2024

Machine Learning Assignment 2

RESULTS AND REPORT

EMMETT FITZHARRIS R00222357

Introduction

My main function is used to call the functions for each task. It also calls the read_data function to load information from the CSV file, and caches the file once read.

```python
def main():
    OutputFormat.print_header('title', 'Machine Learning Assignment 2')

    start_time = time.time()  # start measuring time

    # Create a cache folder if it does not exist
    if not os.path.exists(cache_folder):
        os.makedirs(cache_folder)

    original_data = read_data()
    df = original_data.copy()

    labels, features = task1(df)
    task2(labels, features, num_samples)
    task3(labels, features, num_samples)
    task4(labels, features, num_samples)
    task5(labels, features, num_samples)
    task6(labels, features, num_samples)
    # task7()

    print(f'\nTotal Runtime: {time.time() - start_time:.2f}s')

def read_data():
    cache_path = os.path.join(cache_folder, main_cache_filename)
    if os.path.exists(cache_path):
        with open(cache_path, 'rb') as f:
            df = pickle.load(f)
        print('Loaded data from cache\n')
    else:
        df = pd.read_csv(source_filename + file_extension, header=0)
        with open(cache_path, 'wb') as f:
            pickle.dump(df, f)
        print('Loaded data from csv file and cached it')
    return df

# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    main()
```

Task 1

In task 1, I start by slicing the data frame to only include our target labels, 5,7 and 9.

I then split the labels from the features using the separate_labels_and_features function.

```python
def task1(df):  1 usage  Emmett
    OutputFormat.print_header( section: 'h1',  text: 'Task 1: Pre-processing and Visualisation')

    target_df = df[df.iloc[:, 0].isin(TARGET_LABELS.keys())]
    labels, features = separate_labels_and_features(target_df)
    unique_labels = labels.unique()

    for label in unique_labels:
        first_instance = features[labels == label].iloc[0]
        display_image(label, first_instance)

    print('Unique labels:', unique_labels)
    print('Number of instances:', len(features))
    return labels, features
```

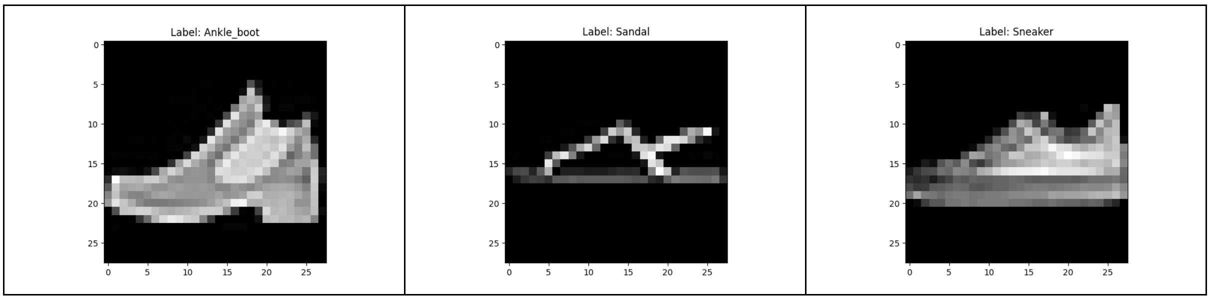
```python
TARGET_LABELS = {
    5: 'Sandal',
    7: 'Sneaker',
    9: 'Ankle_boot'
}
```

```python
def separate_labels_and_features(df):  1 usage  Emmett
    labels = df.iloc[:, 0]  # First column as labels
    features = df.iloc[:, 1:]  # All other columns as feature vectors
    return labels, features
```

The data in this file consists of 28x28 pixels, represented as greyscale brightness 0-255.

Using matplotlib, I create a plot the first of each unique class of labels, “Sandle”, “Ankle_boot” and “Sneaker”.

```python
def display_image(label, features):  1 usage  Emmett
    label = label
    pixels = features.values.reshape(28, 28)

    plt.figure()
    plt.title(f'Label: {TARGET_LABELS[label]}')
    plt.imshow(pixels, cmap='gray')
    plt.show()
```



Task 2

Task 2 is completed in the “run_classifier” function.

Firstly if I pass a value of number of samples, I take a sample of the feature set based on the parameter. If no parameter is passed, I default to using the full data set.

I create a kfold model, using the Sci-Kit Learn package, using 5 splits.

As training the models will be an expensive task in terms of processing power and time, next I implement multi-core processing using pools from the multiprocessing library. Rather than using a single core, I use all but two of the available cores on the system, so I can exploit more of the processing power available on my computer without overloading it.

The pools will call the “train_and_evaluate_fold” function, passing the “args” variable as parameters.

I collate the results for each fold, then print the accuracy, training time and prediction time, and print the confusion matric of each split.

I return the results back to the main function.

```
def run_classifier(labels, features, classifier, num_samples=None):  4 usages  Emmett

    if num_samples:
        features = features.sample(n=num_samples, random_state=42)
        labels = labels.loc[features.index]

    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    results = []
    y_tests = []
    y_preds = []

    num_processes = multiprocessing.cpu_count() - 2 # Use all but two cores to avoid overloading the system
    pool = multiprocessing.Pool(processes=num_processes)
    args = [(train_index, test_index, features, labels, classifier) for train_index, test_index in kf.split(features)]
    fold_results = pool.map(train_and_evaluate_fold, args)
    pool.close()
    pool.join()

    for i, (accuracy, training_time, prediction_time, y_test, y_pred) in enumerate(fold_results):
        results.append((accuracy, training_time, prediction_time))
        y_tests.append(y_test)
        y_preds.append(y_pred)

        OutputFormat.print_divider('h2')
        print(f'Fold {i + 1}: Accuracy: {accuracy:.2f}, Training Time: {training_time:.2f}s, Prediction Time: {prediction_time:.2f}s')
        OutputFormat.print_divider('h3')

        confusion_matrix(y_test, y_pred)


    results_df = pd.DataFrame(results, columns=['Accuracy', 'Training Time', 'Prediction Time'])
    return results_df
```

The “train_and_evaluate_fold” function prepares the data, implementing feature scaling to ensure uniformity.

I measure the time for training, and the time for prediction, then test for the accuracy of predictions against the test data.

```python
def train_and_evaluate_fold(args):
    train_index, test_index, features, labels, classifier = args
    X_train, X_test = features.iloc[train_index], features.iloc[test_index]
    y_train, y_test = labels.iloc[train_index], labels.iloc[test_index]

    # Feature scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    start_time = time.time()
    classifier.fit(X_train_scaled, y_train)
    training_time = time.time() - start_time

    y_pred = classifier.predict(X_test_scaled)
    prediction_time = time.time() - start_time - training_time

    accuracy = (y_pred == y_test).mean()
    return accuracy, training_time, prediction_time, y_test, y_pred
```

I created a function to print the confusion matrix, so I can call it for each split.

I also made one to extract a summary of the results data frame, which will be used later to find required information easily.

```python
def confusion_matrix(y_test, y_pred):
    print('Confusion Matrix\n')
    print(pd.crosstab(y_test, y_pred, rownames=['Actual Label'], colnames=['Predicted Label']), '\n')


def summary_results(results):
    summary_df = pd.DataFrame({
        'Min': results.min(),
        'Max': results.max(),
        'Average': results.mean()
    })
    return summary_df
```

I set the global variable "num_samples" to 10% of the total set data for development. The classifiers will be evaluated at a number of sample sizes.

```python
num_samples=1800
```

Task 2 does not produce any output itself, but will do so when called in future tasks

Task 3

In task 3 I implement the Perceptron Classifier. This is available from SciKit Learn, so the implementation is quite simple. I create an instance of the classifier, then pass it to the "run_classifier" function from task 2.

```
def task3(labels, features, num_samples=None):  1 usage  Emmett *
    OutputFormat.print_header( section: 'h1',  text: 'Task 3: Perceptron Classifier')
    classifier_name = 'Perceptron'

    classifier = Perceptron()
    results_df = run_classifier(labels, features, classifier, num_samples)

    summary_df = summary_results(results_df)

    print(f'Average Prediction Accuracy for {classifier_name}: {summary_df["Average"]["Accuracy"]:.2f}')

    plot_sample_size_vs_runtime(labels, features, classifier, classifier_name)
```

For each fold we see an output like the following:

```
********************************************************************************

Fold 1: Accuracy: 0.91, Training Time: 0.26s, Prediction Time: 0.00s

--------------------------------------------------------------------------------

Confusion Matrix

Predicted    5    7    9
Actual
5          105    3    7
7            6  110    5
9            8    3  113


********************************************************************************
```

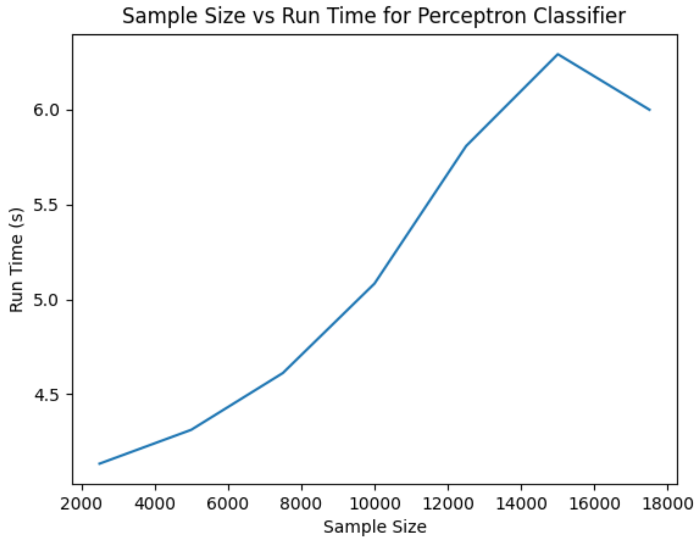
Once the classifier is trained and evaluated, we can see an accuracy score of 0.89 on average.

```
Average Prediction Accuracy for Perceptron: 0.89
```

Finally, I create a function which can be used to plot the sample size against the runtime of a classifier. This not only handles the diagram via matplotlib, but also the collection of the runtime information by looping through a set of sample sizes and calling the "run_classifier" function for each. We will use this function going forward both for this task and subsequent classifiers.

```
def plot_sample_size_vs_runtime(labels, features, classifier, classifier_name):  4 usages  Emmett *
    SampleSizeList = [2500, 5000, 7500, 10000, 12500, 15000, 17500]

    runtimes = []
    start_time = time.time()

    for new_num_samples in SampleSizeList:
        iteration_start_time = time.time()
        run_classifier(labels, features, classifier, new_num_samples)
        runtimes.append(time.time() - iteration_start_time)

        OutputFormat.progressbar(SampleSizeList.index(new_num_samples), len(SampleSizeList), start_time)

    plt.plot( *args: SampleSizeList, runtimes)
    plt.xlabel('Sample Size')
    plt.ylabel('Run Time')
    plt.title(f'Sample Size vs Run Time for {classifier_name} Classifier')
    plt.show()
```

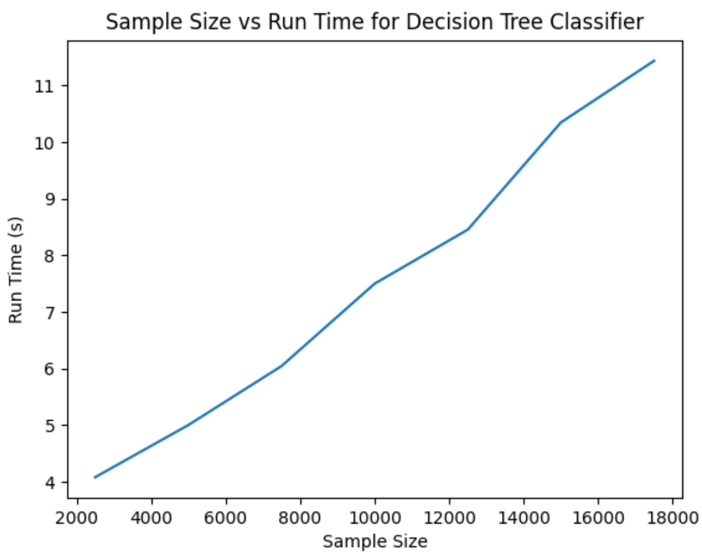
Based on the graph below, we can see that the run time increases slowly at first, then speeds up somewhat after a sample size of 10000. Oddly it seems that the run time dips after 15000, which may indicate that a decision boundary was hit earlier.



Task 4

The implementation of task 4 is almost identical to the implementation of task 3, this time using a Decision Tree Classifier model, also offered by Sci-Kit Learn. Once again I find the average accuracy for the default sample size, then plot the run time at varying sample sizes.

```
def task4(labels, features, num_samples=None):   1 usage   Emmett *
    OutputFormat.print_header('h1', 'Task 4: Decision Tree Classifier')
    classifier_name = 'Decision Tree'

    classifier = DecisionTreeClassifier()
    results_df = run_classifier(labels, features, classifier, num_samples)

    summary_df = summary_results(results_df)

    print(f'Average Prediction Accuracy for {classifier_name}: {summary_df["Average"]["Accuracy"]:.2f}')

    plot_sample_size_vs_runtime(labels, features, classifier, classifier_name)
```



Task 5

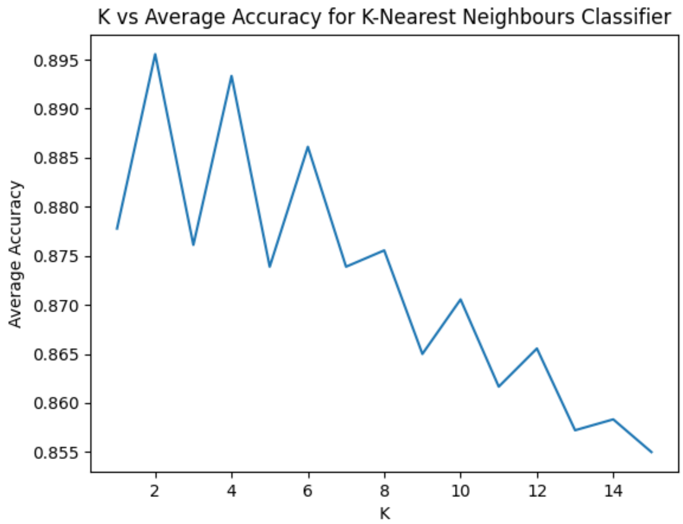
In task 5 we test the K-Nearest Neighbours Classifier. This time the implementation does change quite significantly from previous tasks.

I start by finding the optimal value for k . This is calculated in the "determine_best_k" function, which iterates through a range of values from 1 to 15, training a classifier model for each using the "run_classifier" command from task 2.

The iteration with the highest mean accuracy is taken to be the optimal value of k. A classifier with this value is passed to the "plot_sample_size_vs_runtime" function to gather data on the runtime at differing sample sizes as with the previous tasks.

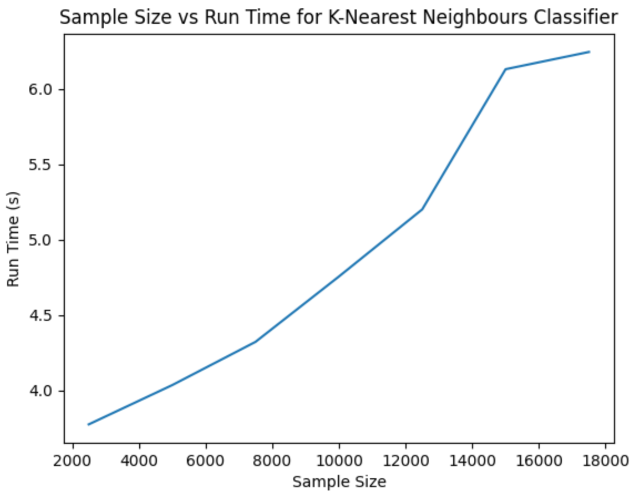
```python
def task5(labels, features, num_samples=None):
    OutputFormat.print_header('h1', 'Task 5: K-Nearest Neighbours Classifier')
    classifier_name = 'K-Nearest Neighbours'

    k = determine_best_k(labels, features, num_samples)
    classifier = KNeighborsClassifier(k)

    plot_sample_size_vs_runtime(labels, features, classifier, classifier_name)
```

```python
def determine_best_k(labels, features, num_samples=None):
    min_k = 1
    max_k = 15
    results = []

    k = min_k
    while k <= max_k:
        classifier = KNeighborsClassifier(k)
        results_df = run_classifier(labels, features, classifier, num_samples)
        summary_df = summary_results(results_df)
        results.append(summary_df['Average']['Accuracy'])
        k += 1

    plt.plot(range(min_k, max_k + 1), results)
    plt.xlabel('K')
    plt.ylabel('Average Accuracy')
    plt.title('K vs Average Accuracy for K-Nearest Neighbours Classifier')
    plt.show()

    best_k = results.index(max(results)) + 1
    print(f'Best K: {best_k} at {max(results):.2f} accuracy')

    return best_k
```



Based on the information calculated above, we can see the ideal value for k is 2 at an accuracy rating of of 0.9

```
Best K: 2 at 0.90 accuracy
```



Task 6

In task 6 I implemented the Support Vector Machine Classifier, using the radial basis function kernel.

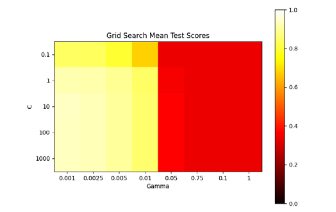
This kernal has two parameters, gamma and C, making the optimization a 2D problem. In the function “determine_best_y_value_for_rbf” I define a list of values for each parameter, in a dictionary. I will then implement a grid search to find the optimal values.

```python
def task6(labels, features, num_samples=None):
    OutputFormat.print_header('h1', 'Task 6: Support Vector Machine Classifier')

    best_params = determine_best_y_value_for_rbf(labels, features, num_samples)

    classifier = SVC(kernel='rbf', gamma=best_params['gamma'], C=best_params['C'])
    plot_sample_size_vs_runtime(labels, features, classifier, 'Support Vector Machine')
```

```python
def determine_best_y_value_for_rbf(labels, features, num_samples=None):
    if num_samples:
        features = features.sample(n=num_samples, random_state=42)
        labels = labels.loc[features.index]

    scaler = StandardScaler()
    features_scaled = scaler.fit_transform(features)

    # param_grid = {'gamma': [0.001, 0.01, 0.1, 1, 10, 100], 'C': [0.1, 1, 10, 100, 1000]}
    param_grid = {'gamma': [0.001, 0.0025, 0.005, 0.01, 0.05, 0.75, 0.1, 1], 'C': [0.1, 1, 10, 100, 1000]}
    svc = SVC(kernel='rbf')

    grid_search = GridSearchCV(svc, param_grid, cv=5, n_jobs= multiprocessing.cpu_count() - 1)
    grid_search.fit(features_scaled, labels)
    best_params = grid_search.best_params_

    plot_heatmap(grid_search.cv_results_, param_grid)
    print(f'Best parameters: {best_params}, with mean test score of {grid_search.best_score_:.2f}')

    return best_params
```

To visually confirm the results are sensible, I use a heatmap, plotting gamma against C, with the heat representing the accuracy.

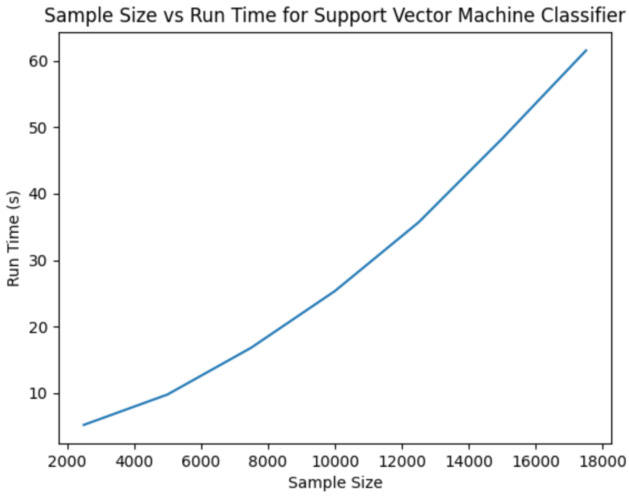
```python
def plot_heatmap(results, param_grid):
    mean_test_scores = results['mean_test_score'].reshape(len(param_grid['C']), len(param_grid['gamma']))

    plt.figure(figsize=(8, 6))
    plt.imshow(mean_test_scores, interpolation='nearest', cmap=plt.cm.hot, vmin=0, vmax=1)
    plt.xlabel('Gamma')
    plt.ylabel('C')
    plt.colorbar()
    plt.xticks(np.arange(len(param_grid['gamma'])), param_grid['gamma'])
    plt.yticks(np.arange(len(param_grid['C'])), param_grid['C'])
    plt.title('Grid Search Mean Test Scores')
    plt.show()
```



Using the above method, It is determined that the best parameters are C of 10 and a Gamma of 0.001.

```
Best parameters: {'C': 10, 'gamma': 0.001}, with mean test score of 0.94
```

As with the tasks before, I then plot varying sample sizes against the run time for this model:



Task 7

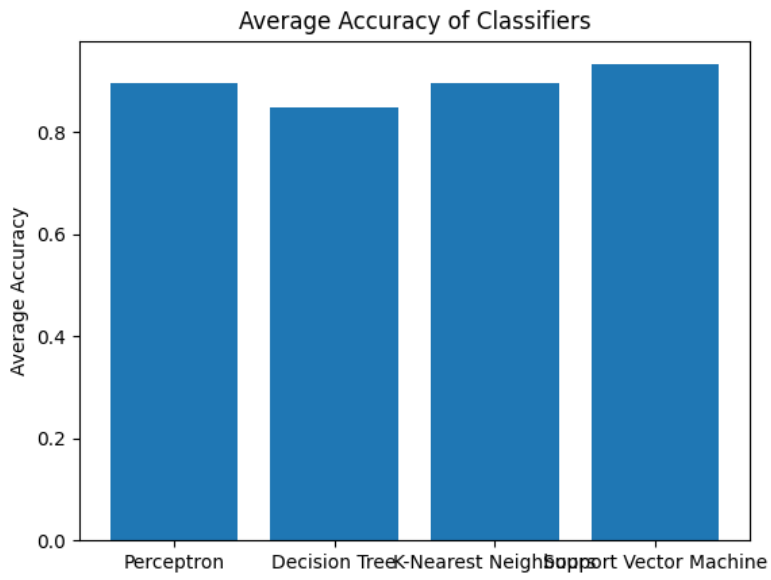
In task 7, I compare each of the above four classifiers, Perceptron, Decision Tree, K-Nearest Neighbour, and Support Vector Machine.

I start by printing their summaries, and follow up by creating three basic bar charts, the first comparing the accuracy of each classifier, the second comparing the training time, and the third comparing the prediction time of each classifier.

```python
def task7(perceptron_summary, decision_tree_summary, knn_summary, svm_summary):
    OutputFormat.print_header('h1', 'Task 7: Classifier Comparison')

    print('Perceptron Summary\n', perceptron_summary)
    print('Decision Tree Summary\n', decision_tree_summary)
    print('K-Nearest Neighbours Summary\n', knn_summary)
    print('Support Vector Machine Summary\n', svm_summary)

    # Plotting the average accuracy, of each classifier
    plt.figure()
    plt.bar(['Perceptron', 'Decision Tree', 'K-Nearest Neighbours', 'Support Vector Machine'],
            [perceptron_summary['Average']['Accuracy'], decision_tree_summary['Average']['Accuracy'],
             knn_summary['Average']['Accuracy'], svm_summary['Average']['Accuracy']])
    plt.ylabel('Average Accuracy')
    plt.title('Average Accuracy of Classifiers')
    plt.show()

    # Plotting the average training time, of each classifier
    plt.figure()
    plt.bar(['Perceptron', 'Decision Tree', 'K-Nearest Neighbours', 'Support Vector Machine'],
            [perceptron_summary['Average']['Training Time'], decision_tree_summary['Average']['Training Time'],
             knn_summary['Average']['Training Time'], svm_summary['Average']['Training Time']])
    plt.ylabel('Average Training Time')
    plt.title('Average Training Time of Classifiers')
    plt.show()

    # Plotting the average prediction time, of each classifier
    plt.figure()
    plt.bar(['Perceptron', 'Decision Tree', 'K-Nearest Neighbours', 'Support Vector Machine'],
            [perceptron_summary['Average']['Prediction Time'], decision_tree_summary['Average']['Prediction Time'],
             knn_summary['Average']['Prediction Time'], svm_summary['Average']['Prediction Time']])
    plt.ylabel('Average Prediction Time')
    plt.title('Average Prediction Time of Classifiers')
    plt.show()
```

```
################################################################################
                          Task 7: Classifier Comparison
################################################################################

Perceptron Summary
                      Min       Max   Average
Accuracy         0.875000  0.916667  0.895000
Training Time    0.160742  0.276662  0.191651
Prediction Time  0.001019  0.063202  0.013908
Decision Tree Summary
                      Min       Max   Average
Accuracy         0.816667  0.875000  0.848333
Training Time    0.355337  0.413229  0.380120
Prediction Time  0.000765  0.000914  0.000801
K-Nearest Neighbours Summary
                      Min       Max   Average
Accuracy         0.855556  0.913889  0.895556
Training Time    0.006886  0.012647  0.009254
Prediction Time  0.203265  0.233599  0.213323
Support Vector Machine Summary
                      Min       Max   Average
Accuracy         0.925000  0.941667  0.932778
Training Time    0.257166  0.488791  0.406442
Prediction Time  0.082371  0.137565  0.094298
```

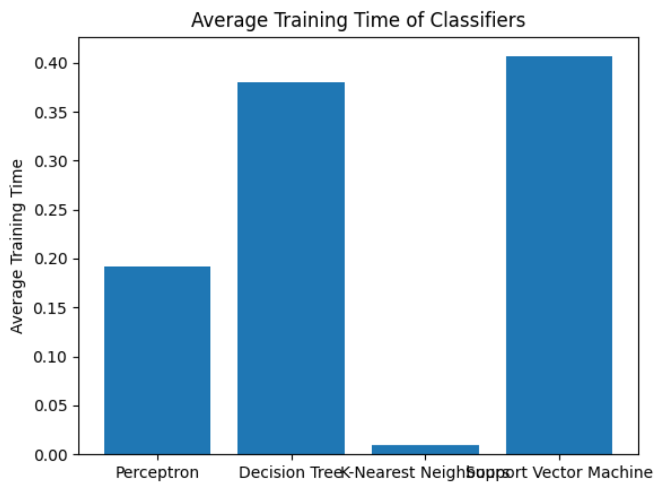
In terms of accuracy, we can see that all models performed within a close margin. The lowest accuracy being Decision tree, and the highest being Support Vector Machine.
This intuitively makes sense, due to the high dimensionality of the data.



For training time we see quite a bit of variance, with high training times for Decision tree and SVM. Decision trees are recursive, which causes a rapid rise in training time as depth increases. SVM models are computationally complex as they handle high degrees of complexity.

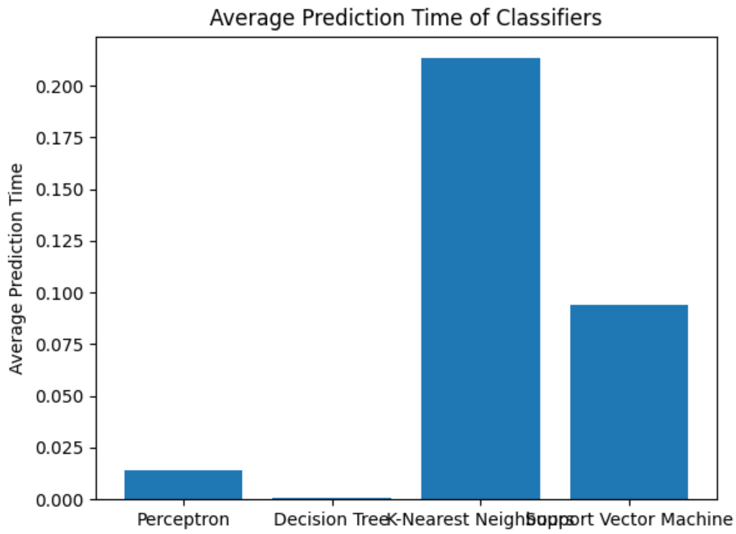
The Nearest Neighbour model has virtually no training time, which makes sense, as this model is just based on distance in prediction. It doesn't require training.

The perceptron model is less computationally expensive, and as such the training time is much lower.



Finally, in terms of prediction time, we see that the nearest neighbour model takes the longest by far, which is expected as the distance between nodes is calculated at prediction time.

The decision tree and perceptron models are almost instantaneous, which is expected as their prediction methods are very simple.



Overall, I would recommend the Support vector machine model if training time is not an obstacle, as it offers the highest accuracy, with a middling prediction time.

Aside

Building on my formatting functions from the last assignment, this time I implemented a class with static methods. This was used to add dividers and headers to the output, as well as provide a progress bar to show visual progress on intensive tasks.

```
#########################################################################################################################
# Output Formatting
#########################################################################################################################
class OutputFormat:  14 usages  Emmett
    SECTION_DIVIDER = {
        'title': '█',
        'h1': '#',
        'h2': '*',
        'h3': '-'
    }

    def __init__(self):  Emmett
        pass

    @staticmethod  8 usages  Emmett
    def print_header(section, text):
        print('\n' + OutputFormat.SECTION_DIVIDER[section] * 80)
        print(text.center(80, ' '))
        print(str(OutputFormat.SECTION_DIVIDER[section] * 80) + '\n')

    @staticmethod  2 usages  Emmett
    def print_divider(section):
        print("\n" + OutputFormat.SECTION_DIVIDER[section] * 80 + "\n")

    @staticmethod  1 usage  Emmett
    # Function to display a progress bar so the user knows the program is still running and how far along it is
    def progressbar(i, upper_range, start_time):
        # Calculate the percentage of completion
        percentage = (i / (upper_range - 1)) * 100
        # Calculate the number of '█' characters to display
        num_blocks = int(percentage/2)
        # Calculate elapsed time and estimated remaining time

        elapsed_time = time.time() - start_time
        if percentage > 0:
            estimated_total_time = elapsed_time / (percentage / 100)
            remaining_time = estimated_total_time - elapsed_time
```

The total runtime of the program is approximately 9 minutes.

```
██████████████████████████████████████████████████ 100.00% | Elapsed: 202.66s | Remaining: 0.00s

Total Runtime: 536.63s

Process finished with exit code 0
```