk}, up to metric M.

(iv) Of all users in {u_m1, u_m2, ..., u_mk}, recommend their watched movies with rating greater than or equal to *threshold* (= 4 by default) that user A hasn't seen.

(3) Report precision, recall, and the F1-score on the validation and test sets for users a, b, and c.

When computing the metrics, we set metric = “cosine”.
User A (top: test set, bottom: validation set):

```
precision, recall, f1, ratings = evaluate(train_a, test_a, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.75 Recall = 1.0 F1 = 0.8571428571428571
```

```
precision, recall, f1, ratings = evaluate(train_a, valid_a, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.5 Recall = 1.0 F1 = 0.6666666666666666
```

User B (top: test set, bottom: validation set):

```
precision, recall, f1, ratings = evaluate(train_b, test_b, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.25 Recall = 1.0 F1 = 0.4
```

```
precision, recall, f1, ratings = evaluate(train_b, valid_b, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.47058823529411764 Recall = 1.0 F1 = 0.6399999999999999
```

User C (top: test set, bottom: validation set):

```
precision, recall, f1, ratings = evaluate(train_c, test_c, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.5263157894736842 Recall = 1.0 F1 = 0.6896551724137931
```

```
precision, recall, f1, ratings = evaluate(train_c, valid_c, movieLens, k = 5, metric = "cosine", demographic = "False")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.46153846153846156 Recall = 1.0 F1 = 0.631578947368421
```

(4) Discuss how M impacts your results.

Precision & F1: Pearson works better on users A and B. Cosine works better on user C.
Recall: There is no impact on recall for the trivial reason that recall is always 1 whatsoever. This is not because our program is incorrect. Rather, our program only gives positive predictions (we only give predictions about what movies our target user *will* like, that is, we don’t predict the movies that our target user *won’t* like).

See the dotted lines in Figs. 1-3 below.

5. (2.0 points)
(1) Try to improve your collaborative filter built in Question #5 by using movie genre or user demographic data (e.g., age, gender and occupation).

See starter.d_get_top_k_similar_users()

(1) Report precision, recall, and the F1-score on the validation and test sets for users a, b, and c.

We use demographic data. When computing the metrics, we set metric = "cosine".
User A (top: test set, bottom: validation set):

```python
precision, recall, f1, ratings = evaluate(train_a, test_a, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.75 Recall = 1.0 F1 = 0.8571428571428571
```

```python
precision, recall, f1, ratings = evaluate(train_a, valid_a, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.5 Recall = 1.0 F1 = 0.6666666666666666
```

User B (top: test set, bottom: validation set):

```python
precision, recall, f1, ratings = evaluate(train_b, test_b, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.25 Recall = 1.0 F1 = 0.4
```

```python
precision, recall, f1, ratings = evaluate(train_b, valid_b, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.47058823529411764 Recall = 1.0 F1 = 0.6399999999999999
```

User C (top: test set, bottom: validation set):

```python
precision, recall, f1, ratings = evaluate(train_c, test_c, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.5263157894736842 Recall = 1.0 F1 = 0.6896551724137931
```

```python
precision, recall, f1, ratings = evaluate(train_c, valid_c, movieLens, k = 5, metric = "cosine", demographic = "True")
print("Precision =", precision, "Recall =", recall, "F1 =", f1)
```
✓ 0.0s Python

```
Precision = 0.46153846153846156 Recall = 1.0 F1 = 0.631578947368421
```

(2) Discuss your approach and whether or not considering additional features improved the performance of your collaborative filter.

From how the plots below look, it is hard to give a yes-or-no answer to this question. Whether there is improvement (as measured by F1) depends on the number of K, the method used, and differs on a per user basis. For example, with user A and user B, there seems a definite improvement when the metric is Pearson and K is large. But with user C, all else being equal, we're worse off with additional features.

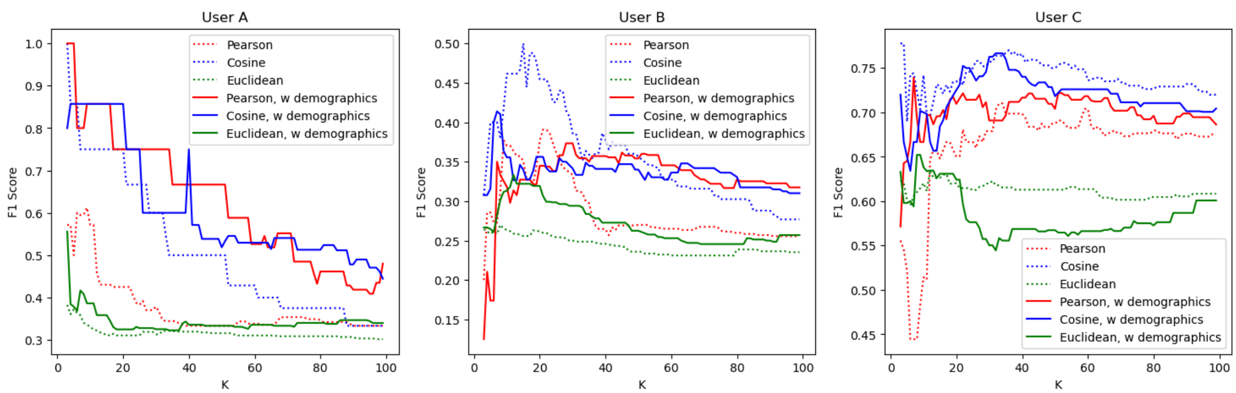


Fig. 1: F1 Score

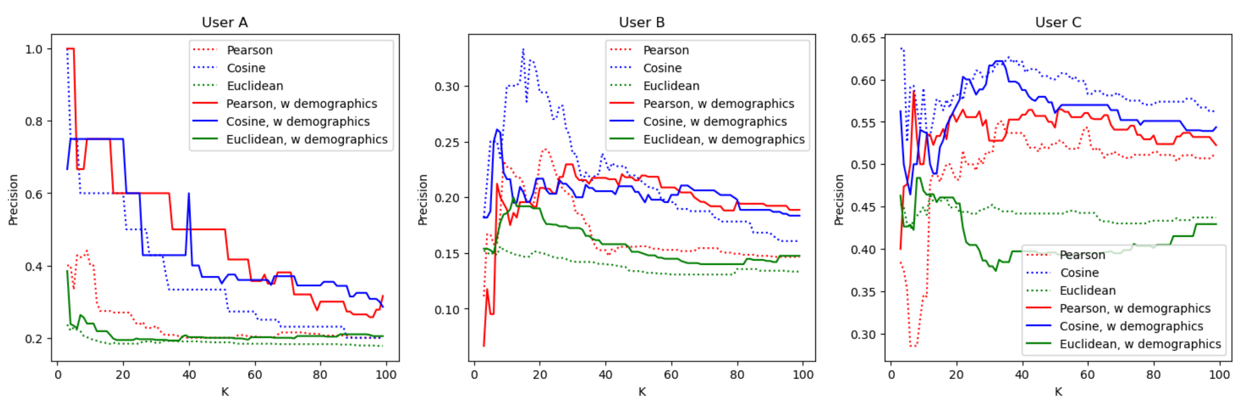


Fig. 2: Precision

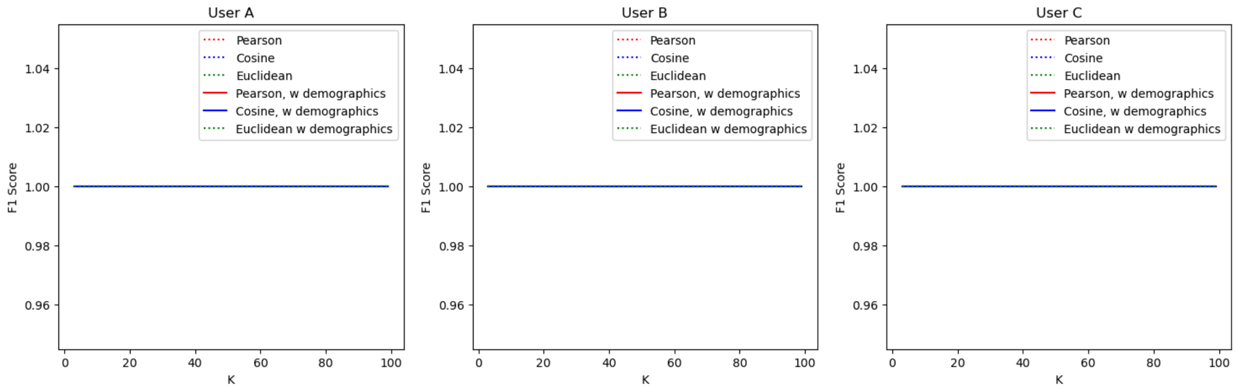


Fig. 3: Recall. Notice that K being constantly 1 is completely normal, since we are only giving positive predictions (we're only predict movies that our target user *will* like).