MUHAMMAD NOMAN SHAHID

COWLAR ASSESSMENT PART 1

Image Classification of CMU-PIE Dataset Using Custom KNN and PCA

1. Introduction

The project aims to classify images from the CMU-PIE dataset using a custom K-Nearest Neighbors (KNN) classifier. The CMU-PIE dataset consists of 1700 rows of 32*32 images (1024 columns) in a CSV file. Each image is labeled from 0 to 9, with each label spanning over 170 images continuously. Thus, there are 10 labels and 170 rows for each label. The project also incorporates Principal Component Analysis (PCA) to reduce the dimensionality of the dataset before applying the KNN classifier.

NOTE: You can run the code from both the notebooks and the '*main.py*' file in the root directory. Please read the README.md files for further instructions and information on the difference between the two.

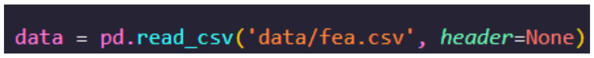
2. Project Structure

```
CMU-PIE/
├── data/            # Raw data (not included in the git repository)
|   └── fea.csv
|
├── notebooks/          # Jupyter notebooks
|   ├── cmu_pie_knn.ipynb
|   ├── cmu_pie_knn_pca.ipynb
|   └── visualize_results.ipynb
|
├── src/             # Source code
|   ├── __init__.py
|   ├── compute_knn.py
|   ├── data_processing.py
|   ├── knn.py
|   ├── pca_data.py
|   └── result_viz.py
|
├── tests/           # Unit tests
|   ├── __init__.py
|   ├── test_compute_knn.py
|   ├── test_data_processing.py
|   ├── test_knn.py
|   └── test_pca_data.py
|
├── results/            # Results in csv files
|   ├── knn_results.csv
|   └── knn_results_pca.csv
├── .gitignore
├── README.md
├── requirements.txt
├── Report.pdf
└── main.py
```

3. Data Processing

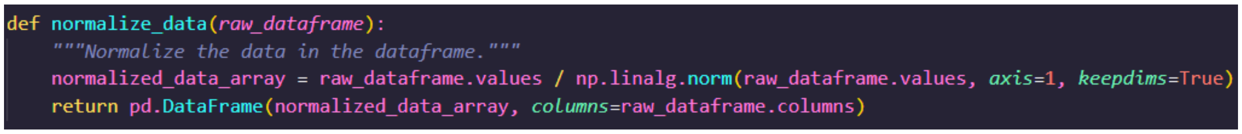
The data preprocessing stage involves several steps:

-Data Loading: The dataset is loaded from a CSV file using pandas.



```python
data = pd.read_csv('data/fea.csv', header=None)
```

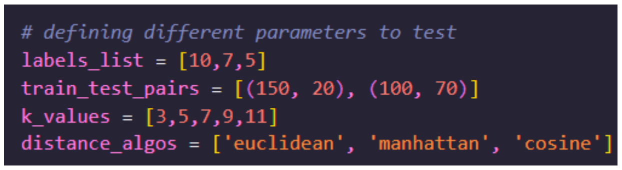
- Data Normalization: The data is normalized by dividing each row in the array by its norm, effectively normalizing each row.



```python
def normalize_data(raw_dataframe):
    """Normalize the data in the dataframe."""
    normalized_data_array = raw_dataframe.values / np.linalg.norm(raw_dataframe.values, axis=1, keepdims=True)
    return pd.DataFrame(normalized_data_array, columns=raw_dataframe.columns)
```

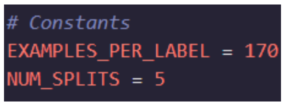
The function 'np.linalg.norm' is calculating the Euclidean norm across each row. In the return statement we could use both headers=None or what we have used.

- Define parameters for testing: We define different parameter lists that we want to test the data on. We test the algorithm on a combination of different number of labels (we have 10 in our original data), different number of training examples and test cases pairs and we test the data with different values of 'k' and different distance algorithms.



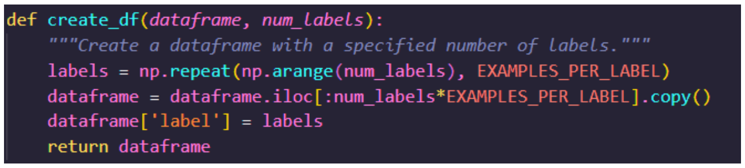
```python
# defining different parameters to test
labels_list = [10,7,5]
train_test_pairs = [(150, 20), (100, 70)]
k_values = [3,5,7,9,11]
distance_algos = ['euclidean', 'manhattan', 'cosine']
```

We also define some constants that help us create labels, dataframes and data splits in the next step.



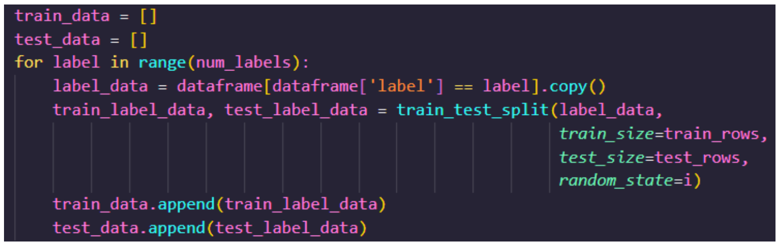
```python
# Constants
EXAMPLES_PER_LABEL = 170
NUM_SPLITS = 5
```

- Creating initial dataframes: After normalizing the data, the first thing we do is create a dataframe based on the number of labels we want. We truncate the dataframe and copy it over in this step. We also add the label column. We will be separating the labels before training but we add them here because it will help when we create data splits and shuffle them.



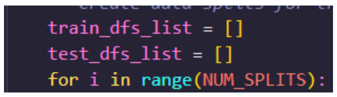
```python
def create_df(dataframe, num_labels):
    """Create a dataframe with a specified number of labels."""
    labels = np.repeat(np.arange(num_labels), EXAMPLES_PER_LABEL)
    dataframe = dataframe.iloc[:num_labels*EXAMPLES_PER_LABEL].copy()
    dataframe['label'] = labels
    return dataframe
```

- Splitting the data: This is a more complex step. Here we are creating training and testing lists. We loop over the number of labels we want to use (the amount of data we want to use for the model). Before this loop we initialize two empty lists that hold the training and testing data. We filter the data frame for the labels we are looking for and then use train_test_split to separate the data into training and testing. We give the random_state=i which changes each time this loop ends completely. The we append our training and testing arrays into the lists that we had initialized.



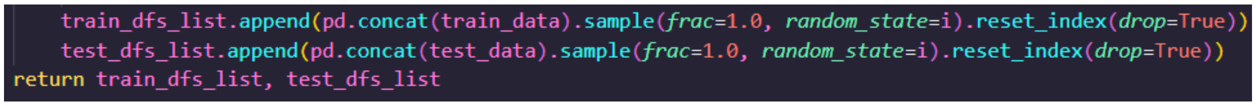
```python
train_data = []
test_data = []
for label in range(num_labels):
    label_data = dataframe[dataframe['label'] == label].copy()
    train_label_data, test_label_data = train_test_split(label_data,
                                                         train_size=train_rows,
                                                         test_size=test_rows,
                                                         random_state=i)
    train_data.append(train_label_data)
    test_data.append(test_label_data)
```

We have one loop outside this loop which loops over the range of NUM_SPLITS constant. So, for each outer loop, the inner loop runs to get all the data and then we use the concat function of pandas to create two dataframe out of it and append the dataframes to two other lists for training and testing which are initialized at the function start



```python
    train_dfs_list = []
    test_dfs_list = []
    for i in range(NUM_SPLITS):
```

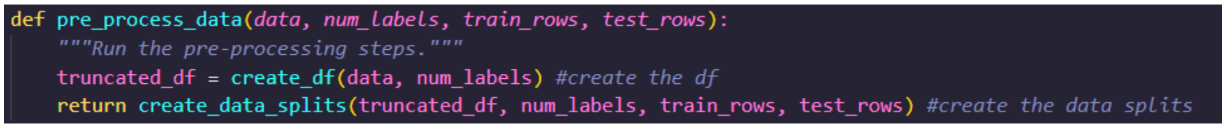
The inner loop shown above comes here.



```python
    train_dfs_list.append(pd.concat(train_data).sample(frac=1.0, random_state=i).reset_index(drop=True))
    test_dfs_list.append(pd.concat(test_data).sample(frac=1.0, random_state=i).reset_index(drop=True))
return train_dfs_list, test_dfs_list
```

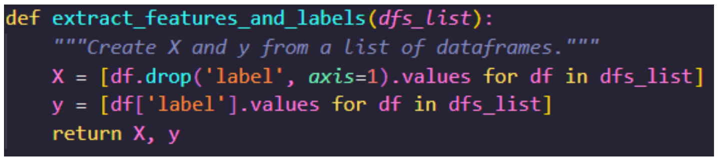
They are randomly sampled as well. So, the result we get is a list of 5 dataframes for each list. Each of the corresponding dataframe has the same number of training and test examples but they are sampled with a different combination every time.

- Combining it all: We use a function to call the dataframe creating function and the function to split it. This makes the next part of code more readable.



```python
def pre_process_data(data, num_labels, train_rows, test_rows):
    """Run the pre-processing steps."""
    truncated_df = create_df(data, num_labels) #create the df
    return create_data_splits(truncated_df, num_labels, train_rows, test_rows) #create the data splits
```

We also create a function to extract the feature and target values.

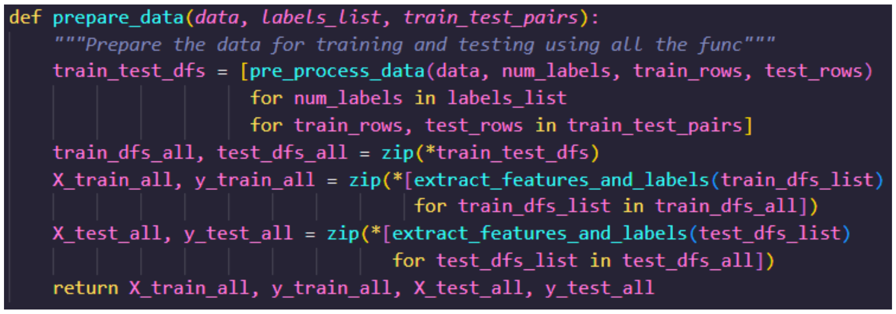


```python
def extract_features_and_labels(dfs_list):
    """Create X and y from a list of dataframes."""
    X = [df.drop('label', axis=1).values for df in dfs_list]
    y = [df['label'].values for df in dfs_list]
    return X, y
```

In the final function of this step, we create a list that runs the pre_process_data function for each num_label and each train_test_pair.

So, we have 6 different tuples for the 6 combinations, each of the tuple contains a pair of lists, one for training data and one for testing. Each of these lists have 5 different dataframes with a different random sample from the data. We use the zip function to separate all the training and test lists and put them in new lists.

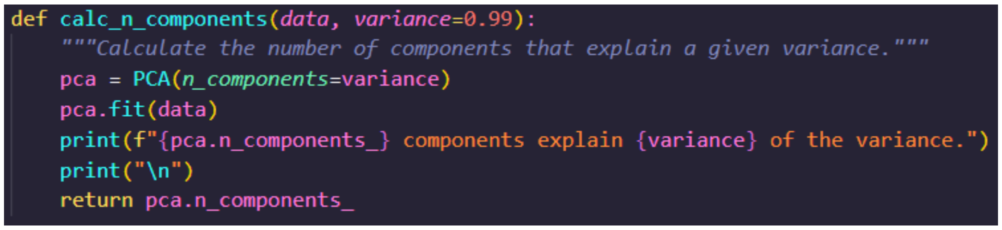
Then we separate each of the dataframes nested inside into features and target for both training and testing lists and turn our two lists into four.



```python
def prepare_data(data, labels_list, train_test_pairs):
    """Prepare the data for training and testing using all the func"""
    train_test_dfs = [pre_process_data(data, num_labels, train_rows, test_rows)
                      for num_labels in labels_list
                      for train_rows, test_rows in train_test_pairs]
    train_dfs_all, test_dfs_all = zip(*train_test_dfs)
    X_train_all, y_train_all = zip(*[extract_features_and_labels(train_dfs_list)
                                     for train_dfs_list in train_dfs_all])
    X_test_all, y_test_all = zip(*[extract_features_and_labels(test_dfs_list)
                                   for test_dfs_list in test_dfs_all])
    return X_train_all, y_train_all, X_test_all, y_test_all
```

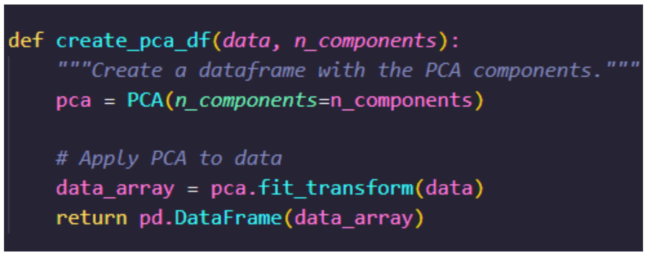
4. Principal Component Analysis (PCA)

PCA is used to reduce the dimensionality of the dataset. This whole step is done after the normalization of data and before any other step of creating dataframes or splitting data. The number of components that explain 99% of the variance in the data is calculated. The number of dimensions turns out to be **188.**



```python
def calc_n_components(data, variance=0.99):
    """Calculate the number of components that explain a given variance."""
    pca = PCA(n_components=variance)
    pca.fit(data)
    print(f"{pca.n_components_} components explain {variance} of the variance.")
    print("\n")
    return pca.n_components_
```

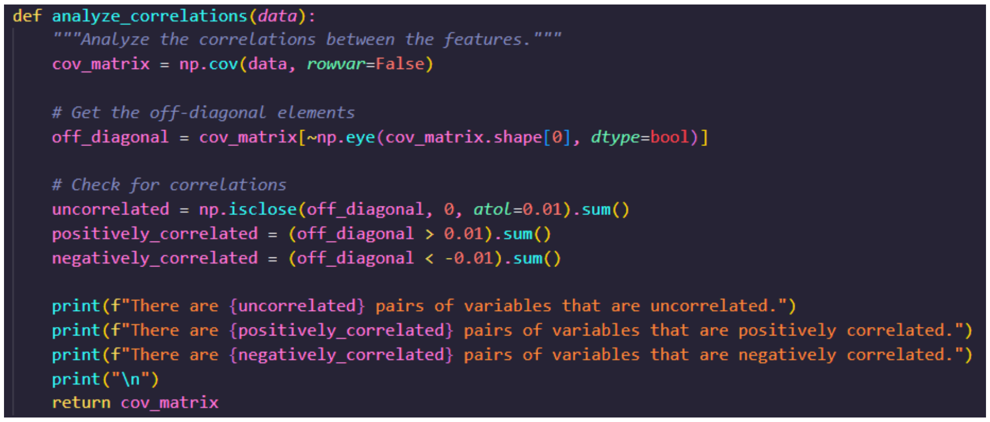
The data is then transformed into this reduced dimensionality space.



```python
def create_pca_df(data, n_components):
    """Create a dataframe with the PCA components."""
    pca = PCA(n_components=n_components)

    # Apply PCA to data
    data_array = pca.fit_transform(data)
    return pd.DataFrame(data_array)
```

We calculate the covariance matrix using np.cov. We check how many correlated and uncorrelated elements does the matrix have. We create an identity matrix with the same rows as the covariance matrix then filter the covariance matrix with the reverse of that so that we have the actual values at off-diagonal and the diagonals are zero. We check if they are 0, but we use absolute tolerance of 0.01 to counter floating point precision problem.



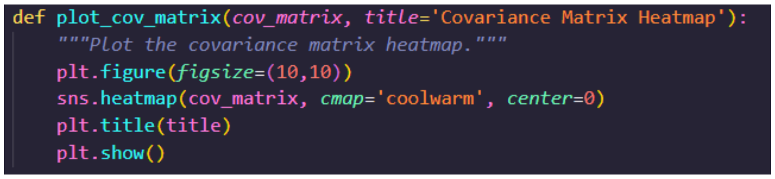
```python
def analyze_correlations(data):
    """Analyze the correlations between the features."""
    cov_matrix = np.cov(data, rowvar=False)

    # Get the off-diagonal elements
    off_diagonal = cov_matrix[~np.eye(cov_matrix.shape[0], dtype=bool)]

    # Check for correlations
    uncorrelated = np.isclose(off_diagonal, 0, atol=0.01).sum()
    positively_correlated = (off_diagonal > 0.01).sum()
    negatively_correlated = (off_diagonal < -0.01).sum()

    print(f"There are {uncorrelated} pairs of variables that are uncorrelated.")
    print(f"There are {positively_correlated} pairs of variables that are positively correlated.")
    print(f"There are {negatively_correlated} pairs of variables that are negatively correlated.")
    print("\n")
    return cov_matrix
```

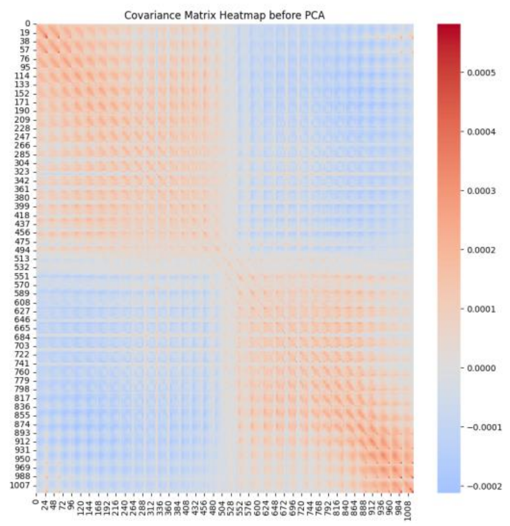
We plot the matrices before and after the pca transformation using the functio**n.**

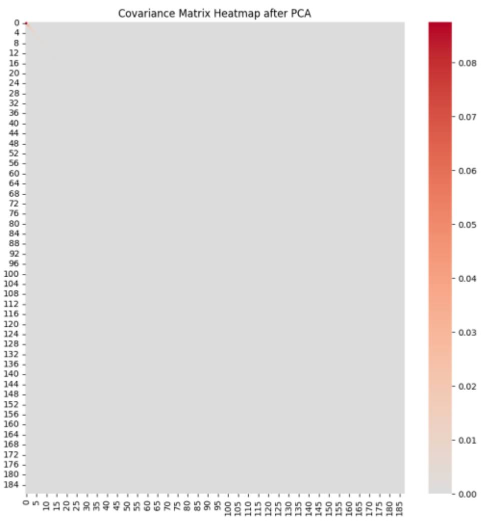


```python
def plot_cov_matrix(cov_matrix, title='Covariance Matrix Heatmap'):
    """Plot the covariance matrix heatmap."""
    plt.figure(figsize=(10,10))
    sns.heatmap(cov_matrix, cmap='coolwarm', center=0)
    plt.title(title)
    plt.show()
```

DOING PCA LEADS TO UNCORRELATED COVARIANCE MATRIX

Before vs After

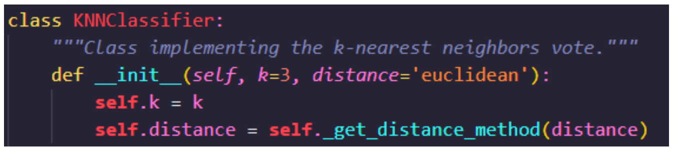




5. K-Nearest Neighbors (KNN) Classifier

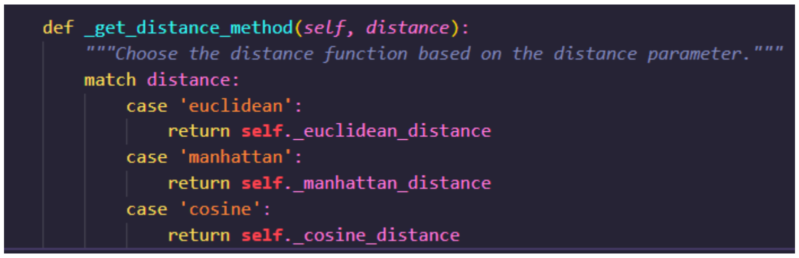
A custom KNN classifier is implemented.

- Initialization: The init method is defined with two instances i.e The value of k and the distance algorithm.



```
class KNNClassifier:
    """Class implementing the k-nearest neighbors vote."""
    def __init__(self, k=3, distance='euclidean'):
        self.k = k
        self.distance = self._get_distance_method(distance)
```

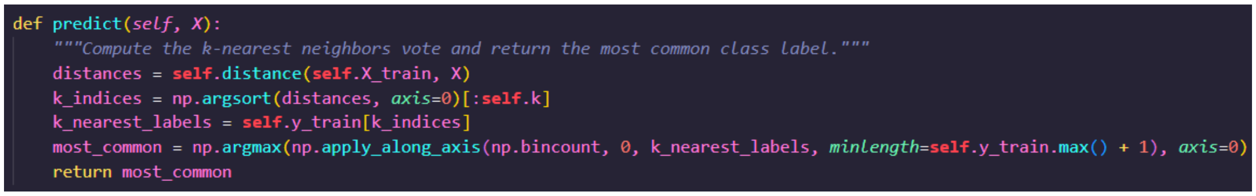
- Distance Metric Selection: This method uses a match-case structure to return the appropriate distance function based on the **distance** parameter.



```
    def _get_distance_method(self, distance):
        """Choose the distance function based on the distance parameter."""
        match distance:
            case 'euclidean':
                return self._euclidean_distance
            case 'manhattan':
                return self._manhattan_distance
            case 'cosine':
                return self._cosine_distance
```

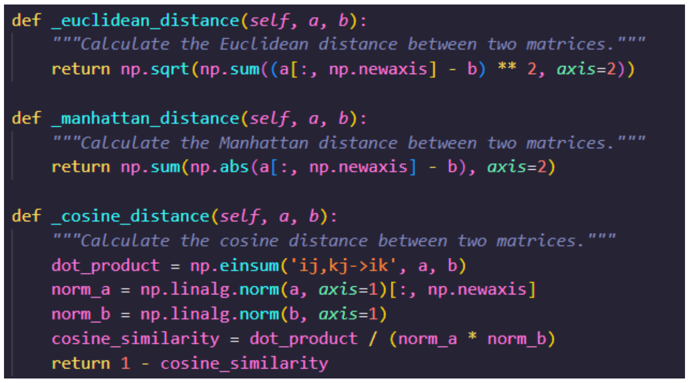
- Fitting the data: The fit method just stores the training data since there is no actual training in this model.

- Predict: It calculates the distances between the new data and all instances in the training data using the selected distance function. It then sorts these distances and selects the **k** nearest instances. The labels of these nearest instances are then used to make a prediction based on a majority vote.



```python
def predict(self, X):
    """Compute the k-nearest neighbors vote and return the most common class label."""
    distances = self.distance(self.X_train, X)
    k_indices = np.argsort(distances, axis=0)[:self.k]
    k_nearest_labels = self.y_train[k_indices]
    most_common = np.argmax(np.apply_along_axis(np.bincount, 0, k_nearest_labels, minlength=self.y_train.max() + 1), axis=0)
    return most_common
```

- Distance Calculation Methods: These methods are used to calculate the distance between instances. In the cosine distance we find the cosine similarity then subtract it from 1 to find the cosine distance.



```python
def _euclidean_distance(self, a, b):
    """Calculate the Euclidean distance between two matrices."""
    return np.sqrt(np.sum((a[:, np.newaxis] - b) ** 2, axis=2))

def _manhattan_distance(self, a, b):
    """Calculate the Manhattan distance between two matrices."""
    return np.sum(np.abs(a[:, np.newaxis] - b), axis=2)

def _cosine_distance(self, a, b):
    """Calculate the cosine distance between two matrices."""
    dot_product = np.einsum('ij,kj->ik', a, b)
    norm_a = np.linalg.norm(a, axis=1)[:, np.newaxis]
    norm_b = np.linalg.norm(b, axis=1)
    cosine_similarity = dot_product / (norm_a * norm_b)
    return 1 - cosine_similarity
```

We use broadcasting techniques and vector operations to perform most of our computations to make things efficient.

6. Testing

We create basic unit tests for most of the functions. We check things like, shape of the data, expected output and type of the data being returned. They also check the expected output of the model. We will not go to explain all the tests in this report. They are very straightforward and easy to understand. We have not created any tests for the visualizations since those require the checking of plots manually as well.

We use **pytest** and **pytest-cov** for testing. The tests are located in the **'tests'** directory.

7. Results Analysis

We have 5 different k values. For each of the k values we have 3 different distance measures. For each of the distance measure we have 3 different number of labels (amount of data). For each of the number of labels value we have 2 different training and test pairs. For each different training and test pair we create 5 random shuffles of data.

In total we run the knn function: 5 x 3 x 3 x 2 x 5 = 450 times

We average out the 5 random shuffles after applying knn so we have 90 rows in result file.

The results are analyzed by creating pivot tables and visualizing them in bar charts. The pivot tables show the mean of the average accuracy, standard deviation of accuracy, and computation time across different parameters. The bar charts visualize these pivot tables, making it easier to compare the performance of the classifier under different parameters.

EACH PLOT IS SORTED LEFT TO RIGHT WITH LEFT HAVING THE BEST VALUE: HGHEST ACCURACY, LOWEST STANDARD DEVIATION AND LOWEST COMPUTATION TIME

Here are the results **without** using ***PCA:***

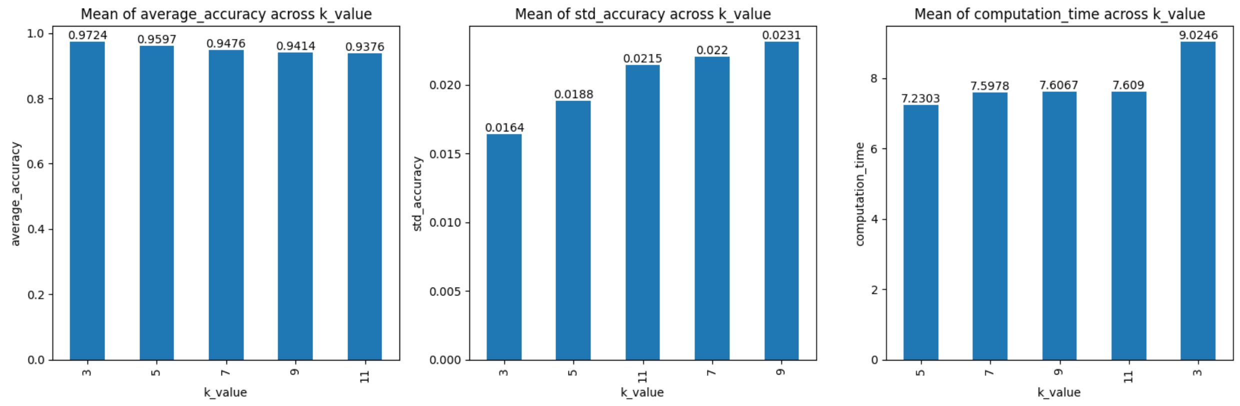


Fig A1) Mean of the accuracy, standard deviation and computation time over different k values

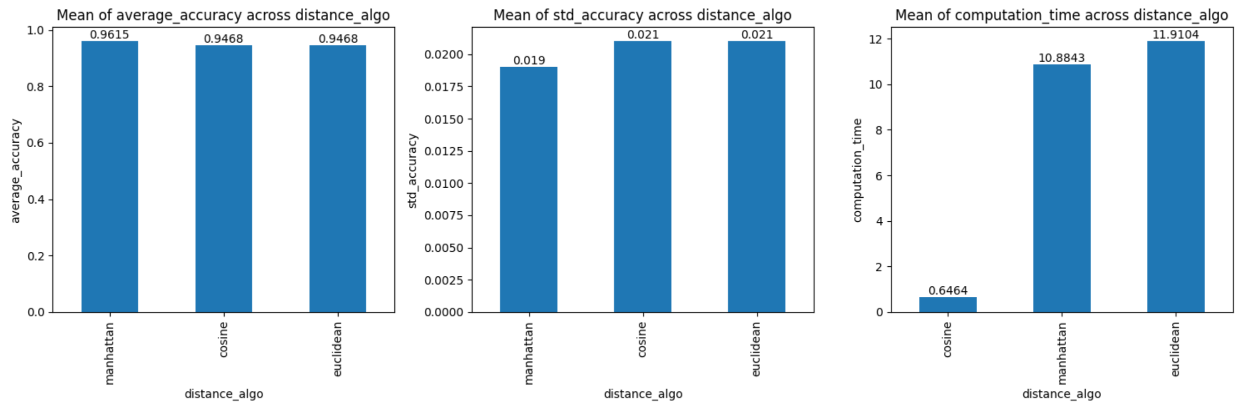


Fig A2) Mean of the accuracy, standard deviation and computation time over different distance metrics

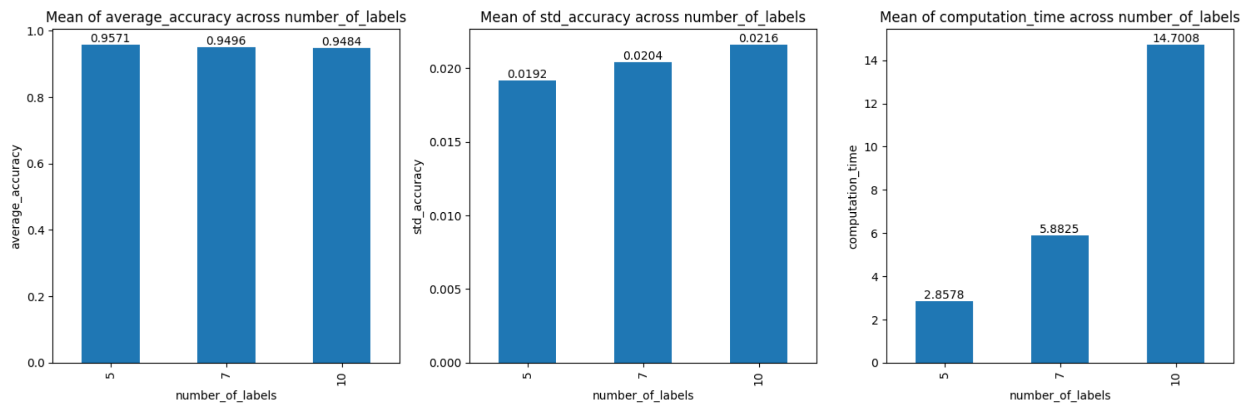


Fig A3) Mean of the accuracy, standard deviation and computation time over different number of labels

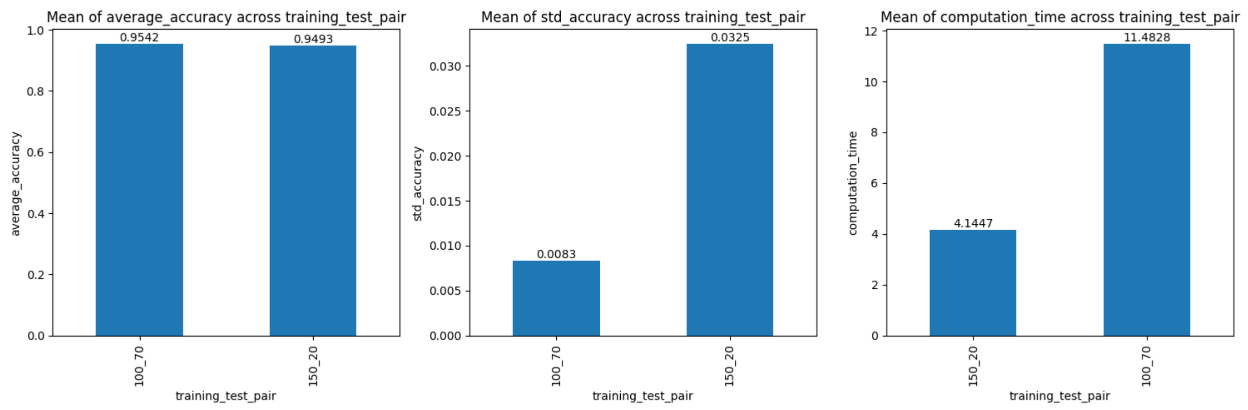


Fig A4) Mean of the accuracy, standard deviation and computation time over different train_test pairs

So, what we have above is the mean of our result parameters across different values that we tested. We keep 1 parameter constant (let's say the k value) and then take all the accuracies associated with that k value and take their mean. Our first figure (A1) shows this for each k value for the accuracy, standard deviation and computation time in seconds. It becomes much easier to understand if you look at the **result_viz.py** file in the code and the **csv** files in the **'results'** directory.

One thing you may have noticed that the labels for each plot say "mean of average_accuracy".

The "average" of the values come from the part where we take the average of 5 different shuffles of the data. We create a list of accuracies and then take the average to calculate the **average accuracy** over the five shuffles as well as the **standard deviation** over the accuracies and the **total computation time** in seconds that it takes to compute **ALL 5 SHUFFLES.** In simple terms:

Accuracy: Calculated as average of accuracies of 5 shuffles

Standard deviation: Calculated over the accuracies of 5 shuffles

Computation time (s): Calculated as a total for all 5 shuffles (it is not calculated as average of shuffles).

In the same way we have the results after applying ***PCA*** with ***188*** dimensions for ***99%*** variance.

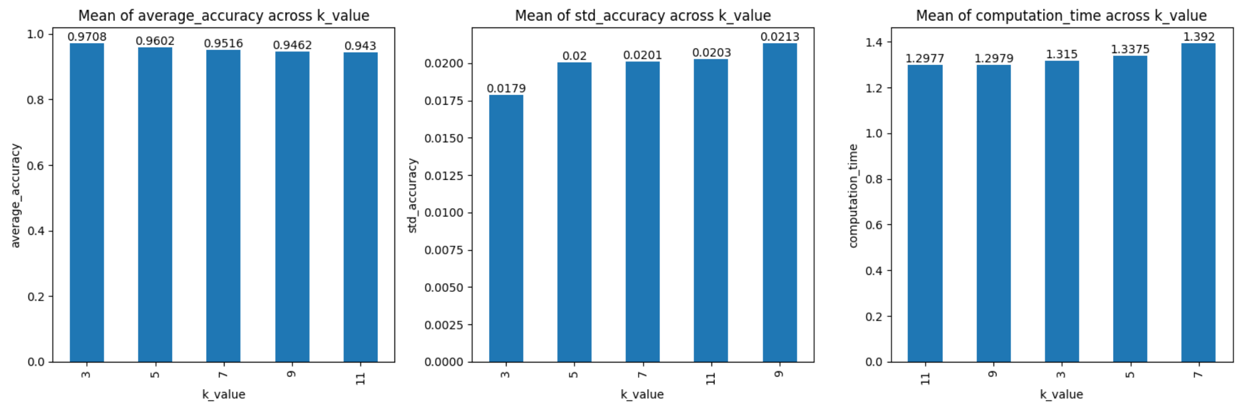


Fig B1) Mean of the accuracy, standard deviation and computation time over different k values

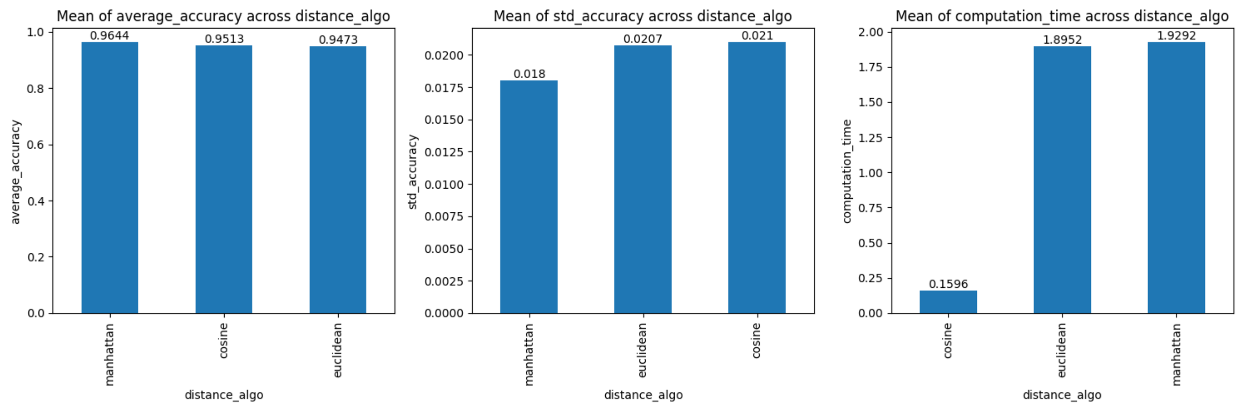


Fig B2) Mean of the accuracy, standard deviation and computation time over different distance metrics

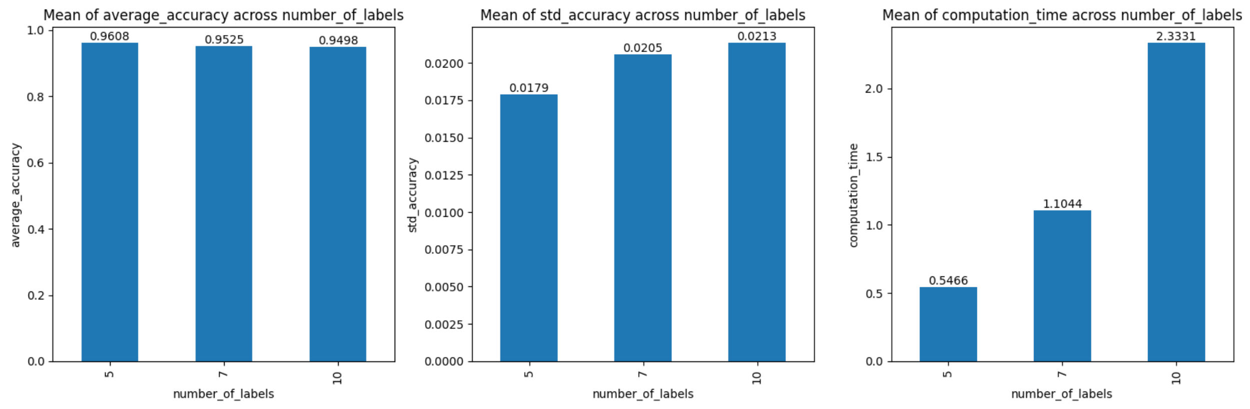


Fig B3) Mean of the accuracy, standard deviation and computation time over different number of labels

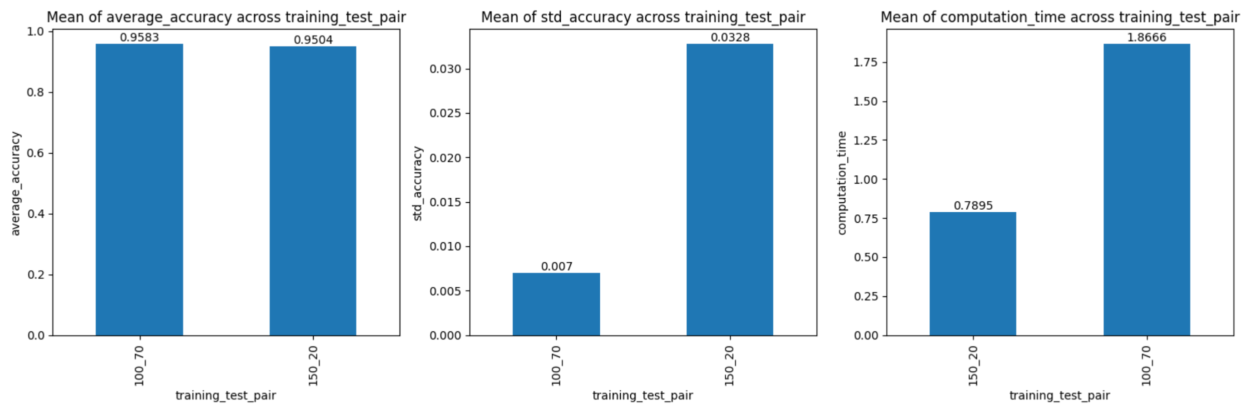


Fig B4) Mean of the accuracy, standard deviation and computation time over different train_test pairs

8. Conclusion

The project successfully implements a custom KNN classifier to classify images from the CMU-PIE dataset. The use of PCA for dimensionality reduction helps to improve the computational efficiency of the classifier. The accuracies only get lowered by negligible fractions but the computation time decreases many times.