Machine Learning Course Project

Course Code: ICS3206-SEM1-A-1920

Date: 06/01/2020

Student: Valerija Holomjova

Table of Contents

Introduction

Overview and Instructions of Code

The MNIST dataset was used and the training and testing '.csv' files can be found in the 'data' directory. To execute the code simply run the 'runner.py' file found in the root code directory. It should be noted that the following libraries are required to run the file:

- numpy
- pandas
- matplotlib
- scikit-learn

When running the file, the main 'runner()' function will be called. This function will call 'buildFinalModel()' to build and train an SVM model using the optimal hyperparameters that were deduced through experimentation carried out in the report, and then evaluate the model's performance using the final test set. It should be noted that since the dataset is very large, it will take around 15 minutes for the function to fully train the model. There are also two sections of the main function that have been uncommented - the 'testAllModels()' function (hyperparameter tuning) and the 'testSingleModel()' function used to test a single model using the test and validation set. The hyperparameter tuning function builds around 60 candidate models and uses k-cross validation of k=3, hence, it has to train around 120 models and will take a long time to execute (around 20 hours) and was commented out for this reason. However, it can be tested out using a much smaller dataset by uncommenting a section of the function which is specified in the code overview section. All functions are described in the report as well as all the experiments that were carried out.

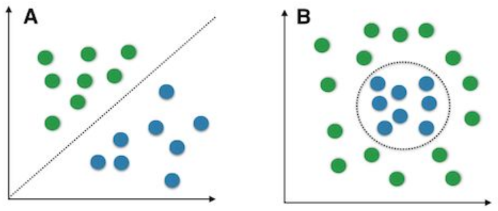
Statement of Completion

All sections of the assignment were completed and described in the report.

Item	**Completed (Yes/No/Partial)**
SVM technical discussion	Yes
A good comparison to alternative methods	Yes
Artifact	Yes
Experimentation with different SVM params	Yes
Experiments and their evaluation	Yes
Overall conclusions	Yes

Discussion of Support Vector Machines

Support vector machines (SVM) are supervised learning models used for both classification and regression tasks. The goal of the SVM is to determine an optimal hyperplane that can classify all the training vectors into their distinct classes. In simpler terms, it aims to segregate a dataset in the most optimal manner. It should be noted that the support vectors are the data points closest to the hyperplane. The best hyperplane is considered to be the one which leaves the maximum margin from all classes (from the hyperplane to support vectors). The figure on the left below illustrates an example of this where a hyperplane (the dotted line) separates two classes of data points at maximum margin (as far away as possible).



In certain cases, it is not possible to simple divide data in a linear way, such as the figure on the right above. As a result, kernel functions are applied of each data instance to transform them from non-linear observations into higher-dimensional linearly separable ones. Some popular kernels are mentioned in the points below. The regularization parameter (C) is the degree of importance given to miss-classifications. For instance, larger C values causes the classifier to search for large-margin hyperplanes even if it ends up misclassifying more points. On the other hand, smaller C values causes the classifier to choose smaller margin hyperplanes provided it does better classification. The gamma (γ) parameter determines the distance of which points are considered in the calculation of the hyperplane. Higher gamma values ensures nearby points are considered whilst lower gamma values ensures further points are also considered.

- Linear Kernel - This is the simplest kernel function and is the dot product of any two given observations (or vectors).

  $$k(x_1,\ x_2)\ = x_1 \cdot\ x_2$$
- Polynomial Kernel - This kernel function has a similar form to the linear kernel and can distinguish nonlinear input space. It should be noted that **d** is the degree of the polynomial.

  $$k(x_1,\ x_2)\ = (\gamma\ x_1 \cdot\ x_2\ )^d$$
- Radial Basis Function (RBF) Kernel - The value of this function depends on the distance from the origin or from some point. It can be noted that it contains the euclidean distance between x_1 and x_2 . It should be noted that as the value of c and gamma increases, the model overfits and vice versa.

  $$k(x_1,\ x_2)\ = (-\gamma\ ||x_1 -\ x_2||^2)$$

Support vector machines are preferred for classification since they are able to produce high accuracy results with less computation power. Since the hyperplane is calculated using support vectors, outliers are less likely to negatively impact results. It should be noted that support vector machines perform well with clear margins of separation and are really effective in high dimensional spaces.

On the other hand, large datasets are not suitable for this type of model and may result in a very high training time. Moreover, support vector machines perform poorly in cases of overlapped classes and are very sensitive to kernels, which results in the need to select appropriate hyperparameters for good performance.

Comparison to other Methods

Other alternative supervised learning models include decision trees, k-nearest neighbours, Naive Bayes, neural networks and more. Decision trees are represented in a tree structure form and can be used to tackle classification or regression tasks. The tree structure is made up of leaf nodes and internal nodes. Internal nodes represent a test on an attribute (eg. Outlook) and can have two or more branches (eg. Windy, Sunny, Rainy). Leaf nodes represent an outcome of a test or a classification. Hence, each leaf node represents the final outcome of a decision path. As a result, decision trees are able to create understandable rules and highlight which attributes are most important for prediction. Although classification doesn’t require much computation, adding more nodes to a decision tree can be computationally expensive which limits them from learning complicated rules and using large data sets. Hence, decision trees are more interpretable but SVM’s may be more accurate and are able to scale to larger data sets.

K-nearest neighbours (KNN) is a supervised machine learning algorithm which aims to assign a test data instance to a class that is most common among its k nearest neighbours. It is able to classify linear and non-linear distributed data automatically, whereas SVM’s require kernels to transform non-linear data. The performance of KNN is improved by selecting optimal k-values and distance metrics, while SVM’s require optimal selection of kernels and parameters. KNN is easy to implement and less computationally intensive, however it requires good feature selection as it is sensitive to bad attributes. Moreover, it is sensitive to outliers than an SVM, hence the outliers would have to be removed for optimal results.

Naive Bayes classifier is a probabilistic machine learning model based on Bayes Theorem and tries to predict the most likely class of a problem instance for a given set of features. It is naive as a result of the assumption that features are independent and their presence does not affect each other. On the other hand, the SVM considers interactions between features when using non-linear kernels. Hence, Naive Bayes classifier may perform better in cases where features are independent which may not be very common. Both models are sensitive to parameter optimization which significantly impacts their output.

Artificial Neural Networks (ANN) are computational networks inspired by the human brain, that uses different layers of mathematical processing which are connected to each other. Training is carried out by feeding in large amounts of training set and predicting a classifier, if the prediction is wrong, back propagation is used to adjust its learning. The ANN attempts to find a hyperplane by minimizing a loss function through back-propagation. It should be noted that the hyperplane is different from the SVM as it attempts to correctly classify points rather than finding the maximum margin width. ANN's are also able to learn and model complex and non-linear relationships. Similarly to SVM, ANN's require experimentation in order to determine the appropriate network structure for good performance. Unfortunately, ANN's have the possibility to suffer from multiple local minima while SVM's provide a unique solution. ANN's are also more likely to overfit compared to SVM's.

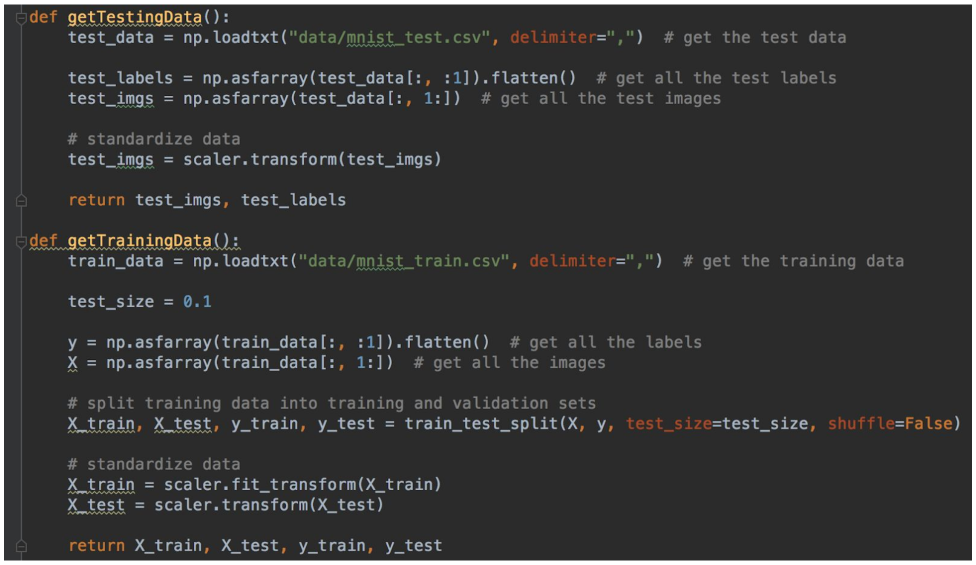
Code Overview

The following section will proceed by describing the functions that were implemented to train, tune and evaluate the SVM model. It will also describe the functions where the test and training data was parsed and split accordingly. It should be noted that the following tutorials[1][2][3] were used to understand how to use the 'scikit-learn' library functions as well the 'GridSearchCV' function for hyperparameter tuning.

Loading and Preparing the Dataset

For the assignment the MNIST dataset was used and the .csv files were obtained from the following link[4]. The 'getTrainingData()' function retrieves images from the 'mnist_train.csv' file then splits it into a training and validation set according to the variable 'test_size'. This variable is altered during experimentation to determine the best size for the best accuracy of the model. These results can be found in the 'Experimentation and Evaluation' section of the report. The split data is then standardized and returned.

The 'getTestingData()' function retrieves the images and labels from the 'mnist_test.csv' file then standardizes it and returns it. This function is used to retrieve test images and labels for the final evaluation of the SVM model.



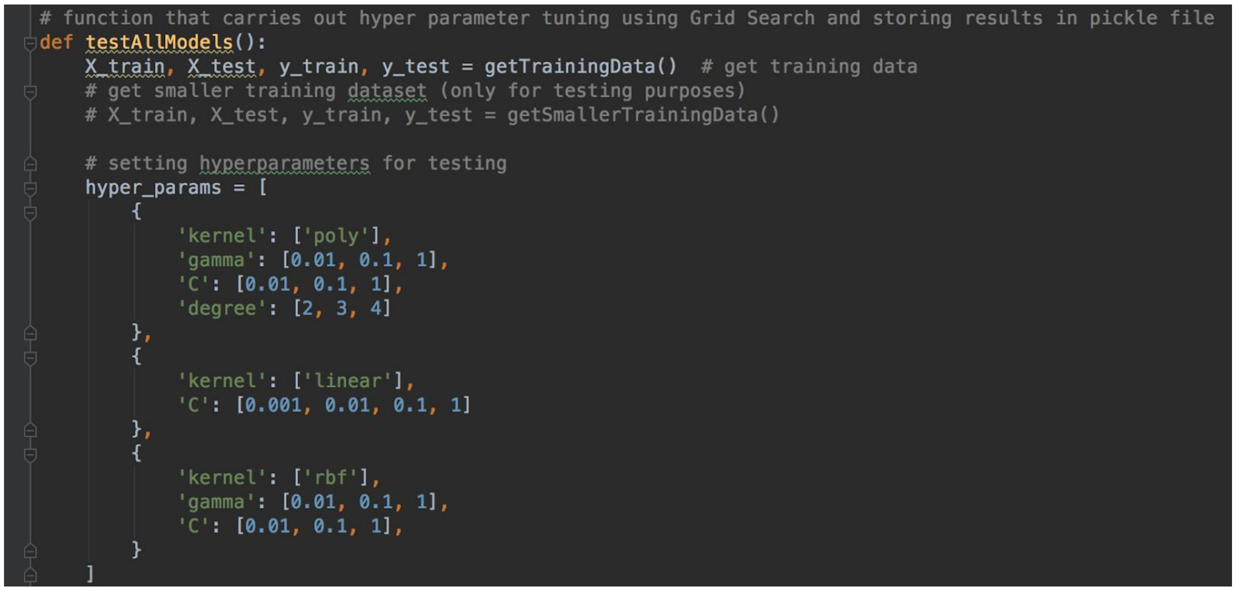
```
def getTestingData():
    test_data = np.loadtxt("data/mnist_test.csv", delimiter=",")  # get the test data

    test_labels = np.asfarray(test_data[:, :1]).flatten()  # get all the test labels
    test_imgs = np.asfarray(test_data[:, 1:])  # get all the test images

    # standardize data
    test_imgs = scaler.transform(test_imgs)

    return test_imgs, test_labels

def getTrainingData():
    train_data = np.loadtxt("data/mnist_train.csv", delimiter=",")  # get the training data

    test_size = 0.1

    y = np.asfarray(train_data[:, :1]).flatten()  # get all the labels
    X = np.asfarray(train_data[:, 1:])  # get all the images

    # split training data into training and validation sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, shuffle=False)

    # standardize data
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    return X_train, X_test, y_train, y_test
```

Figure 1.1 - Functions to retrieve the MNIST dataset and split it into training, testing and validation sets

[1] https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html
[2] https://www.kaggle.com/nishan192/mnist-digit-recognition-using-svm/versions
[3] https://www.kaggle.com/soumya044/mnist-digit-recognizer-using-kernel-svm
[4] https://www.python-course.eu/neural_network_mnist.php

Hyperparameter Tuning

The following function is called to test out various chosen hyperparameters for different SVM models using the training and validation sets and then returns the parameters of the models that achieved the highest accuracy. The hyperparameters can be adjusted accordingly in the 'hyper_params' variable. Figure 1.2 illustrates the hyperparameters that were tested out in the experimentation and evaluation section. It should be noted that this function is commented out in the main 'runner()' function as it takes a very long time to finish testing all the models (at least 20 hours). In addition to this, 'getSmallerTrainingData()' was included (and commented out) solely for testing purposes which uses the sklearn digits dataset instead of MNIST dataset for an example of how the function would work in a small amount of time.

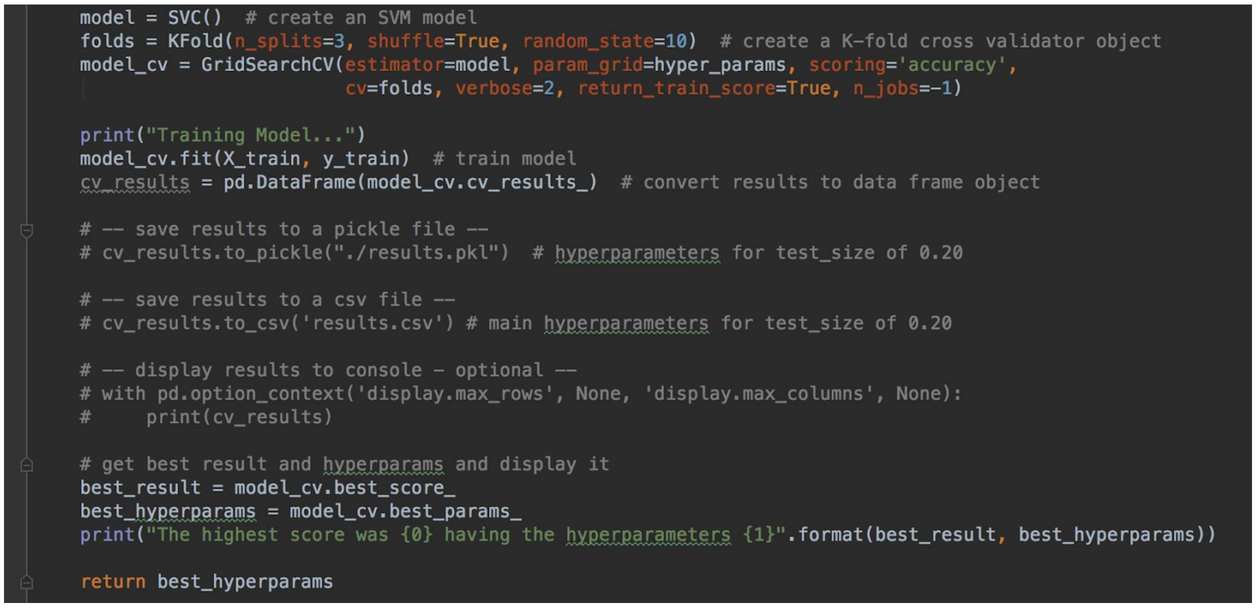


```
# function that carries out hyper parameter tuning using Grid Search and storing results in pickle file
def testAllModels():
    X_train, X_test, y_train, y_test = getTrainingData()  # get training data
    # get smaller training dataset (only for testing purposes)
    # X_train, X_test, y_train, y_test = getSmallerTrainingData()

    # setting hyperparameters for testing
    hyper_params = [
        {
            'kernel': ['poly'],
            'gamma': [0.01, 0.1, 1],
            'C': [0.01, 0.1, 1],
            'degree': [2, 3, 4]
        },
        {
            'kernel': ['linear'],
            'C': [0.001, 0.01, 0.1, 1]
        },
        {
            'kernel': ['rbf'],
            'gamma': [0.01, 0.1, 1],
            'C': [0.01, 0.1, 1],
        }
    ]
```

Figure 1.2 - Code for retrieving the training data and modifying hyperparameters for hyperparameter optimization

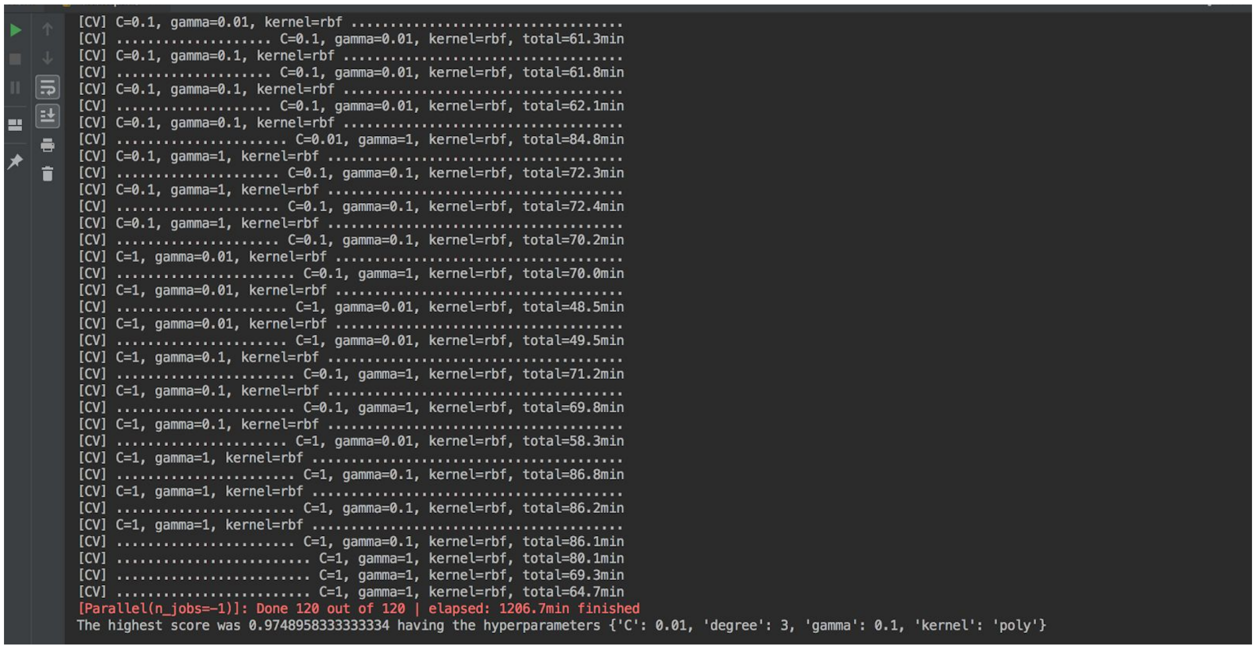
Each model is built using a k-fold cross validation with a k-value of 3. It should be noted that a higher k-value was preferred but a value of 3 was chosen to shorten the time taken to execute the function. Hyperparameter tuning is carried out using the 'GridSearchCV' function. After each model is trained and accuracy is obtained by the function, the results are converted into a data frame object and store in a .csv and pickle file called 'results.csv' and 'results.pkl' respectively.

Next, the model with the highest accuracy and best hyperparameters is determined and displayed to the console. The hyperparameters are then returned from the function to be used for evaluating the final model using the test set. Figure 1.4 illustrates an example of the console output displayed by the 'GridSearchCV' function while training each model and the final determination of the best performing model.



```python
    model = SVC()  # create an SVM model
    folds = KFold(n_splits=3, shuffle=True, random_state=10)  # create a K-fold cross validator object
    model_cv = GridSearchCV(estimator=model, param_grid=hyper_params, scoring='accuracy',
                            cv=folds, verbose=2, return_train_score=True, n_jobs=-1)

    print("Training Model...")
    model_cv.fit(X_train, y_train)  # train model
    cv_results = pd.DataFrame(model_cv.cv_results_)  # convert results to data frame object

    # -- save results to a pickle file --
    # cv_results.to_pickle("./results.pkl")  # hyperparameters for test_size of 0.20

    # -- save results to a csv file --
    # cv_results.to_csv('results.csv') # main hyperparameters for test_size of 0.20

    # -- display results to console - optional --
    # with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    #     print(cv_results)

    # get best result and hyperparams and display it
    best_result = model_cv.best_score_
    best_hyperparams = model_cv.best_params_
    print("The highest score was {0} having the hyperparameters {1}".format(best_result, best_hyperparams))

    return best_hyperparams
```

Figure 1.3 - Code for building the SVM models with various hyperparameters then testing them on the validation set to retrieve the best performing model

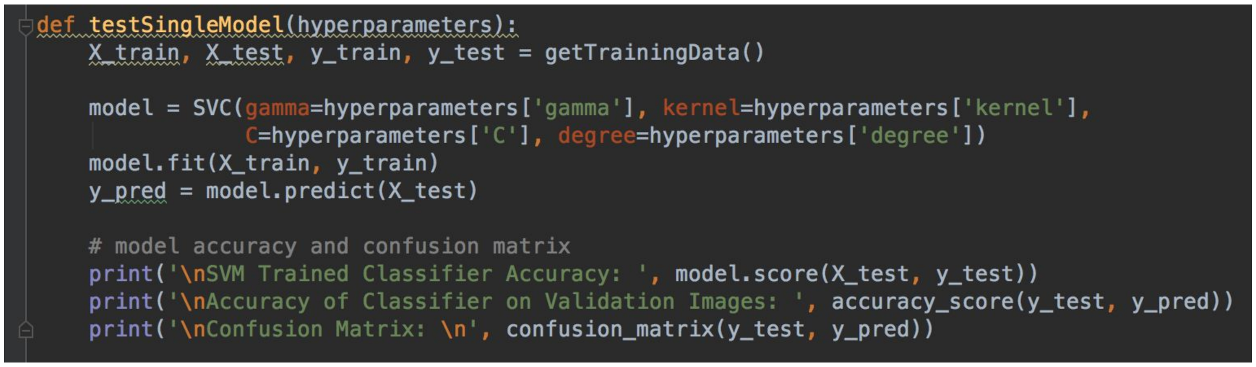


```
[CV] C=0.1, gamma=0.01, kernel=rbf ....................................
[CV] ..................... C=0.1, gamma=0.01, kernel=rbf, total=61.3min
[CV] C=0.1, gamma=0.1, kernel=rbf .....................................
[CV] ..................... C=0.1, gamma=0.01, kernel=rbf, total=61.8min
[CV] C=0.1, gamma=0.1, kernel=rbf .....................................
[CV] ..................... C=0.1, gamma=0.01, kernel=rbf, total=62.1min
[CV] C=0.1, gamma=0.1, kernel=rbf .....................................
[CV] ...................... C=0.01, gamma=1, kernel=rbf, total=84.8min
[CV] C=0.1, gamma=1, kernel=rbf .......................................
[CV] ...................... C=0.1, gamma=0.1, kernel=rbf, total=72.3min
[CV] C=0.1, gamma=1, kernel=rbf .......................................
[CV] ...................... C=0.1, gamma=0.1, kernel=rbf, total=72.4min
[CV] C=0.1, gamma=1, kernel=rbf .......................................
[CV] ...................... C=0.1, gamma=0.1, kernel=rbf, total=70.2min
[CV] C=1, gamma=0.01, kernel=rbf ......................................
[CV] ........................ C=0.1, gamma=1, kernel=rbf, total=70.0min
[CV] C=1, gamma=0.01, kernel=rbf ......................................
[CV] ....................... C=1, gamma=0.01, kernel=rbf, total=48.5min
[CV] C=1, gamma=0.01, kernel=rbf ......................................
[CV] ....................... C=1, gamma=0.01, kernel=rbf, total=49.5min
[CV] C=1, gamma=0.1, kernel=rbf .......................................
[CV] ........................ C=0.1, gamma=1, kernel=rbf, total=71.2min
[CV] C=1, gamma=0.1, kernel=rbf .......................................
[CV] ........................ C=0.1, gamma=1, kernel=rbf, total=69.8min
[CV] C=1, gamma=0.1, kernel=rbf .......................................
[CV] ....................... C=1, gamma=0.01, kernel=rbf, total=58.3min
[CV] C=1, gamma=1, kernel=rbf .........................................
[CV] ........................ C=1, gamma=0.1, kernel=rbf, total=86.8min
[CV] C=1, gamma=1, kernel=rbf .........................................
[CV] ........................ C=1, gamma=0.1, kernel=rbf, total=86.2min
[CV] C=1, gamma=1, kernel=rbf .........................................
[CV] ........................ C=1, gamma=0.1, kernel=rbf, total=86.1min
[CV] .......................... C=1, gamma=1, kernel=rbf, total=80.1min
[CV] .......................... C=1, gamma=1, kernel=rbf, total=69.3min
[CV] .......................... C=1, gamma=1, kernel=rbf, total=64.7min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 1206.7min finished
The highest score was 0.9748958333333334 having the hyperparameters {'C': 0.01, 'degree': 3, 'gamma': 0.1, 'kernel': 'poly'}
```

Figure 1.4 - A sample of the console output from the 'testAllModels()' function

Testing Single Models

The 'testSingleModel()' function trains and evaluates the performance of a single SVM model using the training and validation set for a given set of hyperparameters. This was mainly used when testing out various values for the 'test_size' variable on the model with optimal hyperparameters since the 'testAllModels()' function would take a long time to fully execute with each model. It should be noted that the following function also displays the accuracy and confusion matrix of the performance of the single model in the console.

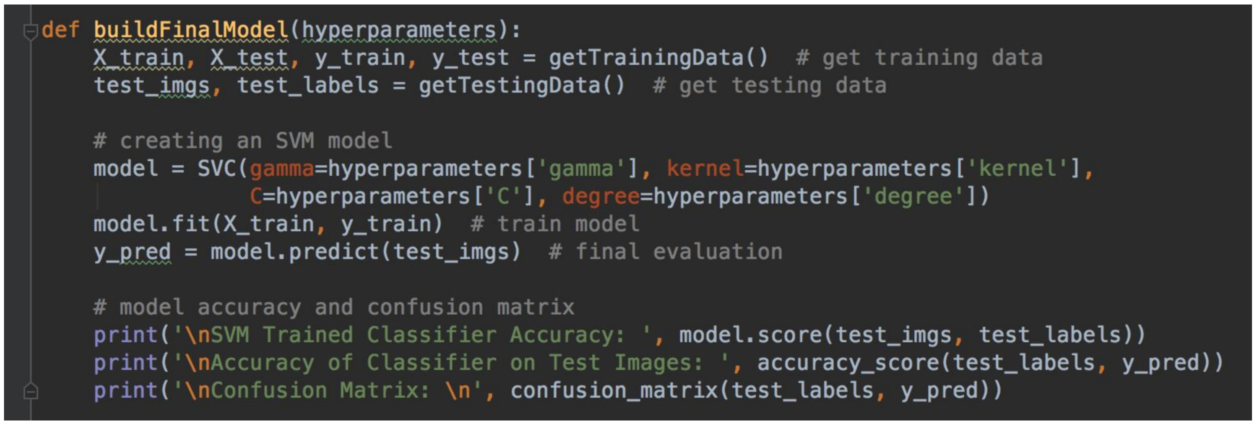


```python
def testSingleModel(hyperparameters):
    X_train, X_test, y_train, y_test = getTrainingData()

    model = SVC(gamma=hyperparameters['gamma'], kernel=hyperparameters['kernel'],
                C=hyperparameters['C'], degree=hyperparameters['degree'])
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # model accuracy and confusion matrix
    print('\nSVM Trained Classifier Accuracy: ', model.score(X_test, y_test))
    print('\nAccuracy of Classifier on Validation Images: ', accuracy_score(y_test, y_pred))
    print('\nConfusion Matrix: \n', confusion_matrix(y_test, y_pred))
```

Figure 1.5 - Code for building and training a single SVM model with given parameters for evaluation and displaying results to the console

Building the Final Model

The 'buildFinalModel()' function trains and evaluates the performance of a single SVM model using the training and final test set for a given set of hyperparameters. The best chosen hyperparameters that were determined from the 'testAllModels()' function are passed into this function for the final evaluation. After training and evaluation, the accuracy of the model is displayed to the console as well as the confusion matrix.



```python
def buildFinalModel(hyperparameters):
    X_train, X_test, y_train, y_test = getTrainingData()  # get training data
    test_imgs, test_labels = getTestingData()  # get testing data

    # creating an SVM model
    model = SVC(gamma=hyperparameters['gamma'], kernel=hyperparameters['kernel'],
                C=hyperparameters['C'], degree=hyperparameters['degree'])
    model.fit(X_train, y_train)  # train model
    y_pred = model.predict(test_imgs)  # final evaluation

    # model accuracy and confusion matrix
    print('\nSVM Trained Classifier Accuracy: ', model.score(test_imgs, test_labels))
    print('\nAccuracy of Classifier on Test Images: ', accuracy_score(test_labels, y_pred))
    print('\nConfusion Matrix: \n', confusion_matrix(test_labels, y_pred))
```

Figure 1.5 - Code for building and training the final SVM model with given parameters for the final evaluation using the testing set and displaying results to the console

Experimentation and Evaluation

Hyperparameter Tuning

The following section of the report analyzes the results obtained during hyperparameter optimization after running the function ‘testAllModels()’ for various hyperparameters. A k-fold cross validation with a k-value of 3 was used for evaluation. Although a higher k-value was preferred, a lower k-value was chosen for a reduced fitting time. Please refer to the ‘Code Overview’ section of the report for an explanation of the function. This section of the report will proceed by discussing the accuracy score obtained by testing different hyperparameters for various kernels. For an explanation of the kernels please refer to the ‘Discussion of Support Vector Machines’ section in the report. It should be noted that the ‘testAllModels()’ function deduced that the highest score was obtained by the SVM model using a polynomial kernel of degree 3 and a C and gamma value of 0.01 and 0.1 respectively.

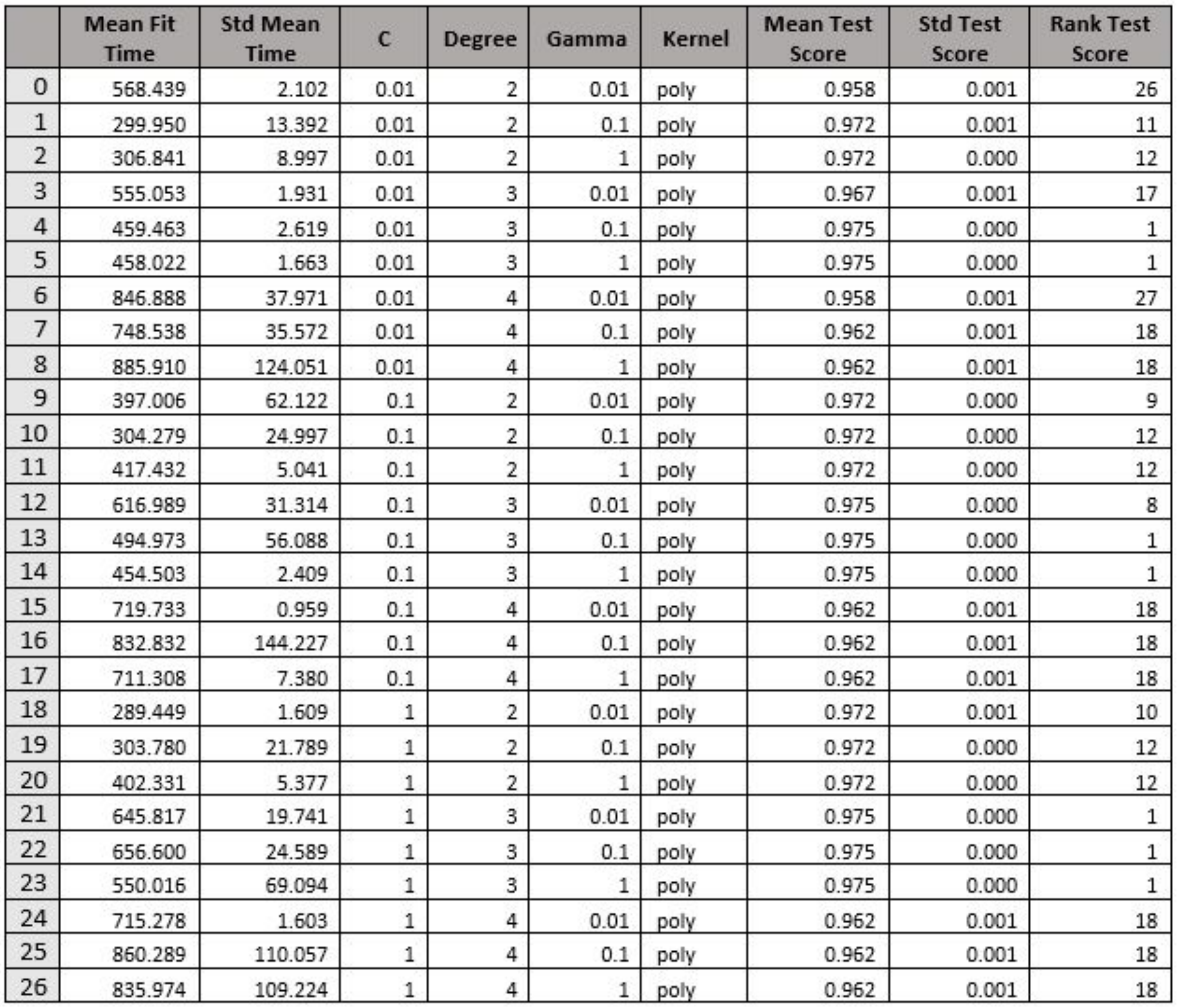
Polynomial Kernel

The polynomial kernel variation of the SVM was tested using the following parameters:

- Gamma: [0.01, 0.1, 1]
- Degree: [2, 3, 4]
- C: [0.01, 0.1, 1]

Please refer to Figure 2.1 for an overview of the performance of each candidate model. It illustrates the model fitting time, accuracy score and ranked score (compared to the score of all various kernel variations) of each model. Figures 2.2 - 2.4 illustrate the change in accuracy scores for the various hyperparameters. According to Figure 2.3, it seems as though the accuracy score does not go higher than 0.974 with different values of C and gamma however this can only be confirmed with further exploration of these values. Figure 2.2 suggest that the accuracy score of polynomial kernel models of degree 2 having C values of 0.1 and 1 could increase slightly if given lower gamma values.

By observation, the polynomial kernel models having a degree of 3 obtained the highest accuracy score of 0.974. It appears that the C and gamma values did not have a large impact on the accuracy score of the models with a degree of 3, as they remained relatively the same (around 0.974). The polynomial kernel models having a degree of 4 obtain the second highest accuracy of 0.972, followed by the models having a degree of 2 with an accuracy of 0.962. Considering there are a total of 32 variations that were experimented on (including the other kernel variations), it seems as though the polynomial kernel models achieved the highest ranked scores and best results. It should also be noted that in comparison to the other two kernels that were tested, the polynomial kernel SVM model had a median fitting time.



	Mean Fit Time	Std Mean Time	C	Degree	Gamma	Kernel	Mean Test Score	Std Test Score	Rank Test Score
0	568.439	2.102	0.01	2	0.01	poly	0.958	0.001	26
1	299.950	13.392	0.01	2	0.1	poly	0.972	0.001	11
2	306.841	8.997	0.01	2	1	poly	0.972	0.000	12
3	555.053	1.931	0.01	3	0.01	poly	0.967	0.001	17
4	459.463	2.619	0.01	3	0.1	poly	0.975	0.000	1
5	458.022	1.663	0.01	3	1	poly	0.975	0.000	1
6	846.888	37.971	0.01	4	0.01	poly	0.958	0.001	27
7	748.538	35.572	0.01	4	0.1	poly	0.962	0.001	18
8	885.910	124.051	0.01	4	1	poly	0.962	0.001	18
9	397.006	62.122	0.1	2	0.01	poly	0.972	0.000	9
10	304.279	24.997	0.1	2	0.1	poly	0.972	0.000	12
11	417.432	5.041	0.1	2	1	poly	0.972	0.000	12
12	616.989	31.314	0.1	3	0.01	poly	0.975	0.000	8
13	494.973	56.088	0.1	3	0.1	poly	0.975	0.000	1
14	454.503	2.409	0.1	3	1	poly	0.975	0.000	1
15	719.733	0.959	0.1	4	0.01	poly	0.962	0.001	18
16	832.832	144.227	0.1	4	0.1	poly	0.962	0.001	18
17	711.308	7.380	0.1	4	1	poly	0.962	0.001	18
18	289.449	1.609	1	2	0.01	poly	0.972	0.001	10
19	303.780	21.789	1	2	0.1	poly	0.972	0.000	12
20	402.331	5.377	1	2	1	poly	0.972	0.000	12
21	645.817	19.741	1	3	0.01	poly	0.975	0.000	1
22	656.600	24.589	1	3	0.1	poly	0.975	0.000	1
23	550.016	69.094	1	3	1	poly	0.975	0.000	1
24	715.278	1.603	1	4	0.01	poly	0.962	0.001	18
25	860.289	110.057	1	4	0.1	poly	0.962	0.001	18
26	835.974	109.224	1	4	1	poly	0.962	0.001	18

Figure 2.1 - Results obtained by hyperparameter tuning using a Polynomial Kernel

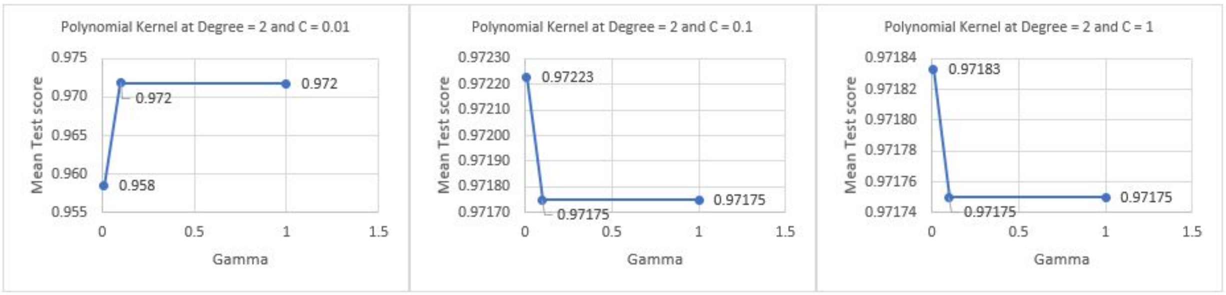


Figure 2.2 - Mean Test Score of a polynomial kernel of degree 2 for varying C and Gamma values

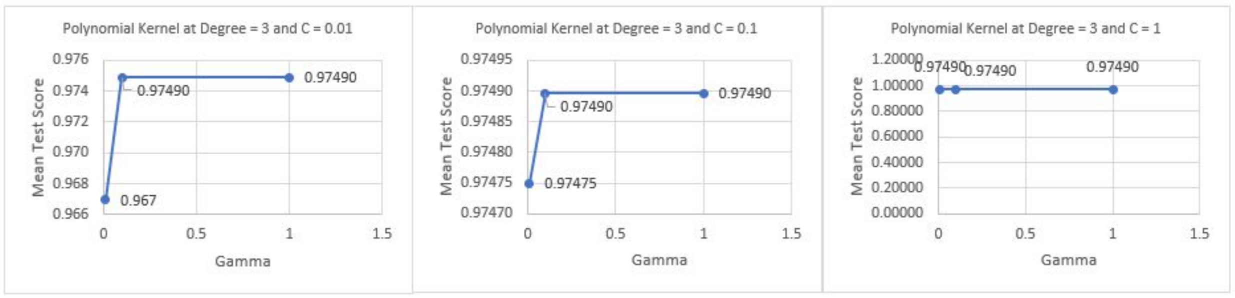


Figure 2.3 - Mean Test Score of a polynomial kernel of degree 3 for varying C and Gamma values

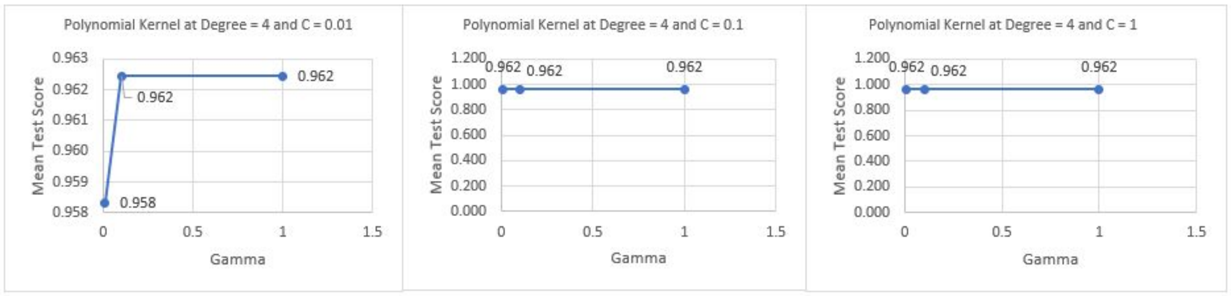


Figure 2.4 - Mean Test Score of a polynomial kernel of degree 4 for varying C and Gamma values

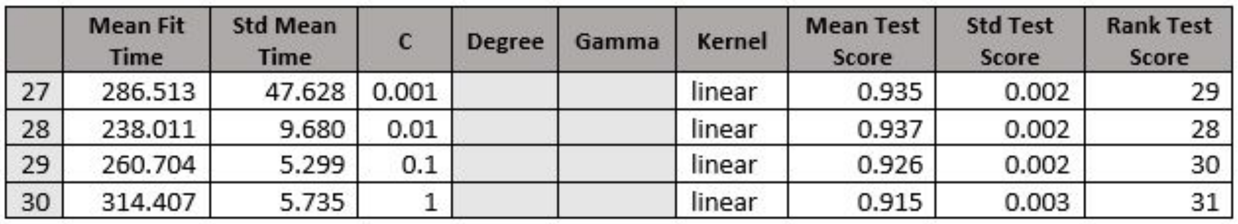
Linear Kernel

The linear kernel variation of the SVM was tested using the following parameters:

- C: [0.001, 0.01, 0.1, 1]

Please refer to Figure 3.1 for an overview of the performance of each candidate model. It illustrates the model fitting time, accuracy score and ranked score (compared to the score of all various kernel variations) of each model. It should be noted that the degree hyperparameter only applies for polynomial kernel, hence it is missing from the table as well as the gamma hyperparameter. Figure 3.2 illustrates the change in accuracy scores for each model of which the various C values that were tested out.

Through observation, it can be seen that linear kernel model having a C value of 0.01 obtain the highest accuracy score. As shown by Figure 3.2, using lower and higher C values than 0.01 decreased the accuracy score. In comparison to the other kernel variations, the linear kernel model obtained the second highest accuracy score (after the polynomial kernel) and had the fastest fitting time.



	Mean Fit Time	Std Mean Time	C	Degree	Gamma	Kernel	Mean Test Score	Std Test Score	Rank Test Score
27	286.513	47.628	0.001			linear	0.935	0.002	29
28	238.011	9.680	0.01			linear	0.937	0.002	28
29	260.704	5.299	0.1			linear	0.926	0.002	30
30	314.407	5.735	1			linear	0.915	0.003	31

Figure 3.1 - Results obtained by hyperparameter tuning using a Linear Kernel

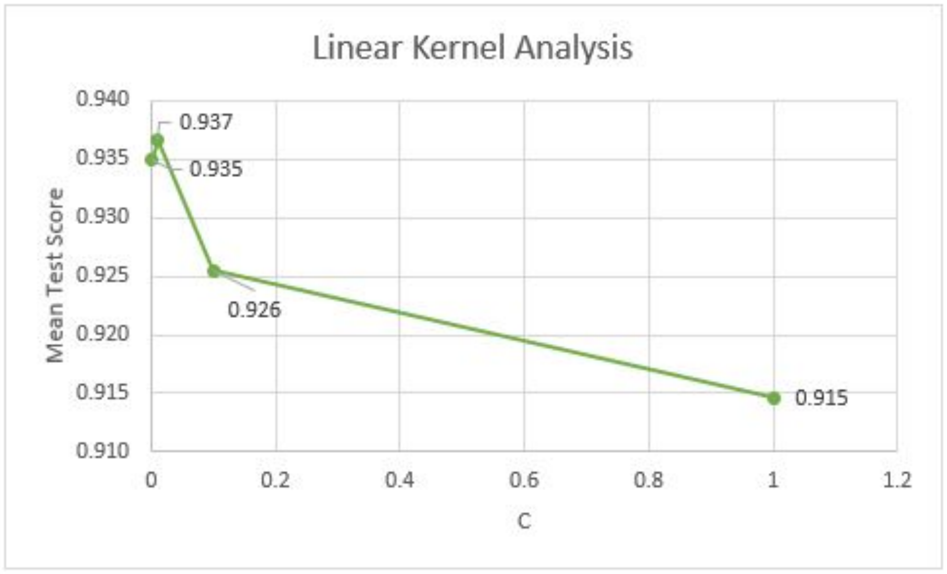


Figure 3.2 - Mean Test Score of a linear kernel for varying C values

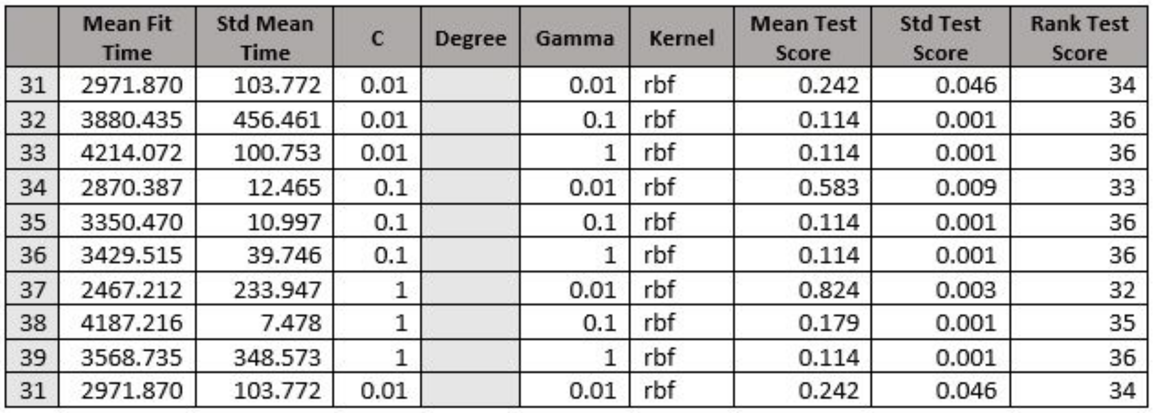
RBF Kernel

The RBF kernel variation of the SVM was tested using the following parameters:

- C: [0.01, 0.1, 1]
- Gamma: [0.01, 0.1, 1]

Please refer to Figure 4.1 for an overview of the performance of each candidate model. It illustrates the model fitting time, accuracy score and ranked score (compared to the score of all various kernel variations) of each model. The degree hyperparameter is missing as it is only used for polynomial kernel models. Figure 4.2 illustrates the change in accuracy scores for the various hyperparameters that were tested.

Through observation, the RBF model that obtained the highest accuracy score (0.824) used a C value of 1 and gamma value of 0.01. As indicated by Figure 4.2, the model performed best with higher values of C and lower values of gamma. In comparison to the other models, the RBF kernel model generated the lowest accuracy score which was surprising considering it took around 5 times longer for fitting than the polynomial kernel model (which had the median fitting time). On the other hand, perhaps the hyperparameters given were not optimal and the obtained results suggest that further reducing the gamma values and raising the C value may further increase accuracy.



	Mean Fit Time	Std Mean Time	C	Degree	Gamma	Kernel	Mean Test Score	Std Test Score	Rank Test Score
31	2971.870	103.772	0.01		0.01	rbf	0.242	0.046	34
32	3880.435	456.461	0.01		0.1	rbf	0.114	0.001	36
33	4214.072	100.753	0.01		1	rbf	0.114	0.001	36
34	2870.387	12.465	0.1		0.01	rbf	0.583	0.009	33
35	3350.470	10.997	0.1		0.1	rbf	0.114	0.001	36
36	3429.515	39.746	0.1		1	rbf	0.114	0.001	36
37	2467.212	233.947	1		0.01	rbf	0.824	0.003	32
38	4187.216	7.478	1		0.1	rbf	0.179	0.001	35
39	3568.735	348.573	1		1	rbf	0.114	0.001	36
31	2971.870	103.772	0.01		0.01	rbf	0.242	0.046	34

Figure 4.1 - Results obtained by hyperparameter tuning using an RBF Kernel

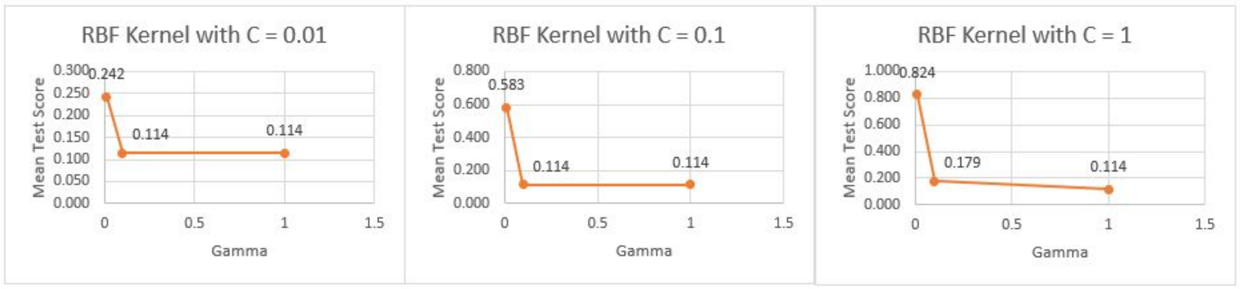
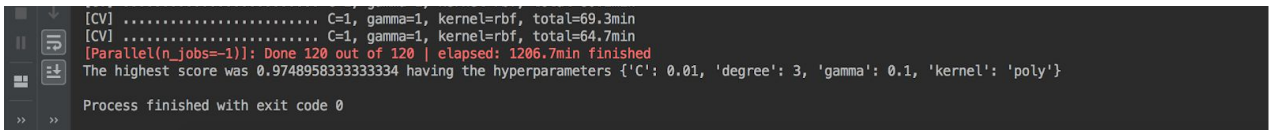


Figure 4.2 - Mean Test Score of an RBF kernel for varying C and gamma values

Summary of Findings

By observing all of the results, it is concluded that the best performing model used a polynomial kernel of degree 3. The values of C and gamma did not appear to have a large impact on the accuracy of the polynomial kernel of degree 3, however, the 'testAllModels()' function deduced that the highest score was obtained using a C and gamma value of 0.01 and 0.1 respectively. The RBF kernel SVM models had a much slower fitting time than all the models and the lowest accuracy, whilst the linear kernel models had the fastest fitting time and their highest accuracy was slightly lower than the polynomial kernel models. Results suggest that higher C values and lower gamma values improved the performance of the RBF kernel model and with further parameter exploring could outperform the polynomial model.



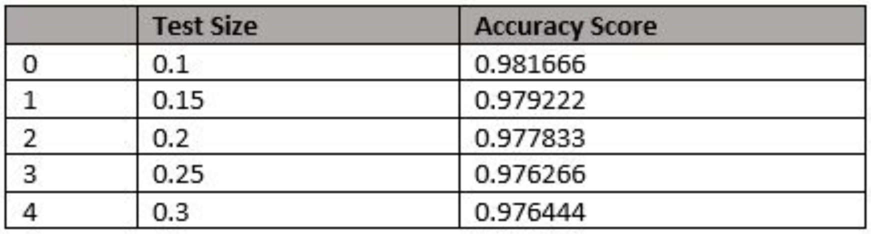
```
[CV] ........................ C=1, gamma=1, kernel=rbf, total=69.3min
[CV] ........................ C=1, gamma=1, kernel=rbf, total=64.7min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 1206.7min finished
The highest score was 0.9748958333333334 having the hyperparameters {'C': 0.01, 'degree': 3, 'gamma': 0.1, 'kernel': 'poly'}

Process finished with exit code 0
```

Figure 4.3 - Results from the 'findAllModels()' function

Different Test Sizes

After deducing the optimal kernel and hyperparameters to use for the SVM model, experimentation proceeded by exploring different test sizes to use for the validation set. This was done by changing the ‘test_size’ variable in the ‘getTrainingData()’ function. Next, the optimal hyperparameters are passed into the ‘testSingleModel()’ function to evaluate the performance of the optimized SVM model on various test sizes. Figure 5.1 - 5.2 illustrate the accuracy score of the SVM model on the different test sizes.

Through observation it can be noted that reducing the test size positively impacted the accuracy of the model. However, the testing size should not be too small otherwise the model will not be properly evaluated. The testing size should also not be too large to avoid decreasing the accuracy score. Based on the results below, an interesting finding was how the accuracy score slightly increased when increasing the test size from 0.25 to 0.30. On the other hand, the change in the accuracy score is very minor to be significant.



	Test Size	Accuracy Score
0	0.1	0.981666
1	0.15	0.979222
2	0.2	0.977833
3	0.25	0.976266
4	0.3	0.976444

Figure 5.1 - Accuracy Scores obtained using a polynomial kernel SVM model of degree 3 and C and gamma value 0.01 and 0.1 respectively

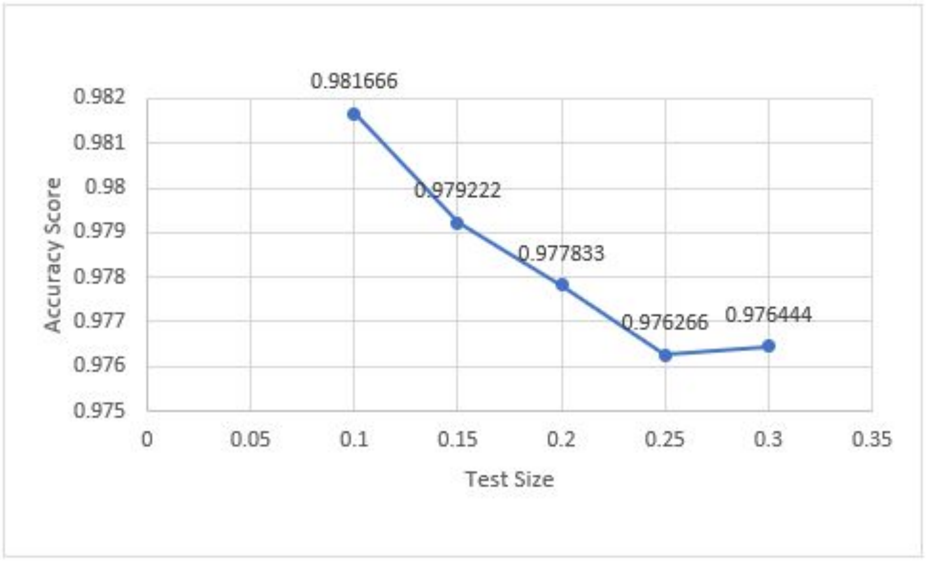
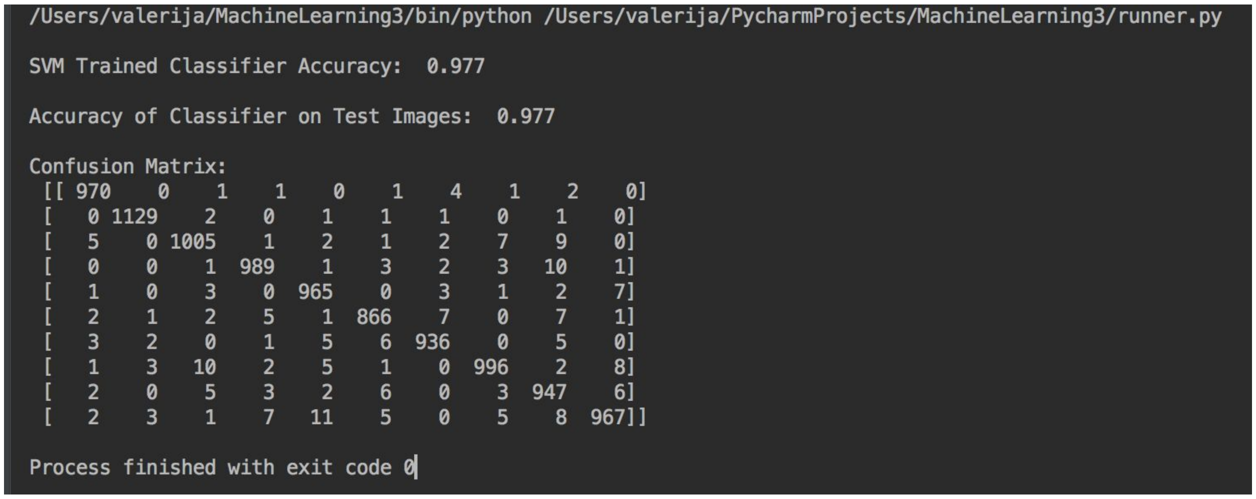


Figure 5.2 - Graph illustrating change in accuracy scores according to varying test sizes

Final Conclusion

Through experimentation carried out in the previous sections, it was deduced that the best performing model was an SVM model with a polynomial kernel of degree 3 and a C and gamma value of 0.01 and 0.1 respectively. The best 'test_size' variable was determined to be a value of 0.9. All of the following parameters were used to build and evaluate the final model on the test set using the 'buildFinalModel()' function and the results of the function are displayed below in Figure 6.1. Figure 6.2 illustrates the count of false negatives and false positives of each label from the confusion matrix.

The results show that the number '8' had the highest number of false negatives whilst the number '1' had the lowest number of false negatives. In addition to this, the number '9' had the highest number of false positives and number '1' had the lowest number of false positives. According to the results, the number '8' was mixed up with the numbers '2,3,5,6 and 9' which are the numbers with a similar written structure to 8. On top of this, number '9' was mostly mixed up for the numbers '3,4,7 and 8'. Overall, the final SVM model achieved a relatively high accuracy of 0.977. Accuracy could potentially be improved by adding more data, exploring further with features and tuning, experimenting with different supervised learning. Another potential method to improve accuracy would be to employ ensemble learning which combines prediction of several estimators to potentially boost the accuracy a single estimator would give.



```
/Users/valerija/MachineLearning3/bin/python /Users/valerija/PycharmProjects/MachineLearning3/runner.py

SVM Trained Classifier Accuracy:  0.977

Accuracy of Classifier on Test Images:  0.977

Confusion Matrix:
 [[ 970    0    1    1    0    1    4    1    2    0]
 [   0 1129    2    0    1    1    1    0    1    0]
 [   5    0 1005    1    2    1    2    7    9    0]
 [   0    0    1  989    1    3    2    3   10    1]
 [   1    0    3    0  965    0    3    1    2    7]
 [   2    1    2    5    1  866    7    0    7    1]
 [   3    2    0    1    5    6  936    0    5    0]
 [   1    3   10    2    5    1    0  996    2    8]
 [   2    0    5    3    2    6    0    3  947    6]
 [   2    3    1    7   11    5    0    5    8  967]]

Process finished with exit code 0
```

Figure 6.1 - Console output from the buildFinalModel() function using the optimal parameters

Labels	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
False Negatives	16	9	25	20	28	23	19	20	46	23
False Positives	10	6	27	21	17	26	22	32	27	42

Figure 6.2 - Evaluation of confusion matrix results