Quentin JONNEAUX
Student Number : R00274704
COMP 8043 - MACHINE LEARNING

ASSIGNMENT 2 - SUPERVISED LEARNING
A STUDY OF CLASSIFIER RUNTIMES

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Table of content

Summary

The file “fashion-mnist_train.csv” contains down-sampled product images from Zalando.com. Every row consists of a label and 28x28 8-bit grayscale pixel values of the product image. The goal of this assignment is to evaluate and optimise the performance of different classifiers for their suitability to classify this dataset.

We are testing our classifier on the data of Ankle boots, Sneakers and Sandals. We are preprocessing the data so that it contains only these classes. We then have created a 5-fold cross-validation procedure to have more consistency on accuracies and runtimes. We are then testing each classifiers with different fractions of the preprocessed sample to observe their pattern on runtimes and also implemented hyper-parameter tuning processes before testing the K-nearest neighbour and the Support Vector Machine.

The **Perceptron** is not the worst classifier with a mean accuracy of 0.91. Our study seems to indicate being the quickest, however we observed a variance in terms of runtime and training, as convergence is not reach solely based on sample size (number of iteration changing).

The **Decision Tree** is the less accurate with a mean accuracy of 0.89 (the worst of the comparison). It is also the second slowest in terms of runtime and training times due to the node computation but it is the quickest to make predictions.

The **KNN** performs reasonably well with a mean accuracy of 0.93. It is also very quick at training and predicts generally in less than a second. We found out the best hyper-parameter as 2 nearest neighbours.

We can see that **SVM** is the most accurate with a mean accuracy of 0.96 but it is at the significant trade of time because it is generally the slowest in general (runtime, training, predicting). We found out the best hyper-parameter as gamma parameter 1e-6.

If runtime is not to be considered, we would recommend the **SVM as best classifier** as it captures the non-linear relationships in the dataset, and performs very well on unseen data.

If runtime is to be considered, we can recommend the **KNN as second best classifier** as it is a relatively simple and intuitive algorithm to train and use. It generally runs in about a second and onboards training data instantly.

Required libraries and functions

```
# Importing necessary libraries
import pandas as pd # Pandas for data reading and manipulation
import numpy as np # Numpy for mathematical calculation and manipulating arrays
from sklearn import metrics # Scikit Learn metrics for computing confusion matrices and accuracy scores
from sklearn import model_selection # Scikit Learn model_selection to apply K-Fold cross validation
import matplotlib.pyplot as plt # matplotlib to display images
import datetime # datetime to create timestamps
from sklearn import linear_model # Scikit Learn linear model for creating a Perceptron
from sklearn import tree # Scikit Learn tree for creating a decision tree
from sklearn import neighbors # Scikit Learn neighbors for creating a KNN
from sklearn import svm # Scikit Learn svm for creating a support vector machine

```

Several libraries are needed to deliver this project.

We are using pandas for reading and manipulating the dataset but also, as additional step, create other data frames summarizing each classifiers performance with metrics for different sample sizes.

Numpy is used for calculation and computing arrays to train the classifier and evaluate it. Also matplotlib is used mainly to create lineplots to visualise the performance of each classifier against different samples. The datetime module is used to create timestamps to monitor training durations, prediction durations and runtimes.

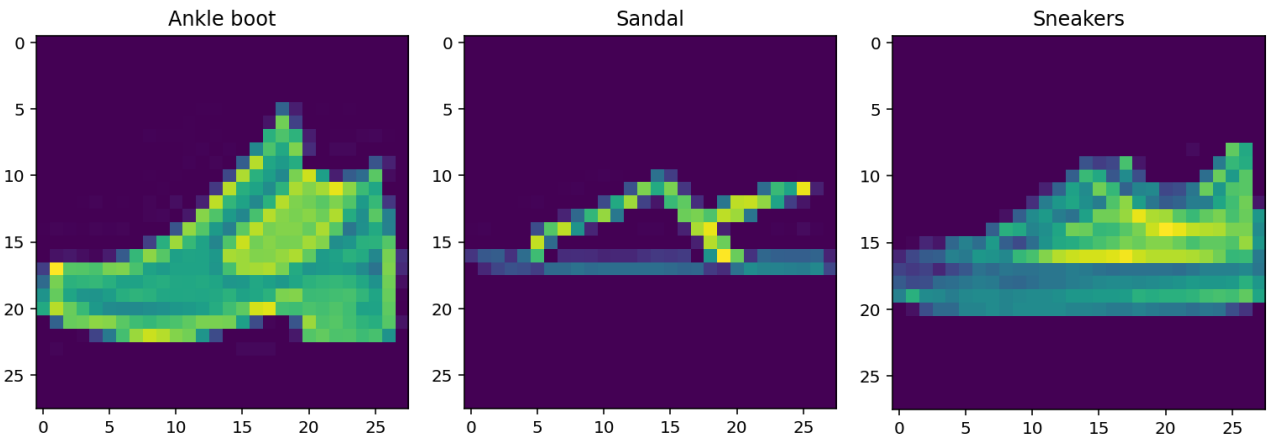
Finally, we import the different modules of Scikit Learn for creating classifiers, create the cross-validation procedure and compute accuracies:
- linear_model to create a Perceptron
- tree to create a decision tree
- neighbours to create a K-nearest neighbour
- svm to create a Support Vector Machine
- metrics to compute accuracies
- model_selection to create a K-Fold cross validation procedure

Task 1 - Pre-processing and visualisation

```python
# Task1 - Pre-processing and visualisation
# Define a function to preprocess the data and provide a visual of an object of each label
def preprocess():
    # Read the data from the csv file
    df = pd.read_csv('/Users/Quentin/Desktop/Hdip Data Science and Analytics/Year 1/Semester 3/COMP8043 - Machine Learning/Assignment 2/fashion-mnist_train.csv')
    # Filter for specific labels
    data = df[df['label'].isin([5,7,9])]

    # Make labels and feature subset
    labels = data['label']
    features = data.drop('label', axis=1)

    # Declare lists of articles and article type
    article_list = []
    article_titles = []

    # For each unique labels
    for label in labels.unique():
        # Associate correct title
        if label == 9:
            article_titles.append('Ankle boot')
        elif label == 5:
            article_titles.append('Sandal')
        elif label == 7:
            article_titles.append('Sneakers')
        # get the indexes of the articl type
        article_indexes = labels[labels == label].index
        # Extract the features at correct indexes
        article_features = features.loc[article_indexes]
        # Get first article features
        first_article_pixels = article_features.loc[article_indexes[0]]
        # append the features to the article list
        article_list.append(first_article_pixels)

    # For each article in the list
    i=0
    for article in article_list:
        # Create a plot and display the article image
        plt.figure()
        plt.title(article_titles[i])
        plt.imshow(article.to_numpy().reshape(28,28))
        i+=1

    # Return pre-processed features and labels
    return features,labels
```

For the pre-processing, we would like to focus on ankle boots, sandals and sneakers. The preprocess function, after reading through the dataset, we subset the rows based on labels (5 for sandals, 7 for sneakers and 9 for ankle boots) and then separate features (pixels) and labels (class).

To confirm the pre-processing, we display the first row of each label subset to make sure the images are corresponding each class. We can see, for each class, we have one instance of type, as shown below:



Task 2 - Evaluation Procedure

We create the evaluate function to be used for each classifier. It will allow to compute metrics of for each classifier and each sample size for our comparison analysis.

We first convert labels and features to numpy arrays for computing scores. We declare lists to store training, prediction and classifier runtimes to compute metrics. We split the feature and labels into test and training set to implement a k-fold cross validation procedure to have more consistency in output and performance. We are splitting into 5 folds, with 80% of data for training. In order to vary the sample sizes, we introduce a parameter n, so that we can replicate procedure, varying the sample size.

Using the datetime module, we record runtimes in seconds, compute confusion matrices and accuracies to be stored in the list. Using those lists, we can provide metrics:
- the training time per training sample
- the prediction time per evaluation sample
- and the prediction accuracy

We also perform a final evaluation using the whole training set and predict the test set and compute a confusion matrix, the final evaluation accuracy and fit and predict process runtime.

The function definition and a sample output can be found in Appendix 1.

Task 3 - Perceptron Classifier

We are creating the perceptron_classify function to assess the runtimes and the accuracies as well as experiment with different sizes.

We first create a instance of perceptron, apply the cross validation procedure on the whole data set. We have a **mean accuracy of 0.91** (which is not too bad). We reapply the same process with different fractions of the sample size (1/20, 2/20, ...) to produce 20 datapoints for the runtimes to compare them. We can then produce a line plot to analyze the relationship between sample sizes and runtimes.

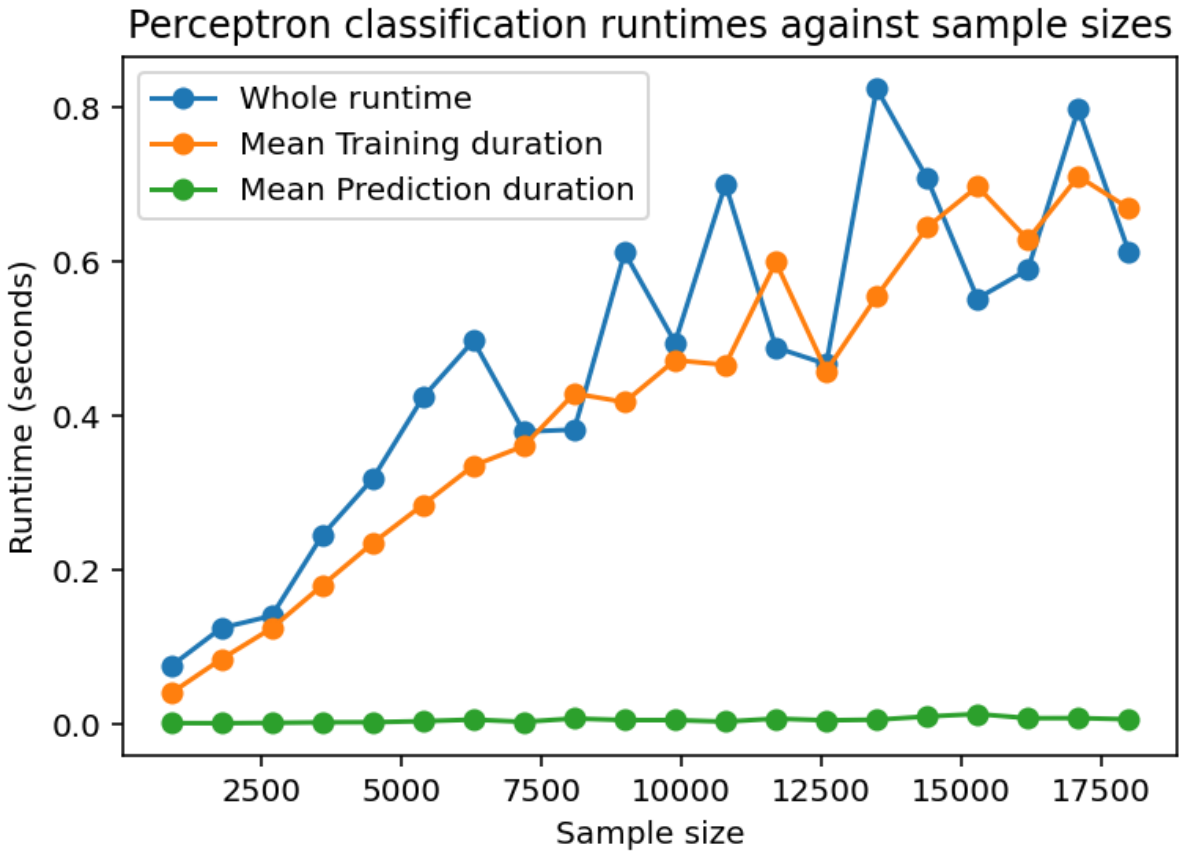


Fig 1: Line plot of runtimes against sample size of the Perceptron classifier

We can observe a relationship that seems to be linear, with a variance that seem to increase with sample size. The mean prediction duration are close 0.01 seconds. The maximum runtimes of the whole process (training + predicting) or training process do not seem however to at the largest sample size (whole data).

It is expected from a perceptron that runtimes will increase with sample sizes as each feature is weighted and then pass through a an activation function. Those are more sensitive with larger datasets as weights needs to address more nuances of the data. We may explain the variance in the computation of those weight and function.

While the perceptron seems an accurate and quick classifier (maximum runtime is less than a second), it seems that runtime performance will be more easily controlled on small sample sizes than on large sample (if we think a second difference is significant) and high dimensionality.

The function definition and the metrics table can be found in Appendix 2.

Task 4 - Decision Tree Classifier

We are creating the tree_classify function to assess the runtimes and the accuracies as well as experiment with different sizes.

We first create a instance of decision tree, apply the cross validation procedure on the whole data set. We have a **mean accuracy of 0.89 (which is not great)**. We reapply the same process with different fractions of the sample size (1/20, 2/20, ...) to produce 20 datapoints for the runtimes to compare them. We can then produce a line plot to analyze the relationship between sample sizes and runtimes.

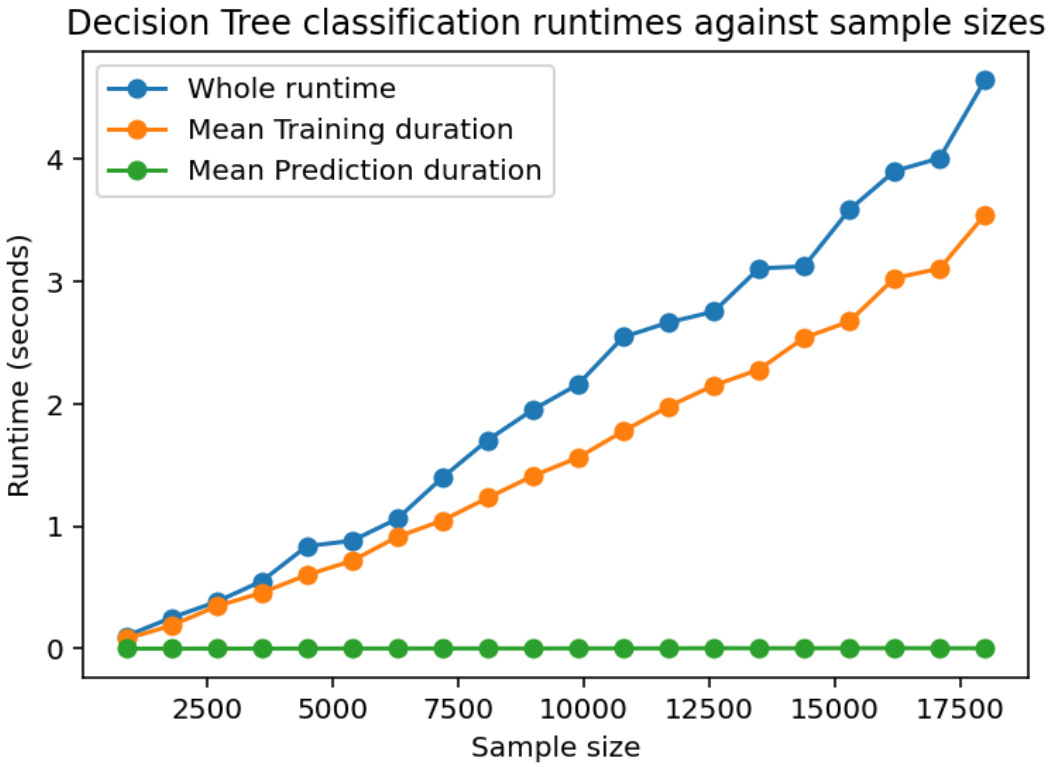


Fig 2: Line plot of runtimes against sample size of the Decision Tree classifier

We can observe a relationship that seems to be logarithmic, with a variance that does not seem to increase with sample size. The mean prediction duration are close 0.01 seconds. The maximum runtimes of the whole process (training + predicting) or training process seem to be at the largest sample size (whole data).

It is expected from a decision tree that runtimes will increase with sample sizes as each node of the tree needs to be computed along with gains according to more data points. We may explain the relationship with sample size as more calculation are required as the sample size increases.

While the Decision seems an accurate and quick classifier (maximum runtime is less than 5 seconds), we can predict the runtimes with more precision and can still handle large sample sizes.

The function definition and the metrics table can be found in Appendix 3.

Task 5 - K-nearest neighbour Classifier

We are creating the knn_classify function to assess the k parameter, the runtimes and the accuracies as well as experiment with different sizes.

We first create a loop to train and evaluate a KNN classifier with parameters comprised between 1 and 9 and store the accuracy of each. It seems the **K parameter 2** is providing the best accuracy (0.93)

We then create a instance of KNN with the **K parameter 2**, apply the cross validation procedure on the whole data set. We have a **mean accuracy of 0.93 (which is also good)**. We reapply the same process with different fractions of the sample size (1/20, 2/20, ...) to produce 20 datapoints for the runtimes to compare them. We can then produce a line plot to analyze the relationship between sample sizes and runtimes.

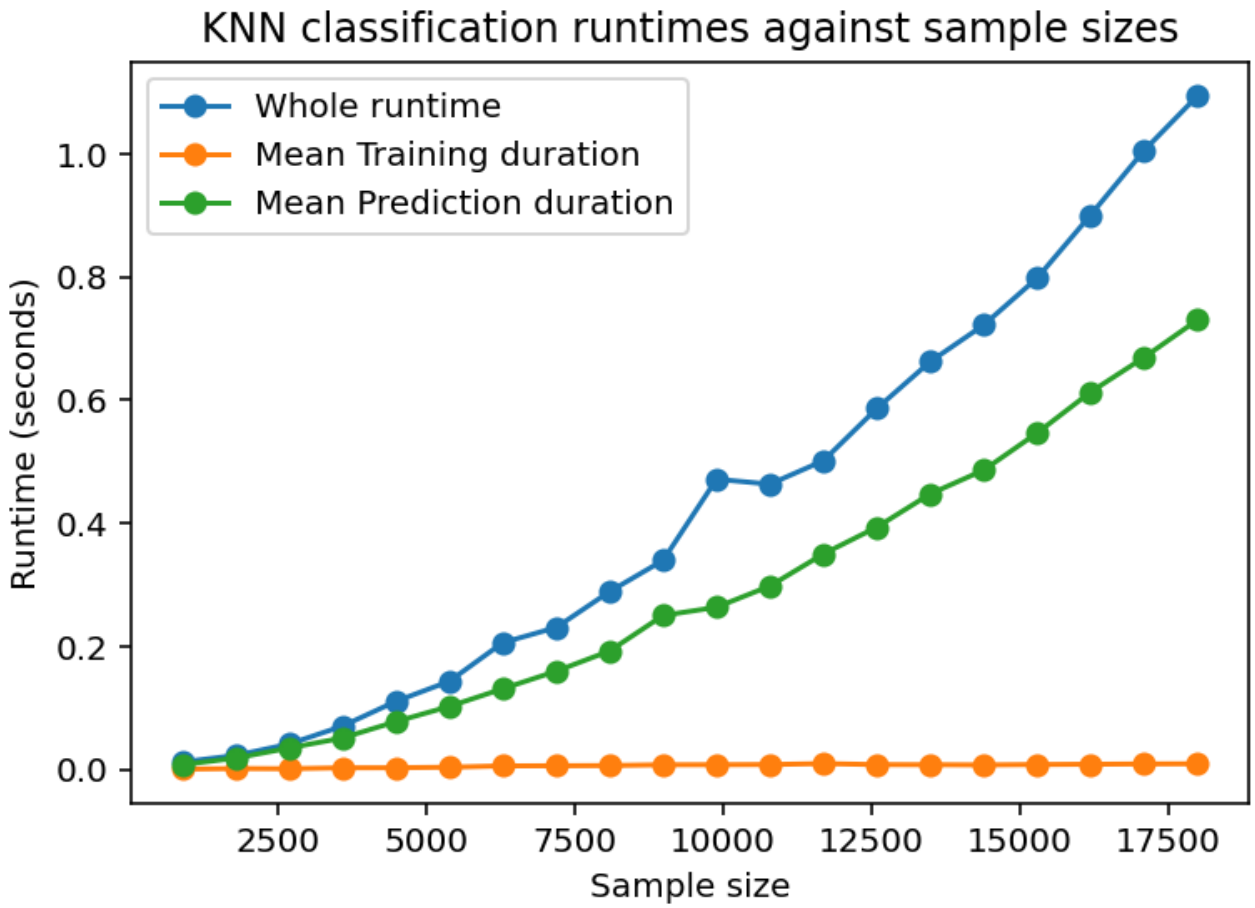


Fig 3: Line plot of runtimes against sample size of the KNN classifier

We can observe a relationship that seems to be logarithmic, with a variance that does not seem to increase with sample size. The mean training duration are close 0.01 seconds. The maximum runtimes of the whole process (training + predicting) or training process seem to be at the largest sample size (whole data).

It is expected from a KNN that training is much quicker than prediction as the training phase of a KNN consist of gather the data only. Calculation happens at the prediction time to find the 2

nearest neighbours in our of the same class. So the more data with have a training phase, the more calculation to do.

While the KNN seems an accurate and quick classifier (maximum runtime is less than 1.5 seconds), we can predict the runtimes with precision and can still handle large sample sizes, with low variance on runtime. The strong advantage is that the training phase is the quickest. However we have to keep in mind that, in this instance we determined a small k parameter (2) to provide best accuracy. When experimenting, a higher k parameter would increase the runtimes.

The function definition and the metrics table can be found in Appendix 4.

Task 6 - Support Vector Machine Classifier

We are creating the svm_classify function to assess the gamma parameter, the runtimes and the accuracies as well as experiment with different sizes.

We first create a loop to train and evaluate a SVM classifier with parameters comprised between 1e-9 and 1e0 and store the accuracy of each. It seems the **gamma parameter 1e-6** is providing the best accuracy (0.96)

We then create a instance of SVM with the **gamma parameter 1e-6**, apply the cross validation procedure on the whole data set. We have a **mean accuracy of 0.96 (which is very good)**. We reapply the same process with different fractions of the sample size (1/20, 2/20, ...) to produce 20 datapoints for the runtimes to compare them. We can then produce a line plot to analyze the relationship between sample sizes and runtimes.

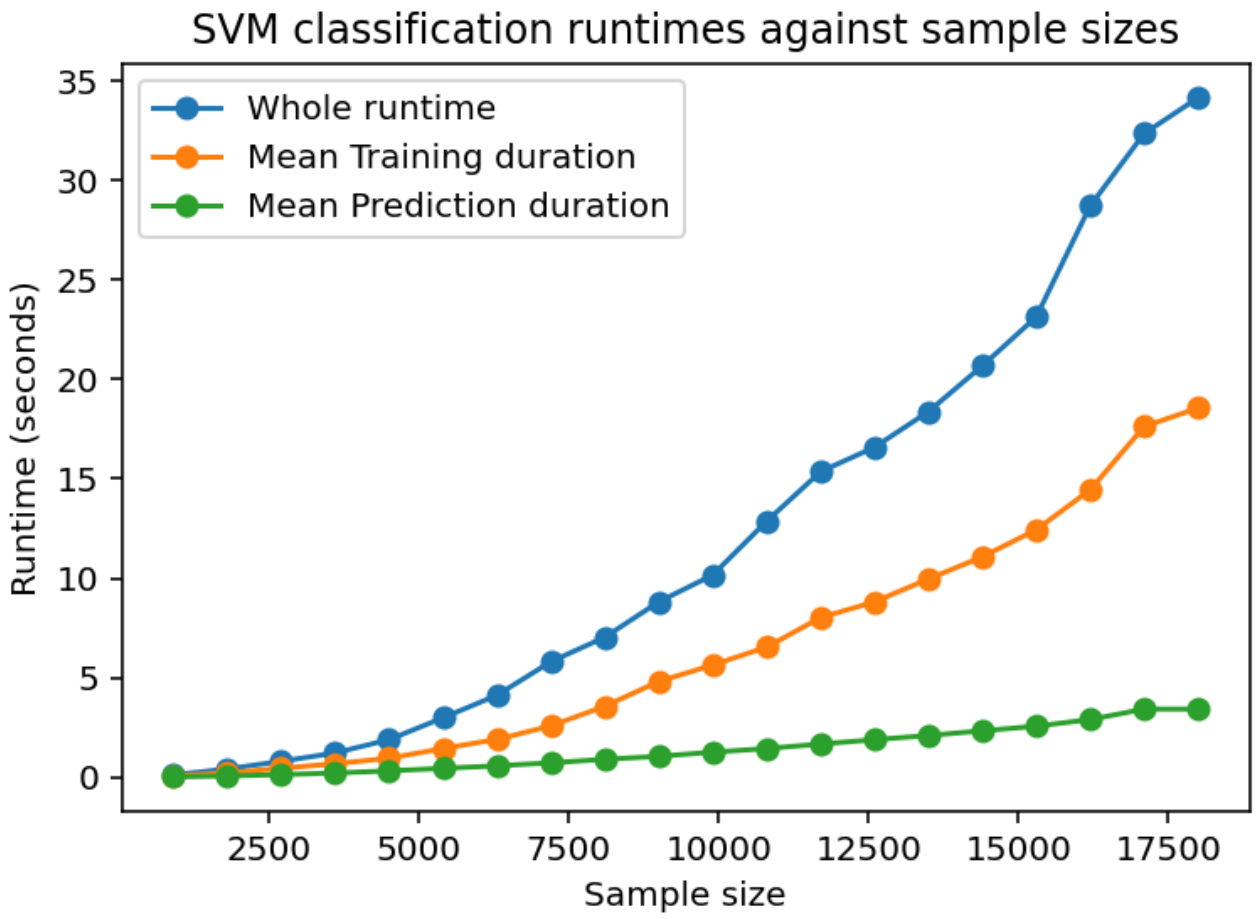


Fig 4: Line plot of runtimes against sample size of the SVM classifier

We can observe a relationship that seems to be logarithmic, with a variance that seems to increase with sample size. The mean training and prediction durations increase with sample size and the maximum runtimes of the whole process (training + predicting) or training process seem to be at the largest sample size (whole data). SVM runtimes increase also drastically with sample size (maximum mean runtime recorded is 35 secs, much more than other classifiers above).

It is expected from a SVM that it performs very well as the MNIST dataset as it is highly dimensional (28*28 pixels). The more dimension there is, the easier it is the fit hyperplanes to separate classes. However this accuracy comes at a cost of much more computation with size as more calculation is needed to take every data has to assessed if they are support vectors.

While the SVM seems the most accurate it takes much longer to perform. However, unlike the perceptron, variance still allows to predict runtime, based on data size.

The function definition and the metrics table can be found in Appendix 5.

Task 7 - Classifiers comparison

Perceptron

For the perceptron, we can observe a positive linear relationship between the mean training duration and the sample size. This is explained as each input needs to be weighted and pass through an activation function to be weighted, in order to determine the class. More data points would mean, more input and therefore more computation to develop the weight sensitivity,

We also notice an increased variance in the training runtime. The perceptron learning algorithm is as follows:
- Start with an initial weight vector
- Repeat until convergence:
  - Extract the set of misclassified examples
  - Update weights

The intuition of this procedure is that the weights are adjusted based on how much the corresponding feature contributes to the misclassification.The learning rate controls the speed of convergence. This procedure will converge if the samples are linearly separable.

In other words, the perceptron repeats iterations after experiencing with a set of weights and reassess them until convergence. Varying the sample size therefore affects the first definitions of those weights, therefore the set of misclassified examples and therefore, the number of iterations needed for convergence.

On the other hand, once the model is trained, it is very quick to make the prediction as we simply weight the input, aggregate them and pass this aggregation to an activation function.

Our perceptron **mean accuracy is 0.91** and, despite variance, seems reasonably quick to implement (maximum training duration recorded is 0.71 seconds and max prediction time recorded is 0.01 second).

Decision Tree

For the decision tree, we can observe a positive linear relationship between the mean training duration and the sample size. This is explained as the attributes splitting the classes needs to be calculated, therefore, the more data points, the more calculation needed to determine the attribute, especially that the can change as more information is passed.

High Level Steps for Creating a Decision Tree:
- Begin with a dataset containing all attributes and resulting classes
- Find the attribute that best splits the dataset class
- Divide the data in groups (leaves) based on the attribute that you have used to split (node)
- With each leaf find the attribute to best split the data set class (note only applies to data for leaf subgroup). Continue this process.

In contrast to the Perceptron, the decision tree display less variance in the training phase, which is expected as, since the dimensionality is not changing but only the sample size, only the calculations takes longer to determine best attribute splitting and defining the node. The number of nodes in the decision tree does not increase or decrease with sample size but with the attribute themselves.

In addition, once the model is trained, it is very quick to make the prediction as we simply derive attributes and follow the tree made in training phase.

Our decision tree **mean accuracy is 0.89** and, seems reasonably quick to implement (maximum training duration recorded is 3.54 seconds and max prediction time recorded is 0.005 second).

K-nearest neighbour

For the KNN, we can observe a positive logarithmic relationship between the mean prediction duration and the sample size while the training duration remains close to 0 seconds. This is explained as the KNN simply takes the training data is and make computation only at the stage of prediction. In our case, since the best k parameter is 2, it computes the 2 nearest neighbours of the same class to make its prediction, meaning if there are more data points, more calculation is needed.

The nearest neighbour rule is the simple idea to assign a new label based on the label of the nearest neighbour in feature space .

In contrast to the Perceptron and decision tree, since the data is taken as is in training phase, most runtime duration involve making predications rather than training the classifier.

Our KNN **mean accuracy is 0.93** and, seems reasonably quick to implement (maximum training duration recorded is 0.013 seconds and max prediction time recorded is 0.74 second).

Support Vector Machine

For the KNN, we can observe a positive logarithmic relationship between the mean training duration and the sample size and also between prediction duration and sample size, but with a less important slope. The runtimes themselves, however, appear much longer than the other classifiers tested (maximum training durations recorded at 20.71 seconds and whole runtime at 34.14 seconds). This explained by the dimensionality of the dataset (28x28 pixels making 784 dimensions of the feature space).

The support vector machine computes the optimal hyperplane that best separates the two classes is the hyperplane that maximises the margin, i.e. the distance between the plane and the closest points in the two classes. In other words, increasing sample size increase the calculation of the hyperplane parameters as there are more vectors (data points) to take into account.

In contrast to the Perceptron, decision tree and KNN, the dimensionality is strongly affecting the SVM, increasing significantly training runtimes with sample size.

Our KNN **mean accuracy is 0.96** and, seems much longer to implement (maximum training duration recorded is 20.71 seconds and max prediction time recorded is 4.47 seconds).

Final comparison

Metric	Perceptron	Decision Tree	KNN	SVM	Best	Worst
Mean accuracy	0.91	0.89	0.93	0.96	SVM	Decision Tree
Max runtime	0.79	4.64	1.1	34.14	Perceptron	SVM
Max Training duration	0.71	3.54	0.013	20.71	KNN	SVM
Max Prediction duration	0.01	0.005	0.74	4.47	Decision tree	SVM

Tab 1: Comparison table of classifier metrics

We can see that SVM is the most accurate with a mean accuracy of 0.96 but it is at the significant trade of time because it is generally the slowest in general (runtime, training, predicting).

The Decision Tree is the less accurate with a mean accuracy of 0.89 (the worst of the comparison). It is also the second slowest in terms of runtime and training times due to the node computation but it is the quickest to make predictions.

The KNN performs reasonably well with a mean accuracy of 0.93. It is also very quick at training and predicts generally in less than a second.

Finally, the perceptron is not the worst classifier with a mean accuracy of 0.91. Our study seems to indicate being the quickest, however we observed a variance in terms of runtime and training, as convergence is not reach solely based on sample size (number of iteration changing).

If runtime is not to be considered, we would recommend the **SVM as best classifier** as it captures the non-linear relationships in the dataset, and performs very well on unseen data.

If runtime is to be considered, we can recommend the **KNN as second best classifier** as it is a relatively simple and intuitive algorithm to train and use. It generally runs in about a second and onboards training data instantly.

Appendix 1 - Evaluation Procedure

Definition:

```
################################################################################
# Task2 - evaluation procedure
# Create a k-fold evaluation procedure to be used by each classifier
def evaluate(features,labels,n,clf):

    # Compute Numpy arrays for features and labels
    labels=labels.to_numpy()
    features = features.to_numpy()

    # Declare lists for training and predictions durations and accuracies
    all_train_durations = []
    all_pred_durations = []
    allResults = []


    # Split set into training set (80% of data) and test set (20% of data) with n sample of data
    test_features, train_features, test_labels, train_labels = model_selection.train_test_split(features[:n],
                                                                                                labels[:n],
                                                                                                test_size=0.8,
                                                                                                random_state=0)



    # Creating a K-Fold cross validation procedure (Using 5 folds, shuffling indexes, setting random_state as 42 for reproducibility)
    kf = model_selection.KFold(n_splits=5, shuffle=True, random_state=42)

    i = 1
    # For each train and test indexes in the K-fold procedure split
    for train_index, test_index in kf.split(train_features, train_labels):
        # For each split, train the classifier and record training duration
        print('Split: ',i)
        train_start = datetime.datetime.now()
        clf.fit(train_features[train_index], train_labels[train_index])
        train_end = datetime.datetime.now()
        train_duration = (train_end-train_start).total_seconds()
        print("Processing time required for training: ",train_duration)
        all_train_durations.append(train_duration)

        # Record prediction duration
        prediction_start = datetime.datetime.now()
        prediction = clf.predict(train_features[test_index])
        prediction_end = datetime.datetime.now()
        prediction_duration = (prediction_end - prediction_start).total_seconds()
        print('processing time required for prediction: ',prediction_duration)
        all_pred_durations.append(prediction_duration)

        # Compute a confusion matrix and accuracy score
        print('confusion matrix: ',metrics.confusion_matrix(train_labels[test_index], prediction))
        # print('accuracy score of the classification: ',metrics.accuracy_score(train_labels[test_index], prediction))
        allResults.append(metrics.accuracy_score(train_labels[test_index], prediction))
        i+=1

    # Compute minimum, maximum and average training duration
    min_train_duration = min(all_train_durations)
    print('minimum training time per training sample: ',min_train_duration)
    max_train_duration = max(all_train_durations)
    print('maximum training time per training sample: ',max_train_duration)
    mean_train_duration = np.mean(all_train_durations)
    print('average training time per training sample: ',mean_train_duration)

    # Compute minimum, maximum and average prediction duration
    min_pred_duration = min(all_pred_durations)
    print('minimum prediction time per evaluation sample: ',min_pred_duration)
    max_pred_duration = max(all_pred_durations)
    print('maximum prediction time per evaluation sample: ',max_pred_duration)
    mean_pred_duration = np.mean(all_pred_durations)
    print('average prediction time per evaluation sample: ',mean_pred_duration)

    # Compute minimum, maximum and mean accuracies
    min_accuracy = min(allResults)
    print('minimum prediction accuracy: ',min_accuracy)
    max_accuracy = max(allResults)
    print('maximum prediction accuracy: ',max_accuracy)
    mean_accuracy = np.mean(allResults)
    print('average prediction accuracy: ',mean_accuracy)

    # Compute a runtime using the whole set
    start_runtime = datetime.datetime.now()
    clf.fit(train_features, train_labels)
    prediction = clf.predict(test_features)
    end_runtime = datetime.datetime.now()
    runtime = (end_runtime - start_runtime).total_seconds()

    # Display confusion matrix and accuracy score on final evaluation
    print('Final confusion matrix: ',metrics.confusion_matrix(test_labels, prediction))
    final_accuracy = metrics.accuracy_score(test_labels, prediction)
    print('Final accuracy score of the classification: ',metrics.accuracy_score(test_labels, prediction))
    print('Classifier runtime: ',runtime)

    # Return metrics computed
    return min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy


################################################################################
```

Output:

minimum training time per training sample: 4.532327
maximum training time per training sample: 4.718156
average training time per training sample: 4.6074150000000005
minimum prediction time per evaluation sample: 1.475564
maximum prediction time per evaluation sample: 1.501669
average prediction time per evaluation sample: 1.4933094
minimum prediction accuracy: 0.941358024691358
maximum prediction accuracy: 0.9537037037037037
average prediction accuracy: 0.9466049382716049
Final confusion matrix: [[519 10   4]
 [18 519 13]
 [11 12 514]]
Final accuracy score of the classification: 0.9580246913580247
Classifier runtime: 9.387084
Split: 1
Processing time required for training: 5.7986
processing time required for prediction: 1.815194
confusion matrix: [[448   9   6]
 [15 487 10]
 [14 10 441]]
Split: 2
Processing time required for training: 5.744912
processing time required for prediction: 1.784127
confusion matrix: [[474 11   3]
 [17 464 20]
 [13 15 423]]
Split: 3
Processing time required for training: 5.752294
processing time required for prediction: 1.805085
confusion matrix: [[465 12   1]
 [13 440 18]
 [12 16 463]]
Split: 4
Processing time required for training: 5.682398
processing time required for prediction: 1.791992
confusion matrix: [[470 12   2]
 [11 461 24]
 [8 16 436]]
Split: 5
Processing time required for training: 5.737485
processing time required for prediction: 1.770781
confusion matrix: [[464   7   2]
 [17 416 14]
 [7 16 497]]
minimum training time per training sample: 5.682398
maximum training time per training sample: 5.7986
average training time per training sample: 5.7431377999999995
minimum prediction time per evaluation sample: 1.770781
maximum prediction time per evaluation sample: 1.815194
average prediction time per evaluation sample: 1.7934358
minimum prediction accuracy: 0.9451388888888889
maximum prediction accuracy: 0.95625
average prediction accuracy: 0.9512500000000002
Final confusion matrix: [[570 15   5]
 [25 582 15]
 [18 21 549]]
Final accuracy score of the classification: 0.945
Classifier runtime: 11.35197

Appendix 2 - Perceptron

Definition:

```
########################################################################
# Task3 – Perceptron
# Define a function to train and evaluate a Perceptron classifer
def perceptron_classify(features,labels):
    # Instantiate a Perceptron
    clf = linear_model.Perceptron()

    # Make an evaluation of the perceptron and display the mean accuracy
    min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = evalu
    print("Perceptron Mean Accuracy: ",mean_accuracy)

    # Declare lists to store sample sizes and metrics
    sample_sizes = []
    min_train_durations = []
    max_train_durations = []
    mean_train_durations = []
    min_pred_durations = []
    max_pred_durations = []
    mean_pred_durations = []
    runtimes = []
    min_accuracies = []
    max_accuracies = []
    mean_accuracies = []
    final_accuracies = []



    # for 20 different sample sizes
    for size in range(1 ,21):
        # Increase sample size by 1/20 and store it
        sample_size = (len(features)//20)*size
        sample_sizes.append(sample_size)

        # Evaluate the perceptor with different size and store metrics
        min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = e
        min_train_durations.append(min_train_duration)
        max_train_durations.append(max_train_duration)
        mean_train_durations.append(mean_train_duration)
        min_pred_durations.append(min_pred_duration)
        max_pred_durations.append(max_pred_duration)
        mean_pred_durations.append(mean_pred_duration)
        runtimes.append(runtime)
        min_accuracies.append(min_accuracy)
        max_accuracies.append(max_accuracy)
        mean_accuracies.append(mean_accuracy)
        final_accuracies.append(final_accuracy)

    # Display a lineplot of the runtimes for each sample sizes
    plt.figure()
    plt.title('Perceptron classification runtimes against sample sizes')
    plt.xlabel('Sample size')
    plt.ylabel('Runtime (seconds)')
    plt.plot(sample_sizes, runtimes,'o-')
    plt.plot(sample_sizes, mean_train_durations,'o-')
    plt.plot(sample_sizes, mean_pred_durations,'o-')
    plt.legend(["Whole runtime", "Mean Training duration",'Mean Prediction duration'], loc="upper left")
    plt.show()

    # Create a dataframe of the metrics against sample sizes
    perceptron_df = pd.DataFrame({'sample_sizes': sample_sizes,'min_train_durations': min_train_durations,'max_train_durations': max_train_durations,
                                  'mean_train_durations': mean_train_durations,'min_pred_durations': min_pred_durations,'max_pred_durations': max_pred_durations,
                                  'mean_pred_durations': mean_pred_durations,'runtimes': runtimes,'min_accuracies': min_accuracies,
                                  'max_accuracies': max_accuracies,'mean_accuracies': mean_accuracies,'final_accuracies': final_accuracies})

    # Output the datain a a CSV file
    perceptron_df.to_csv('Perceptron.csv', index=False)

########################################################################
```

Metrics table for sample sizes

Perceptron

sample_sizes	min_train_durations	max_train_durations	mean_train_durations	min_pred_durations	max_pred_durations	mean_pred_durations	runtimes	min_accuracies	max_accuracies	mean_accuracies	final_accuracies
900	0.030194	0.049573	0.0395528	0.00038	0.000825	0.0006784	0.074754	0.8680555555555556	0.9166666666666666	0.8916666666666666	0.9
1800	0.075325	0.104719	0.0838342	0.000568	0.000706	0.0006072	0.124177	0.8784722222222222	0.9236111111111112	0.8958333333333334	0.8944444444444445
2700	0.105285	0.148995	0.12393639999999999	0.000856	0.00115	0.0010145999999999998	0.140028	0.8726851851851852	0.9259259259259259	0.9032407407407408	0.9148148148148149
3600	0.147924	0.216507	0.1794366	0.001201	0.002241	0.0016778000000000001	0.244364	0.8993055555555556	0.9184027777777778	0.9059027777777777	0.8986111111111111
4500	0.137975	0.315963	0.2342632	0.001424	0.002494	0.0018924	0.317644	0.8833333333333333	0.9222222222222223	0.8983333333333332	0.9255555555555556
5400	0.237241	0.312943	0.2841728	0.001804	0.008331	0.0032032000000000007	0.423687	0.8622685185185185	0.9166666666666666	0.9011574074074075	0.9092592592592592
6300	0.28666	0.403707	0.33489840000000004	0.002072	0.013023	0.0052284	0.497003	0.9047619047619048	0.9206349206349206	0.9134920634920635	0.8928571428571429
7200	0.274545	0.425814	0.36036	0.002151	0.002731	0.0023848000000000003	0.379084	0.8923611111111112	0.9227430555555556	0.9107638888888889	0.9069444444444444
8100	0.354155	0.537192	0.4285862	0.002648	0.012828	0.006674200000000001	0.381582	0.8819444444444444	0.9282407407407407	0.9070987654320988	0.9265432098765433
9000	0.352503	0.487659	0.4173410000000001	0.002214	0.012427	0.0045934	0.611436	0.8833333333333333	0.9138888888888889	0.9001388888888888	0.9172222222222223
9900	0.420706	0.613039	0.47173999999999994	0.002712	0.007534	0.0045836	0.494736	0.8705808080808081	0.9248737373737373	0.9095959595959595	0.898989898989899
10800	0.402336	0.497716	0.46572899999999995	0.002047	0.00416	0.0026089999999999998	0.700214	0.8825231481481481	0.9120370370370371	0.8987268518518519	0.9166666666666666
11700	0.428234	0.730247	0.5989912	0.002526	0.012385	0.00654	0.488188	0.8846153846153846	0.9284188034188035	0.910576923076923	0.9222222222222223
12600	0.393409	0.522665	0.45773980000000003	0.004083	0.004924	0.0043062000000000005	0.46671	0.9077380952380952	0.9300595238095238	0.9194444444444445	0.926984126984127
13500	0.42317	0.682407	0.554613	0.004294	0.006566	0.0051296	0.824191	0.8981481481481481	0.9287037037037037	0.9137037037037038	0.9259259259259259
14400	0.560004	0.718195	0.6442368	0.004497	0.017648	0.009307	0.707944	0.90625	0.9292534722222222	0.917795138888889	0.9079861111111112
15300	0.629365	0.859144	0.6971546	0.005739	0.01891	0.012531599999999999	0.551949	0.9068627450980392	0.9268790849673203	0.917483660130719	0.9179738562091503
16200	0.463897	0.832133	0.6283152	0.005002	0.014779	0.0070704	0.589278	0.9085648148148148	0.9166666666666666	0.9125	0.9058641975308642
17100	0.54903	0.85991	0.7104001999999999	0.00512	0.014922	0.0072284	0.797623	0.8658625730994152	0.9269005847953217	0.9073099415204678	0.9160818713450293
18000	0.489053	0.837733	0.6690456000000001	0.005537	0.005852	0.005687	0.613013	0.9003472222222222	0.9319444444444445	0.9109027777777777	0.9247222222222222

Appendix 3 - Decision Tree

Definition:

```
229  ######################################################################
230  # Task4 - Decision trees
231  # Define a function to train and evaluate a Decision Tree classifer
232  def tree_classify(features,labels):
233      # Instantiate a Decision Tree
234      clf = tree.DecisionTreeClassifier()
235
236      # Make an evaluation of the Decision tree and display the mean accuracy
237      min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,
238      print("Decision Tree Mean Accuracy: ",mean_accuracy)
239
240      # Declare lists to store sample sizes and metrics
241      sample_sizes = []
242      min_train_durations = []
243      max_train_durations = []
244      mean_train_durations = []
245      min_pred_durations = []
246      max_pred_durations = []
247      mean_pred_durations = []
248      runtimes = []
249      min_accuracies = []
250      max_accuracies = []
251      mean_accuracies = []
252      final_accuracies = []
253
254
255      # for 20 different sample sizes
256      for size in range(1 ,21):
257          # Increase sample size by 1/20 and store it
258          sample_size = (len(features)//20)*size
259          sample_sizes.append(sample_size)
260          # Evaluate the Decision Tree with different size and store metrics
261          min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runt
262          min_train_durations.append(min_train_duration)
263          max_train_durations.append(max_train_duration)
264          mean_train_durations.append(mean_train_duration)
265          min_pred_durations.append(min_pred_duration)
266          max_pred_durations.append(max_pred_duration)
267          mean_pred_durations.append(mean_pred_duration)
268          runtimes.append(runtime)
269          min_accuracies.append(min_accuracy)
270          max_accuracies.append(max_accuracy)
271          mean_accuracies.append(mean_accuracy)
272          final_accuracies.append(final_accuracy)
273
274      # Display a lineplot of the runtimes for each sample sizes
275      plt.figure()
276      plt.title('Decision Tree classification runtimes against sample sizes')
277      plt.xlabel('Sample size')
278      plt.ylabel('Runtime (seconds)')
279      plt.plot(sample_sizes, runtimes,'o-')
280      plt.plot(sample_sizes, mean_train_durations,'o-')
281      plt.plot(sample_sizes, mean_pred_durations,'o-')
282      plt.legend(["Whole runtime", "Mean Training duration",'Mean Prediction duration'], loc="upper left")
283      plt.show()
284
285      # Create a dataframe of the metrics against sample sizes
286      decision_tree_df = pd.DataFrame({'sample_sizes': sample_sizes,'min_train_durations': min_train_durations,'max_train_durations': max_train_durations,
287                                       'mean_train_durations': mean_train_durations,'min_pred_durations': min_pred_durations,'max_pred_durations': max_pred_durations,
288                                       'mean_pred_durations': mean_pred_durations,'runtimes': runtimes,'min_accuracies': min_accuracies,
289                                       'max_accuracies': max_accuracies,'mean_accuracies': mean_accuracies,'final_accuracies': final_accuracies})
290
291      # Output the datain a a CSV file
292      decision_tree_df.to_csv('Decision Tree.csv', index=False)
293
```

Metrics table for sample sizes

Decision Tree

sample_sizes	min_train_durations	max_train_durations	mean_train_durations	min_pred_durations	max_pred_durations	mean_pred_durations	runtimes	min_accuracies	max_accuracies	mean_accuracies	final_accuracies
900	0.069358	0.09524	0.0802788	0.000223	0.000257	0.00023899999999999998	0.102846	0.7847222222222222	0.8472222222222222	0.826388888888889	0.7888888888888889
1800	0.163203	0.216731	0.1884722	0.000374	0.00042	0.00039880000000000004	0.252162	0.7916666666666666	0.8506944444444444	0.8270833333333332	0.8416666666666667
2700	0.298807	0.409448	0.34572220000000004	0.000648	0.000742	0.0007004000000000001	0.382518	0.8240740740740741	0.8587962962962963	0.8425925925925926	0.8611111111111112
3600	0.437589	0.501536	0.4566438	0.000869	0.000911	0.0008928	0.551154	0.8350694444444444	0.8888888888888888	0.8569444444444443	0.8722222222222222
4500	0.546591	0.678808	0.6018798000000001	0.001069	0.001205	0.0011137999999999999	0.836953	0.8472222222222222	0.8708333333333333	0.8613888888888889	0.8755555555555555
5400	0.663254	0.746254	0.7173904	0.001244	0.001356	0.0012889999999999998	0.881354	0.8622685185185185	0.876157407407074	0.8706018518518519	0.8833333333333333
6300	0.880885	0.966919	0.9129609999999999	0.001511	0.001579	0.0015456	1.060599	0.8521825396825397	0.8759920634920635	0.8660714285714286	0.8849206349206349
7200	0.914465	1.135474	1.046259	0.001574	0.001654	0.0015982	1.397833	0.8654513888888888	0.8854166666666666	0.875173611111111	0.8736111111111111
8100	1.19206	1.281243	1.2319144	0.001846	0.001915	0.0018854	1.703096	0.8734567901234568	0.8865740740740741	0.8799382716049383	0.8839506172839506
9000	1.347812	1.505547	1.4102374	0.001649	0.002029	0.0017408	1.950936	0.8659722222222223	0.8798611111111111	0.8743055555555556	0.8866666666666667
9900	1.491121	1.664253	1.5605562	0.001861	0.003139	0.002379	2.15851	0.8762626262626263	0.8876262626262627	0.8834595959595959	0.8898989898989899
10800	1.668545	1.906526	1.7758969999999998	0.002033	0.003381	0.0023506	2.54421	0.8686342592592593	0.8987268518518519	0.8832175925925926	0.8902777777777777
11700	1.830293	2.094124	1.9777321999999997	0.002266	0.003421	0.0025644	2.663572	0.8733974358974359	0.906517094017094	0.8867521367521368	0.8846153846153846
12600	2.007469	2.244528	2.1487098	0.004213	0.00431	0.0042562	2.751709	0.8715277777777778	0.8968253968253969	0.879265873015873	0.8845238095238095
13500	2.084396	2.484233	2.2792252	0.003233	0.003729	0.0034604	3.104312	0.8768518518518519	0.8958333333333334	0.8872222222222221	0.9018518518518519
14400	2.428737	2.681626	2.5382176000000003	0.003427	0.003816	0.0035782	3.121634	0.8810763888888888	0.89453125	0.8871527777777777	0.890625
15300	2.470487	2.772975	2.6713596	0.003723	0.004835	0.0041982	3.583477	0.8794934640522876	0.8986928104575164	0.8883169934640524	0.884640522875817
16200	2.804875	3.269212	3.0252026	0.003845	0.005085	0.0043556	3.899344	0.8796296296296297	0.8981481481481481	0.8877314814814815	0.894753086419753
17100	2.924567	3.237062	3.1042122	0.003424	0.003678	0.0035568	4.00539	0.8797514619883041	0.9016812865497076	0.8910087719298246	0.8964912280701754
18000	3.374966	3.823676	3.544511	0.003582	0.003846	0.003718	4.642643	0.8871527777777778	0.9048611111111111	0.8946527777777777	0.8902777777777777

Appendix 4 - K-nearest neighbours

Definition:

```
#########################################################################
# Task5 - k-nearest Neighbours
# Define a function to train and evaluate a KNN classifer
def knn_classify(features,labels):

    # Declare list for accuracies and parameter
    knn_accuracies =[]
    n_neighbors = []

    # for each parameter between 1 and 9
    for n_neighbor in range(1,10):
        # Store k parameter
        n_neighbors.append(n_neighbor)
        # train the classifier on this parameter
        clf = neighbors.KNeighborsClassifier(n_neighbors=n_neighbor, metric='minkowski')
        # Evaluate the classifier with the k parameter and store the accuracy
        min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = e
        knn_accuracies.append(mean_accuracy)

    # Display best accuracies
    print('best mean accuracy: ',max(knn_accuracies))

    # Select parameter with best accuracy
    best_k_param = n_neighbors[knn_accuracies.index(max(knn_accuracies))]
    print('best k parameter: ', best_k_param)

    # Declare lists to store sample sizes and metrics
    sample_sizes = []
    min_train_durations = []
    max_train_durations = []
    mean_train_durations = []
    min_pred_durations = []
    max_pred_durations = []
    mean_pred_durations = []
    runtimes = []
    min_accuracies = []
    max_accuracies = []
    mean_accuracies = []
    final_accuracies = []

    # Train the KNN classifier with best parameter
    clf = neighbors.KNeighborsClassifier(n_neighbors=best_k_param, metric='minkowski')

    # for 20 different sample sizes
    for size in range(1 ,21):
        # Increase sample size by 1/20 and store it
        sample_size = (len(features)//20)*size
        sample_sizes.append(sample_size)
        # Evaluate the Decision Tree with different size and store metrics
        min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = e
        min_train_durations.append(min_train_duration)
        max_train_durations.append(max_train_duration)
        mean_train_durations.append(mean_train_duration)
        min_pred_durations.append(min_pred_duration)
        max_pred_durations.append(max_pred_duration)
        mean_pred_durations.append(mean_pred_duration)
        runtimes.append(runtime)
        min_accuracies.append(min_accuracy)
        max_accuracies.append(max_accuracy)
        mean_accuracies.append(mean_accuracy)
        final_accuracies.append(final_accuracy)

    # Display a lineplot of the runtimes for each sample sizes
    plt.figure()
    plt.title('KNN classification runtimes against sample sizes')
    plt.xlabel('Sample size')
    plt.ylabel('Runtime (seconds)')
    plt.plot(sample_sizes, runtimes,'o-')
    plt.plot(sample_sizes, mean_train_durations,'o-')
    plt.plot(sample_sizes, mean_pred_durations,'o-')
    plt.legend(["Whole runtime", "Mean Training duration","Mean Prediction duration"], loc="upper left")
    plt.show()

    # Create a dataframe of the metrics against sample sizes
    knn_df = pd.DataFrame({'sample_sizes': sample_sizes,'min_train_durations': min_train_durations,'max_train_durations': max_train_durations,
                           'mean_train_durations': mean_train_durations,'min_pred_durations': min_pred_durations,'max_pred_durations': max_pred_durations,
                           'mean_pred_durations': mean_pred_durations,'runtimes': runtimes,'min_accuracies': min_accuracies,
                           'max_accuracies': max_accuracies,'mean_accuracies': mean_accuracies,'final_accuracies': final_accuracies})

    # Output the datain a a CSV file
    knn_df.to_csv('KNN.csv', index=False)
#########################################################################
```

Metrics table for sample sizes

KNN

sample_sizes	min_train_durations	max_train_durations	mean_train_durations	min_pred_durations	max_pred_durations	mean_pred_durations	runtimes	min_accuracies	max_accuracies	mean_accuracies	final_accuracies
900	0.000414	0.000591	0.0005261999999999999	0.005272	0.015617	0.0077188000000000005	0.012118	0.8125	0.9097222222222222	0.8722222222222221	0.8888888888888888
1800	0.000949	0.002	0.00141	0.014445	0.020952	0.0183534	0.022723	0.8645833333333334	0.9166666666666666	0.8951388888888889	0.9138888888888889
2700	0.00092	0.001358	0.0011303999999999997	0.028408	0.040253	0.034793	0.041183	0.8796296296296297	0.9097222222222222	0.8976851851851851	0.9018518518518519
3600	0.001852	0.003578	0.0027064	0.043026	0.062399	0.051096600000000006	0.070383	0.8871527777777778	0.9166666666666666	0.9041666666666666	0.8930555555555556
4500	0.002164	0.003349	0.0027957999999999998	0.06941	0.093749	0.07766700000000001	0.110577	0.9	0.9236111111111112	0.9113888888888889	0.9122222222222223
5400	0.002657	0.00486	0.003578799999999999	0.096106	0.109494	0.1023646	0.143157	0.9039351851851852	0.9189814814814815	0.9108796296296295	0.9296296296296296
6300	0.003114	0.009739	0.005799	0.119593	0.148032	0.13105499999999998	0.205896	0.8918650793650794	0.9236111111111112	0.9123015873015874	0.9238095238095239
7200	0.004236	0.009486	0.0060322	0.146152	0.18657	0.15955039999999998	0.230568	0.90625	0.9322916666666666	0.91875	0.9284722222222223
8100	0.004305	0.007211	0.0063486	0.182419	0.215155	0.1924308	0.288909	0.9120370370370371	0.9297839506172839	0.9199074074074074	0.924074074074074
9000	0.006403	0.009677	0.0075432	0.226603	0.27865	0.2500576	0.33954	0.9145833333333333	0.9298611111111111	0.9205555555555556	0.9316666666666666
9900	0.005142	0.009415	0.0076568	0.251712	0.274504	0.2634404	0.471406	0.9103535353535354	0.9286616161616161	0.9229797979797979	0.9262626262626262
10800	0.005486	0.010854	0.0079464	0.292623	0.306049	0.297976	0.463376	0.9212962962962963	0.9357638888888888	0.9268518518518519	0.9277777777777778
11700	0.006208	0.013077	0.0094838	0.331801	0.363729	0.35000859999999995	0.501438	0.9198717948717948	0.936965811965812	0.9255341880341881	0.9277777777777778
12600	0.006629	0.009168	0.0078256	0.385086	0.404903	0.3935324	0.586318	0.9285714285714286	0.9375	0.9313492063492064	0.9321428571428572
13500	0.006921	0.008376	0.0076495999999999995	0.437117	0.457775	0.4475478	0.662435	0.9189814814814815	0.9351851851851852	0.9297222222222222	0.9311111111111111
14400	0.007176	0.007384	0.0073076	0.476144	0.507295	0.4861904	0.721868	0.9214409722222222	0.9379340277777778	0.9310763888888889	0.93125
15300	0.007551	0.008932	0.0079556	0.537276	0.558295	0.5465014	0.798111	0.9215686274509803	0.9395424836601307	0.9332516339869281	0.9323529411764706
16200	0.008011	0.009226	0.0084108	0.607229	0.619988	0.6129756	0.900495	0.9266975308641975	0.9359567901234568	0.9318672839506172	0.9320987654320988
17100	0.008334	0.00972	0.008923200000000001	0.659075	0.685215	0.6692032	1.005314	0.9261695906432749	0.9367690058479532	0.9307748538011695	0.9333333333333333
18000	0.008726	0.010123	0.009136599999999998	0.717844	0.745395	0.7300616	1.093943	0.9284722222222223	0.9375	0.9333333333333333	0.9355555555555556

Appendix 5 - Support Vector Machine

Definition:

```
#########################################################################
# Task6 - Support Vector Machine
# Define a function to train and evaluate a SVM classifer
def svm_classify(features,labels):

    # Declare list for accuracies and parameter
    svm_accuracies =[]
    gammas = []

    # for each parameter between -8 and 0
    for gamma in range(-8,0):
        # Store gamma parameter
        gammas.append(gamma)
        # train the classifier on this parameter (10^-gamma)
        clf = svm.SVC(gamma=float(f'1e{gamma}'))
        # Evaluate the classifier with the gamma parameter and store the accuracy
        min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = evaluate(features,lab
        svm_accuracies.append(mean_accuracy)

    # Display best accuracy
    print('best mean accuracy: ',max(svm_accuracies))

    # Select parameter with best accuracy
    best_gamma_param = gammas[svm_accuracies.index(max(svm_accuracies))]
    print('best gamma parameter: ', best_gamma_param)

    # Declare lists to store sample sizes and metrics
    sample_sizes = []
    min_train_durations = []
    max_train_durations = []
    mean_train_durations = []
    min_pred_durations = []
    max_pred_durations = []
    mean_pred_durations = []
    runtimes = []
    min_accuracies = []
    max_accuracies = []
    mean_accuracies = []
    final_accuracies = []

    # Train the SVM classifier with best parameter
    clf = svm.SVC(gamma=float(f'1e{best_gamma_param}'))

    # for 20 different sample sizes
    for size in range(1 ,21):
        # Increase sample size by 1/20 and store it
        sample_size = (len(features)//20)*size
        sample_sizes.append(sample_size)
        # Evaluate the Decision Tree with different size and store metrics
        min_train_duration,max_train_duration,mean_train_duration,min_pred_duration,max_pred_duration,mean_pred_duration,min_accuracy,max_accuracy,mean_accuracy,runtime,final_accuracy = evaluate(features,lab
        min_train_durations.append(min_train_duration)
        max_train_durations.append(max_train_duration)
        mean_train_durations.append(mean_train_duration)
        min_pred_durations.append(min_pred_duration)
        max_pred_durations.append(max_pred_duration)
        mean_pred_durations.append(mean_pred_duration)
        runtimes.append(runtime)
        min_accuracies.append(min_accuracy)
        max_accuracies.append(max_accuracy)
        mean_accuracies.append(mean_accuracy)
        final_accuracies.append(final_accuracy)

    # Display a lineplot of the runtimes for each sample sizes
    plt.figure()
    plt.title('SVM classification runtimes against sample sizes')
    plt.xlabel('Sample size')
    plt.ylabel('Runtime (seconds)')
    plt.plot(sample_sizes, runtimes,'o-')
    plt.plot(sample_sizes, mean_train_durations,'o-')
    plt.plot(sample_sizes, mean_pred_durations,'o-')
    plt.legend(["Whole runtime", "Mean Training duration",'Mean Prediction duration'], loc="upper left")
    plt.show()

    # Create a dataframe of the metrics against sample sizes
    svm_df = pd.DataFrame({'sample_sizes': sample_sizes,'min_train_durations': min_train_durations,'max_train_durations': max_train_durations,
                           'mean_train_durations': mean_train_durations,'min_pred_durations': min_pred_durations,'max_pred_durations': max_pred_durations,
                           'mean_pred_durations': mean_pred_durations,'runtimes': runtimes,'min_accuracies': min_accuracies,
                           'max_accuracies': max_accuracies,'mean_accuracies': mean_accuracies,'final_accuracies': final_accuracies})


    # Output the datain a a CSV file
    svm_df.to_csv('SVM.csv', index=False)

    print('best mean accuracy: ',max(svm_accuracies))

    best_gamma_param = gammas[svm_accuracies.index(max(svm_accuracies))]
    print('best gamma parameter: ', best_gamma_param)

#########################################################################
```

Metrics table for sample sizes

SVM

sample_sizes	min_train_durations	max_train_durations	mean_train_durations	min_pred_durations	max_pred_durations	mean_pred_durations	runtimes	min_accuracies	max_accuracies	mean_accuracies	final_accuracies
900	0.065146	0.066705	0.0660866	0.017631	0.019819	0.0186104	0.116379	0.8888888888888888	0.9513888888888888	0.9083333333333334	0.9277777777777778
1800	0.213643	0.217827	0.2162322	0.060335	0.061527	0.060848	0.412303	0.8923611111111112	0.9305555555555556	0.9166666666666666	0.9388888888888889
2700	0.444639	0.448654	0.44630080000000005	0.124725	0.127511	0.1257882	0.795948	0.9189814814814815	0.9444444444444444	0.9356481481481481	0.924074074074074
3600	0.674064	0.691029	0.6785058	0.205635	0.212045	0.2088912	1.217929	0.9184027777777778	0.9444444444444444	0.935763888888889	0.9333333333333333
4500	0.895294	1.019414	0.9471073999999999	0.310728	0.325627	0.31683760000000005	1.884876	0.9194444444444444	0.9458333333333333	0.9341666666666667	0.9511111111111111
5400	1.372481	1.562032	1.4421186000000001	0.426976	0.461326	0.4430998	2.983743	0.9305555555555556	0.9525462962962963	0.9398148148148149	0.9481481481481482
6300	1.845983	2.008928	1.8954818	0.564932	0.577473	0.5700736	4.118045	0.9375	0.9490476190476190	0.9430555555555555	0.9515873015873015
7200	2.507017	2.634997	2.5779729999999996	0.698378	0.737281	0.7192504	5.803911	0.9305555555555556	0.9539930555555556	0.9440972222222221	0.9527777777777777
8100	3.278667	3.641872	3.5649068	0.870558	0.97159	0.898606	7.034245	0.941358024691358	0.9537037037037037	0.9466049382716049	0.9580246913580247
9000	4.629031	5.033818	4.8093426	1.030181	1.055713	1.0457034	8.811339	0.9451388888888889	0.95625	0.9512500000000002	0.945
9900	5.43188	5.802549	5.6364876	1.23297	1.286576	1.257108	10.162311	0.9431818181818182	0.9539141414141414	0.9502525252525252	0.9580808080808081
10800	6.297552	6.95035	6.5362064	1.417563	1.474961	1.4379598000000002	12.863589	0.9415509259259259	0.9612268518518519	0.9525462962962962	0.9546296296296296
11700	7.866185	8.135962	8.0192034	1.635848	1.688185	1.6640576	15.35612	0.9471153846153846	0.9588675213675214	0.9532051282051281	0.9547008547008548
12600	8.518399	9.387481	8.813559600000001	1.86531	1.896009	1.8834098000000001	16.583552	0.9533730158730159	0.9588293650793651	0.9561507936507937	0.9551587301587302
13500	9.694315	10.139516	9.960016399999999	2.075523	2.116775	2.0938262	18.374352	0.9527777777777777	0.961574074074074	0.9569444444444445	0.9566666666666667
14400	10.947475	11.178228	11.076003199999999	2.308339	2.343536	2.3293926	20.676133	0.9505208333333334	0.9631076388888888	0.9555555555555555	0.9576388888888889
15300	12.367669	12.491454	12.4381974	2.541723	2.585749	2.5624279999999997	23.136135	0.954248366013072	0.9697712418300654	0.9584967320261437	0.9571895424836602
16200	13.89875	16.380718	14.442776400000003	2.809255	2.997376	2.8945008000000003	28.720033	0.9533179012345679	0.9610339506172839	0.9571759259259259	0.9595679012345679
17100	16.443949	19.726948	17.622879	3.085447	4.469595	3.4212207999999995	32.354439	0.952485380116959	0.9612573099415205	0.9571637426900585	0.9605263157894737
18000	17.141611	20.71087	18.5585454	3.361053	3.463956	3.420625	34.149211	0.9548611111111112	0.9628472222222222	0.9576388888888889	0.96

Table of content