MSAI349: Machine Learning
Homework 2

Sayantani Bhattacharya, Harrison Bounds, Andrew Kwolek, Sharwin Patil

November 3, 2024

Contents

1 Distance Metrics, K-Nearest Neighbors, K-Means-Clustering

1.1 Distance Metrics

We implemented 4 distance metrics that are capable of comparing any 2 n dimensional vectors to each other:

1. Euclidean Distance
2. Cosine Similarity
3. Hamming Distance
4. Pearson Distance

Each of these metrics were implemented without the use of python libraries and were verified against the output of those metrics offered by libraries such as `sklearn`, `numpy`, and `scipy`. These tests were written using `pytest` and can be run with the command: `pytest -s` *(-s flag will show stdout for debugging)*.

1.1.1 Euclidean Distance

Euclidean distance scales similarly in low vs high dimensions and is a good option for low dimension comparisons. Additionally, euclidean distance maps "physical" proximity which may be useful for data that has geometric features such as image data.

$$d(x,y)_{\text{euclidean}} = \sqrt{\sum (x_i - y_i)^2} \tag{1}$$

```
euclidean_distance = np.sqrt(np.sum(np.subtract(a, b)**2))
```

1.1.2 Cosine Similarity

Cosine Similarity will measure how similar two vectors are by finding the "angle" between them. Despite this concept being impossible to visualize in higher dimensions, cosine similarity excels in high dimensional data, making it a good metric for image data, natural language processing, and other high-dimensional data. Note that the value is bounded between $[-1, 1]$ so the magnitude of the data isn't encapsulated in the return value.

$$d(x,y)_{\text{cosim}} = \frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}} \tag{2}$$

```
cosim = np.dot(a, b) / np.sqrt(np.sum(a**2)) * np.sqrt(np.sum(b**2))
```

1.1.3 Hamming Distance

Hamming distance counts the number of differences between the vectors (Both vectors must be the same dimension and size). Hamming distance is useful for categorical data, error detection, and comparing strings (edit distance for spell-checking).

$$d(x, y)_{\text{hamming}} = \sum \delta(x_i, y_i), \ \delta = 1 \text{ if } x_i \neq y_i \text{ else } \delta = 0 \tag{3}$$

```
hamming_distance = np.array([ai != bi for ai, bi in zip(a, b)]).sum()
```

1.1.4 Pearson Distance

Pearson distance measures the linear relationship between the vectors.

$$d(x, y)_{\text{pearson}} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}} \tag{4}$$

```
pearson_distance = np.sum((a - a_bar)*(b - b_bar)) /  np.sqrt(np.sum((a - a_bar)**2)) *
↪  np.sqrt(np.sum((b - b_bar)**2))
```

1.2 K-Nearest Neighbors Classifier

When implementing K-Nearest Neighbors, our only hyper-parameter was k. When attempting to use less observations during the distance comparison, there was a drop in accuracy compared to using all the of the given observations. Dimensionality reduction improved the algorithm's efficiency but had no impact (positive or negative) on performance so dimensionality reduction was always done before using KNN.

We got satisfactory results with $k = 5$ using euclidean distance as our distance metric. Dimensionality reduction didn't impact our results at all, however made execution much faster ($\approx$ 15% faster). The results observed for the mnist testing and validation datasets upon KNN classification:

```
Testing Data Metrics (euclidean):
    Accuracy: 0.9800000000000001
    Precision: 0.9025205627705628
    Recall: 0.8905366767735188
    F1_score: 0.8900603439819568
Validation Data Metrics (euclidean):
    Accuracy: 0.8150000000000001
    Precision: 0.08002930605013658
    Recall: 0.07614514712340799
    F1 Score: 0.07582969280837747
```

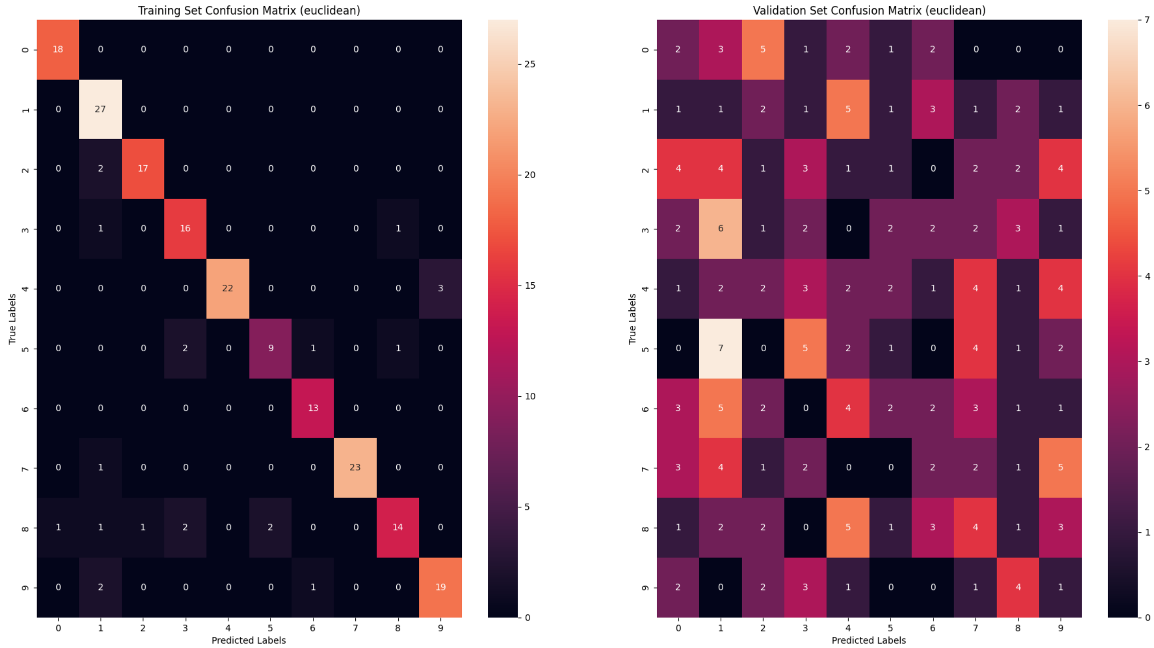

Figure 1: Confusion Matrices for KNN Classifier using the Euclidean distance metric

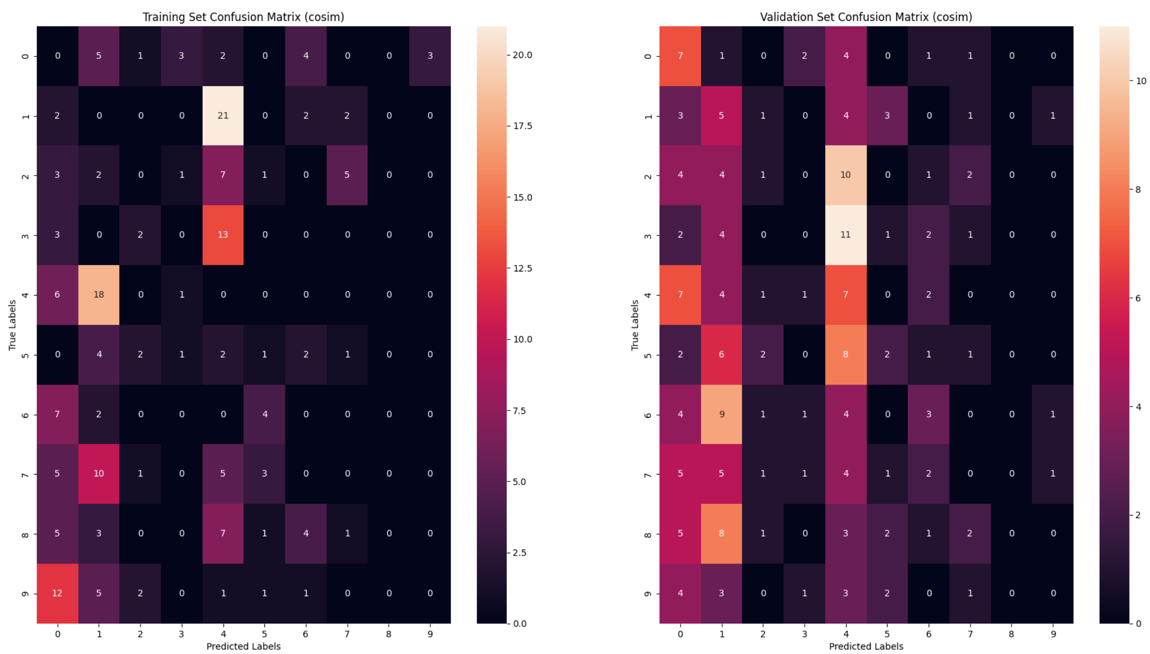

Figure 2: Confusion Matrices for KNN Classifier using the Cosine Similarity distance metric

1.3 K-Means Classifier

In order to determine how well our clusters aligned with the labels in `mnist_test.csv`, we created a metric that calculates the total accuracy of the clusters based on how many points are correctly labeled. We do this by looping through the K clusters and assigning each cluster a label based on the mode of the true labels in the cluster. Then we calculate how many points are correctly classified based on the assigned cluster label. Ideally, each cluster would only contain points of the same label when run on the testing set. We tried this for different K values and found different accuracies.

```
Accuracy(K=10, Euclidean): 0.625
Accuracy(K=36, Euclidean): 0.775
```

When K was increased, the overall accuracy was improved, however the computational cost also increased with higher K values resulting in trade-offs between speed and accuracy. Better performance at higher K values could be attributed to the points in mnist forming clusters that aren't fully representative of the 10 labels, but rather there are multiple clusters embedded in each label, resulting in better accuracy with a higher K. Euclidean distance and cosine similarity performed relatively the same, so we chose to use Euclidean. Some unique design choices we made include using variance thresholding for dimensionality reduction and a random sampling technique for initializing centroids which calculates probabilities based on distances from the currently initialized centroids. This prevents centroids from being initialized very close to one another which breaks the algorithm.

2 Movie Reccomender

2.1 Collaborative Filter

The collaborative filter is made up of multiple steps to ensure a good recommendation is given to the target user. First, a user item matrix is built where the rows are user ids and the columns are movie ids. Each value in this matrix is the user's rating for that particular movie. This is made with the training set for each user, combined into one single matrix for all training data. Second, we take each row as a vector, and calculate the cosine similarity with a function that is used in part 1. Doing this gets us a value that tells us how similar each user is to one another. We can get the K top most similar users and look at their movies that the target user hasn't seen. To do this, we look through these top K users and assign each rating a weight based on the actual rating, and the similarity with the target user. This ensures that the most similar users have higher weighted ratings, which will give more accurate recommendations. Finally, this weighted ratings list returns the top M movies that the target user has not rated, but will enjoy based on its most similar K users.

K and M are the hyper parameters for this collaborative filtering, where K is the number of similar users closest to the target user, and M is the number of movies to recommend to the target user. Tuning K to a higher number would allow us to choose from a wider variety of movies, increasing the performance of our model. M is the number of movies that is recommended to the target user. Increasing M can have a negative affect on the model because each new recommendation must pass the evaluation standards. Here we are calculating the precision, recall, and f1 score. With less variety, increasing M might make your movies fall below a certain rating threshold set (another hyper-parameter), decreasing the performance of the model. Yet, if there are many good movies to recommend, this is a good way to decrease your false negatives and achieve a higher recall and f1 score.

Overall, testing the model turned out to be tricky. With a truth label between 1-5, there was no clear-cut way to accumulate true positives, false positives, and true negatives (each of the values needed to evaluate the model). So, a rating threshold was set and if the recommended movie exceeded this rating threshold, the value was counted as a true positive. Another roadblock was the differences in the training, testing, and validation sets. If a movie recommended in the training set did not appear in the validation or testing sets, it would impact the model negatively, influencing the evaluation metrics even if the recommended movies were accurate.

```
Testing Data Metrics (cosine similarity):
    Precision: 0.09090909090909091,
    Recall: 0.001937984496124031,
    F1 Score: 0.0018975332068311196
Validation Data Metrics (cosine similarity):
    Precision: 0.5,
    Recall: 0.008988764044943821,
    F1 Score: 0.008830022075055188
```

Our metrics perform poorly because the movies recommended in the training set are not found in the test set, therefore there is no way to represent that value. This negatively impacts both the training and validation metrics.

2.2 Collaborative Filter with demographic information

The approach of including demographic information enhances the recommended system by making it more personalized. The metric used is the normalized version of the similarity, to ensure the metric is comparable with respect to the rating metric and the range of age variation in the dataset.

We added weight values to each metric (rating value and age), with the weights being the hyper parameters for the filter. Tuning them would determine the dominance of each metric over deciding the final movie to recommend. To check, if we set the age weight to be 0, we get the same result as above. Upon tuning we got a stabilized result at 50-50 weight to both the metrics, and improved the performance by a good margin.

```
Testing Data Metrics (cosine similarity):
    Precision: 0.3125,
    Recall: 0.007246376811594203,
```

```
    F1 Score: 0.00708215297450425
Validation Data Metrics (cosine similarity):
    Precision: 0.36363636363636365
    Recall: 0.007029876977152899,
    F1 Score: 0.00689655172413793
```

3 Code Appendix

3.1 Distance Metrics

```python
import numpy as np


# returns Euclidean distance between vectors and b
def euclidean(a, b):
    """ Returns the euclidean distance between vectors a and b

    Args:
        a (np.ndarray): A vector of any dimension
        b (np.ndarray): A vector of any dimension

    Returns:
        float: The euclidean distance between the two vectors

    Raises:
        ValueError: If the given vectors are different dimensions
    """
    return np.sqrt(np.sum(np.subtract(a, b)**2))


# returns Cosine Similarity between vectors a and b
def cosim(a, b):
    """ Returns the cosine similarity between vectors a and b

    Args:
        a (np.ndarray): A vector of any dimension
        b (np.ndarray): A vector of any dimension

    Returns:
        float: The cosine similarity between the two vectors

    Raises:
        ValueError: If the given vectors are different dimensions
    """

    if not in_same_dimension(a, b):
        print(
            f"Given vectors have different shapes: " +
            f"{np.shape(a)} != {np.shape(b)}"
        )
        raise ValueError(
            "Cosine Similarity requires 2 identically-shaped vectors"
        )
    numerator = np.dot(a, b)
    denominator = np.sqrt(np.sum(a**2)) * np.sqrt(np.sum(b**2))

    # If denominator is 0, cosim is undefined not 0
    if denominator == 0:
        return np.nan

    return numerator / denominator


# returns Pearson Correlation between vectors a and b
def pearson(a: np.ndarray, b: np.ndarray):
    """ Returns the pearson correlation between vectors a and b

```

```
    Args:
        a (np.ndarray): A vector of any dimension
        b (np.ndarray): A vector of any dimension

    Returns:
        float: The pearson correlation between the two vectors

    Raises:
        ValueError: If the given vectors are different dimensions
    """
    a_bar = np.sum(a)/len(a)
    b_bar = np.sum(b)/len(b)

    numerator = np.sum((a - a_bar)*(b - b_bar))
    denominator = np.sqrt(np.sum((a - a_bar)**2)) * \
        np.sqrt(np.sum((b - b_bar)**2))

    if numerator == 0 or denominator == 0:
        return 0

    return numerator / denominator


def hamming(a: np.ndarray, b: np.ndarray) -> int:
    """ Returns the Hamming distance between vectors a and b

    Args:
        a (np.ndarray): A vector of any dimension
        b (np.ndarray): A vector of any dimension

    Returns:
        int: The Hamming distance between the two vectors

    Raises:
        ValueError: If the given vectors are different dimensions
    """
    # Ensure that the two vectors occupy the same dimension
    if not in_same_dimension(a, b):
        print(
            f"Given vectors have different shapes: " +
            f"{np.shape(a)} != {np.shape(b)}"
        )
        raise ValueError("Hamming requires 2 identically-shaped vectors")
    # Create a vector
    comparison_vector = np.array([ai != bi for ai, bi in zip(a, b)])
    return comparison_vector.sum()


def in_same_dimension(a: np.ndarray, b: np.ndarray) -> bool:
    """ Determines if the two given vectors are in the
        same dimension or not

    Args:
        a (np.ndarray): The first vector of any dimension
        b (np.ndarray): The second vector of any dimension

    Returns:
        bool: Whether the 2 vectors are the same dimension or not
    """
    return np.shape(a) == np.shape(b)
```

3.2 K-Nearest-Neighbors Implementation

```python
from distance_metrics import euclidean, cosim
from starter import read_data, reduce_data, reduce_query
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# returns a list of labels for the query dataset based upon observations in the train dataset.
# labels should be ignored in the training set
# metric is a string specifying either "euclidean" or "cosim".
# All hyper-parameters should be hard-coded in the algorithm.


def knn(train: list, query: list, metric: str, k: int = 5, debug: bool = False) -> list:
    """
    Returns a list of labels for the query dataset based upon observations in the train dataset.

    Note that the length of the labels returned is the same as the length of the query dataset
    since each query is assigned a label.

    Args:
        train (list): The training dataset or examples that KNN will utilize to calculate distance and assign labels
        query (list): The dataset of queries that KNN will assign labels to [query_label, [list(pixels)]]
        metric (str): The distance metric to use for KNN. Either 'euclidean' or 'cosim'
        k (int, optional): The number of neighbors to consider. Defaults to 5.
        debug (bool, optional): Whether to print debug information. Defaults to False.

    Raises:
        ValueError: If the distance metric is not 'euclidean' or 'cosim'
        ValueError: If the query data is not the same size as the data in the training set.

    Returns:
        list: The labels assigned to each query in the query dataset
    """
    f_d = None
    if metric == 'euclidean':
        f_d = euclidean
    elif metric == 'cosim':
        f_d = cosim
    else:
        raise ValueError('Invalid distance metric given')
    if debug:
        print(
            f'K-Nearest Neighbors using {metric} distance metric and k={k}, ' +
            f'{len(train)} training examples and {len(query)} queries:'
        )
    labels = []
    for q_label, q in query:
        # Sort neighbors using distance metric and take the first k entries
        nearest_neighbors = sorted(
            [t for t in train], key=lambda x: f_d(x[1], q)
        )[:k]
        labels_for_neighbor = [int(t[0]) for t in nearest_neighbors]
        # Find the most common label among the k closest neighbors
```

```python
        most_common_label = np.argmax(np.bincount(labels_for_neighbor))
        if debug:
            print(
                f'Query {q}\n' +
                f'Nearest neighbors: {nearest_neighbors}\n' +
                f'Labels for neighbors: {labels_for_neighbor}\n' +
                f'Most common label: {most_common_label}\n' +
                f'Expected label: {q_label}'
            )
        labels.append(most_common_label)
    return labels


def evaluate_knn_accuracy(labels: list, query: list) -> tuple:
    """
    Calculates and prints metrics, i.e. Accuracy, Precision, Recall and F1 Score for a trained KNN model.

    Args:
        labels (list): The labels assigned to each query in the query dataset by the KNN model, whose accuracy is being measured.
        query (list): The dataset of queries that KNN will assign labels. [query_label, [list(pixels)]]

    Returns:
        tuple: A tuple containing the accuracy, precision, recall, and F1 score of the KNN model

    """
    # For 100% accuracy, diagonal elements of confusion matrix need to be non-zero and rest all needs to be 0.
    expected_result = [row[0] for row in query]
    confusion_matrix = generate_confision_matrix(labels, expected_result)
    num_labels = confusion_matrix.shape[0]
    metrics = []  # (accuracy, precision, recall, f1_score)
    for i in range(num_labels):
        # True Positives: Diagonal elements (Correctly classified)
        tp = confusion_matrix[i][i]
        # False Negatives: Everything in this row except TP since it is not classified as i
        fn = np.sum(confusion_matrix[i, :]) - tp
        # False Positives: Everything in this column except TP since it is not classified as i
        fp = np.sum(confusion_matrix[:, i]) - tp
        # True Negatives: Everything except TP, FN, FP
        tn = np.sum(confusion_matrix) - (tp + fn + fp)
        accuracy = (tp + tn) / np.sum(confusion_matrix)
        precision = tp / (tp + fp) if tp + fp > 0 else 0
        recall = tp / (tp + fn) if tp + fn > 0 else 0
        f1_score = (2 * precision * recall) / \
            (precision + recall) if precision + recall > 0 else 0
        metrics.append(
            [accuracy, precision, recall, f1_score]
        )
    accuracy, precision, recall, f1_score = np.mean(metrics, axis=0)
    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"F1_score: {f1_score}")
    return (accuracy, precision, recall, f1_score)
```

```python

def generate_confision_matrix(labels: list, expected_result: list):
    """
    Returns the confusion matrix with the input of label and expected result.

    Args:
        labels (list): The labels assigned to each query in the query dataset by the KNN model, whose accuracy is being measured.
        expected_result (list): The correct label values of the query dataset.

    Returns: Confusion matrix (a 2D array): is a square (n*n) matrix, with n = number of label options, as the union of knn and actual label of the querry set.
            In this case, if the input data is sufficiently large: CM -> 10*10
    """
    n = len(set(expected_result))
    confusion_matrix = np.zeros((n, n), dtype=int)
    for expected_label, predicted_label in zip(expected_result, labels):
        confusion_matrix[int(expected_label)][int(predicted_label)] += 1
    return confusion_matrix


def display_confusion_matrices(train_matrix, validation_matrix, metric):
    fig, axs = plt.subplots(1, 2, figsize=(10, 10))
    axes = axs.flatten()
    sns.heatmap(train_matrix, annot=True, fmt='d', ax=axes[0])
    axes[0].set_xlabel("Predicted Labels")
    axes[0].set_ylabel("True Labels")
    axes[0].set_title(f"Training Set Confusion Matrix ({metric})")
    sns.heatmap(validation_matrix, annot=True, fmt='d', ax=axes[1])
    axes[1].set_xlabel("Predicted Labels")
    axes[1].set_ylabel("True Labels")
    axes[1].set_title(f"Validation Set Confusion Matrix ({metric})")
    plt.tight_layout()
    plt.show()


def run_knn():
    # Parse the MNIST dataset
    mnist_training_data = read_data("mnist_train.csv")
    mnist_testing_data = read_data("mnist_test.csv")
    mnist_validation_data = read_data("mnist_valid.csv")

    print(
        f'Training Data Size: {len(mnist_training_data)}\n' +
        f'Testing Data Size: {len(mnist_testing_data)}\n' +
        f'Validation Data Size: {len(mnist_validation_data)}'
    )

    # Before using training data, we may need to run dimensionality reduction on it
    # to reduce the number of features. We should try reduce() that we wrote
    # but we should try other methods that the assignment reccomends as well:
    # grayscale to binary, dimension scaling, etc.
    reduced_training_data, train_features = reduce_data(mnist_training_data)
    reduced_testing_data, test_features = reduce_data(mnist_testing_data)
    reduced_validation_data, valid_features = reduce_data(
        mnist_validation_data)

    test_query = reduce_query(mnist_testing_data, train_features)
    valid_query = reduce_query(mnist_validation_data, train_features)

```

```
    # Run training data through KNN and receive the labels for each query
    # We might have to modify KNN so the query is [label, list(pixels)] instead of just
    ↪ list(pixels)
    # so that we can compare the assigned label to the actual label
    # Not actually sure if this is how we do this
    predicted_labels = knn(
        train=reduced_training_data,
        query=test_query,
        metric='euclidean',
        k=5
    )

    training_matrix = generate_confision_matrix(
        predicted_labels, [q[0] for q in mnist_testing_data]
    )

    (accuracy, precision, recall, f1_score) = evaluate_knn_accuracy(
        labels=predicted_labels,
        query=test_query
    )

    print(
        f'Test Data Metrics (euclidean):\n' +
        f'Accuracy: {accuracy}\n' +
        f'Precision: {precision}\n' +
        f'Recall: {recall}\n' +
        f'F1 Score: {f1_score}'
    )

    validation_matrix = generate_confision_matrix(
        predicted_labels, [q[0] for q in mnist_validation_data]
    )

    (accuracy, precision, recall, f1_score) = evaluate_knn_accuracy(
        labels=predicted_labels,
        query=mnist_validation_data
    )

    print(
        f'Validation Data Metrics (euclidean):\n' +
        f'Accuracy: {accuracy}\n' +
        f'Precision: {precision}\n' +
        f'Recall: {recall}\n' +
        f'F1 Score: {f1_score}'
    )

    display_confusion_matrices(training_matrix, validation_matrix, 'euclidean')

    predicted_labels = knn(
        train=reduced_training_data,
        query=test_query,
        metric='cosim',
        k=5
    )

    training_matrix = generate_confision_matrix(
        predicted_labels, [q[0] for q in mnist_testing_data]
    )

    (accuracy, precision, recall, f1_score) = evaluate_knn_accuracy(
        labels=predicted_labels,
```

```
        query=reduced_testing_data
    )

    print(
        f'Test Data Metrics (cosim):\n' +
        f'Accuracy: {accuracy}\n' +
        f'Precision: {precision}\n' +
        f'Recall: {recall}\n' +
        f'F1 Score: {f1_score}'
    )

    validation_matrix = generate_confision_matrix(
        predicted_labels, [q[0] for q in mnist_validation_data]
    )

    (accuracy, precision, recall, f1_score) = evaluate_knn_accuracy(
        labels=predicted_labels,
        query=reduced_validation_data
    )

    print(
        f'Validation Data Metrics (cosim):\n' +
        f'Accuracy: {accuracy}\n' +
        f'Precision: {precision}\n' +
        f'Recall: {recall}\n' +
        f'F1 Score: {f1_score}'
    )

    display_confusion_matrices(training_matrix, validation_matrix, 'cosim')


if __name__ == "__main__":
    run_knn()
```

3.3 K-Means Clustering Implementation

```
import numpy as np
from distance_metrics import euclidean, cosim
from starter import reduce_data, reduce_query, read_data


np.random.seed(30)


def initialize_centroids(k, data):
    centroids = []

    ind = np.random.randint(0, len(data))
    centroids.append(data[ind][1])

    for i in range(1, k):
        distances = []
        for x in data:
            to_centroid = []
            for c in centroids[:i]:
                to_centroid.append(euclidean(x[1], c))
            distances.append(np.min(to_centroid))

        distances = np.array(distances)

```

```
        probabilities = distances**2
        probabilities /= np.sum(probabilities)

        next_centroid = np.random.choice(len(data), p=probabilities)
        centroids.append(data[next_centroid][1])

    return np.array(centroids)


# returns a list of labels for the query dataset based upon observations in the train
↪ dataset.
# labels should be ignored in the training set
# metric is a string specifying either "euclidean" or "cosim".
# All hyper-parameters should be hard-coded in the algorithm.
def kmeans(train, query, metric, k=10, threshold=0.01):
    max_iters = 100
    labels = []
    train_reduced, removed_features = reduce_data(train, threshold=threshold)
    centroids = initialize_centroids(k, train_reduced)

    for it in range(max_iters):
        distances = []
        for c in centroids:
            centroid_dist = []
            for x in train_reduced:
                if metric == "euclidean":
                    centroid_dist.append(euclidean(c, x[1]))
                elif metric == "cosim":
                    centroid_dist.append(cosim(c, x[1]))
            distances.append(np.array(centroid_dist))
        distances = np.array(distances)

        # Assign clusters based on minimum distance to centroids
        if metric == "euclidean":
            clusters = np.argmin(distances, axis=0)
        elif metric == "cosim":
            clusters = np.argmax(distances, axis=0)

        new_centroids = []
        for i in range(k):
            cluster_group = []
            for j in range(len(train_reduced)):
                if clusters[j] == i:
                    cluster_group.append(train_reduced[j][1])

            cluster_group = np.array(cluster_group)

            if len(cluster_group) > 0:
                centroid = cluster_group.mean(axis=0)
            else:
                centroid = np.zeros(len(train_reduced[0][1]))

            new_centroids.append(centroid)

        new_centroids = np.array(new_centroids)

        if np.all(new_centroids == centroids):
            print(f"Exited at {it}")
            break
        else:
            centroids = new_centroids
```

```

    query_reduced = reduce_query(query, removed_features)

    query_distances = []
    for c in centroids:
        centroid_dist = []
        for q in query_reduced:
            if metric == "euclidean":
                centroid_dist.append(euclidean(q[1], c))
            elif metric == "cosim":
                centroid_dist.append(cosim(q[1], c))
        query_distances.append(np.array(centroid_dist))
    query_distances = np.array(query_distances)

    if metric == "euclidean":
        classes = np.argmin(query_distances, axis=0)
    elif metric == "cosim":
        classes = np.argmax(query_distances, axis=0)

    for c in classes:
        labels.append(int(c))

    return labels


def calculate_clustering_accuracy(labels, test_data, k=10):
    label_mapping = {}
    correct = 0
    true_labels = []
    for x in test_data:
        true_labels.append(int(x[0]))

    for c in range(k):
        indices = []
        for i, x in enumerate(labels):
            if x == c:
                indices.append(i)
        cluster_labels = []
        for x in indices:
            cluster_labels.append(true_labels[x])
        if len(cluster_labels) > 0:
            vals, count = np.unique(
                np.array(cluster_labels), return_counts=True)
            common = vals[np.argmax(count)]
            label_mapping[c] = [common, cluster_labels]

    for key in label_mapping.keys():
        id = label_mapping[key][0]
        values = label_mapping[key][1]
        for val in values:
            if int(id) == val:
                correct += 1

    acc = correct / len(true_labels)
    print(f"K-Means Clustering Accuracy: {acc}")
    return acc


def main():
    mnist_training_data = read_data("mnist_train.csv")
    mnist_testing_data = read_data("mnist_test.csv")
```

```
    mnist_validation_data = read_data("mnist_valid.csv")

    labels = kmeans(mnist_training_data,
                    mnist_testing_data, "euclidean", k=36, threshold=3.0)
    calculate_clustering_accuracy(labels, mnist_testing_data, k=36)


if __name__ == "__main__":
    main()
```

3.4 Collaborative Filtering Implementation

```
import numpy as np
import pandas as pd
from distance_metrics import cosim

# Hyperparameters

K = 10
M = 7
# Weight values for different metrics of the dataset.
rating_weight = 3
age_weight = 3


# rating_threshold: is to discretize the value for calculating the precision etc metrics.
rating_threshold = 4
target_users = [405, 655, 13]
user_ids = []
similar_users = []
similarity_dict = {}
ratings_vector_1 = []
ratings_vector_2 = []

total_matrix = pd.read_csv("movielens.txt", delimiter='\t')
user_age = dict(zip(total_matrix['user_id'], total_matrix['age']))

user_a_train = pd.read_csv("train_a.txt", delimiter='\t')
user_age[user_a_train['user_id'].values[0]] = user_a_train['age'].values[0]
user_b_train = pd.read_csv("train_b.txt", delimiter='\t')
user_age[user_b_train['user_id'].values[0]] = user_b_train['age'].values[0]
user_c_train = pd.read_csv("train_c.txt", delimiter='\t')
user_age[user_c_train['user_id'].values[0]] = user_c_train['age'].values[0]

user_a_valid = pd.read_csv("valid_a.txt", delimiter='\t')
user_b_valid = pd.read_csv("valid_b.txt", delimiter='\t')
user_c_valid = pd.read_csv("valid_c.txt", delimiter='\t')

user_a_test = pd.read_csv("test_a.txt", delimiter='\t')
user_b_test = pd.read_csv("test_b.txt", delimiter='\t')
user_c_test = pd.read_csv("test_c.txt", delimiter='\t')

combined_data = pd.concat([user_a_train, user_b_train, user_c_train])
combined_data_valid = pd.concat([user_a_valid, user_b_valid, user_c_valid])
combined_data_test = pd.concat([user_a_test, user_b_test, user_c_test])


user_total_matrix = total_matrix.pivot_table(index='user_id', columns='movie_id', values='rating').fillna(0)
```

```python


user_item_matrix = combined_data.pivot_table(index='user_id', columns='movie_id', values='rating').fillna(0)
user_valid_matrix = combined_data_valid.pivot_table(index='user_id', columns='movie_id', values='rating').fillna(0)
user_test_matrix = combined_data_test.pivot_table(index='user_id', columns='movie_id', values='rating').fillna(0)


# print("user item matrix: ", user_item_matrix)
# print("user total matrix: ", user_total_matrix)

print("user ids: ", user_ids)

def normalised_metric_similarity(user_id_first:int, user_id_second:int):
    max_age_difference = max(user_age.values()) - min(user_age.values())
    age_similarity = 1 - abs(user_age.get(user_id_first) - user_age.get(user_id_second)) / max_age_difference
    return age_similarity

#Calculate similarity between users
for id1 in target_users:
    print("user id: ", id1)
    for id2 in user_total_matrix.index:
        if id1 == id2:
            continue
        for movie_id in user_total_matrix.columns:
            if movie_id in user_item_matrix.columns and movie_id in user_total_matrix.columns:
                rating1 = user_item_matrix.loc[id1, movie_id]
                rating2 = user_total_matrix.loc[id2, movie_id]
                if rating1 > 0 and rating2 > 0:
                    ratings_vector_1.append(rating1)
                    ratings_vector_2.append(rating2)

        # Calculate the similarity
        rating_sim = cosim(np.array(ratings_vector_1), np.array(ratings_vector_2))
        age_sim = normalised_metric_similarity(id1,id2)
        user_sim = rating_sim * rating_weight + age_sim * age_weight
        similar_users.append((id2, user_sim))

    similar_users = sorted(similar_users, key=lambda x: x[1], reverse=True)
    # Keep only top K similar users
    similarity_dict[id1] = similar_users[:K]

# Recommend movies for each user
true_positive = 0
false_positive = 0
false_negative = 0

for user_id in target_users:
    weighted_ratings = {}

    top_k_users = similarity_dict[user_id][:K]
    #print("Top K Users: ", top_k_users)
    #target_user_ratings = user_item_matrix.loc[user_id]

    for sim_user_id in top_k_users:
        sim_user_ratings = user_total_matrix.loc[sim_user_id[0]]
        similarity_score = sim_user_id[1]
```

```python

        for movie_id in user_total_matrix.columns:
            rating = sim_user_ratings[movie_id]
            if movie_id in user_item_matrix.columns:
                if user_item_matrix.loc[user_id][movie_id] == 0 and rating > 0:
                    if movie_id not in weighted_ratings:
                        weighted_ratings[movie_id] = 0.0
                    weighted_ratings[movie_id] += rating * similarity_score

    best_movies = sorted(weighted_ratings, key=weighted_ratings.get, reverse=True)[:M]

    print(f"The best movies for user {user_id} based off of {K} similar users: {best_movies}")

    # Evaluate using the validation set
    for best_movie_id in best_movies:
        if best_movie_id in user_test_matrix.columns:
            actual_rating = user_test_matrix.loc[user_id, best_movie_id]
            if actual_rating >= rating_threshold:
                true_positive += 1
            else:
                false_positive += 1

            user_ratings = user_test_matrix.loc[user_id]
            for test_movie_id, user_rating in user_ratings.items():
                if test_movie_id not in best_movies and user_rating >= rating_threshold:
                    false_negative += 1

# Evaluation Metrics
precision = true_positive / (true_positive + false_positive)
recall = true_positive / (true_positive + false_negative)
f1_score = (precision * recall) / (precision + recall)

print(f"Precision: {precision}\nRecall: {recall}, \nF1 Score: {f1_score}")
```

3.4.1 Dimensionality Reduction Implementation

```python
import numpy as np
from copy import deepcopy


np.random.seed(30)


def reduce_data(data_set, threshold=0.01):
    """ Returns the reduced dataset using variance thresholding

    Args:
        data_set (ndarray(int, ndarray)): processed data
        threshold (float): variance threshold, default 0.01

    Returns:
        tuple of (ndarray(int, ndarray), list): Reduced dataset and removed features
    """
    data_cp = deepcopy(data_set)
    features = np.array([feature[1] for feature in data_cp])
    variances = np.var(features, axis=0)
    removed_features = [index for index, variance in enumerate(
        variances) if variance < threshold]

```

```python
    for entry in data_cp:
        entry[1] = np.delete(entry[1], removed_features)

    return data_cp, removed_features


def reduce_query(data_set, removed_features):
    """ Returns the reduced query point

    Args:
        (int, ndarray): image
        removed_features (list): list of removed features

    Returns:
        (int, ndarray): Reduced image
    """
    query_cp = deepcopy(data_set)
    for entry in query_cp:
        entry[1] = np.delete(entry[1], removed_features)

    return query_cp


def read_data(file_name: str) -> list:

    data_set = []
    with open(file_name, 'rt') as f:
        for line in f:
            line = line.replace('\n', '')
            tokens = line.split(',')
            label = tokens[0]
            attribs = []
            for i in range(784):
                attribs.append(tokens[i+1])
            data_set.append([label, np.array(attribs, dtype=float)])
    return (data_set)


def show(file_name, mode):

    data_set = read_data(file_name)
    for obs in range(len(data_set)):
        for idx in range(784):
            if mode == 'pixels':
                if data_set[obs][1][idx] == '0':
                    print(' ', end='')
                else:
                    print('*', end='')
            else:
                print('%4s ' % data_set[obs][1][idx], end='')
            if (idx % 28) == 27:
                print(' ')
        print('LABEL: %s' % data_set[obs][0], end='')
        print(' ')
```