INF 264 Project 3: Digit predictor

Ketil Fagerli Iversen

October 2020

Summary

The goal of the project is to make a classifier which can predict the labels of handwritten digits. We were given a set of 70000 images of handwritten digits and their corresponding labels. I split this dataset into three parts; training, validation and test data. The training data contains 70% of the datapoints, the validation data contains 15% of the data and the test data consists of the last 15% of the data. As each sample contains over 784 features, we are dealing with a 784 dimensional dataset. Thus, during the preprocessing step, i perform feature selection on the data to remove constant features and 'irrelevant' features. I then manage to cut down the amount of dimensions down to 355. This is done to not only shorten training time, but also to make our learners less prone to overfitting.

I then perform model selection on several different models trained on the feature selected data. I evaluate each candidate model on the validation data and select the best performing model, with its hyperparameters. Finally I create a function which takes said selected model as input, some image of a digit, which then will plot the image and give the model's prediction.

Technical report

0.1 Helper functions

normalizer(X):

This function simply normalizes each feature to a value between 0 and 1. It does so by first defining the maximum value in our feature data (255), then dividing each value in the data by this maximum value. This is done so that all features lie within the same range of values. However as all features were within the range of 0 to 255 anyway, this should not be strictly necessary.

```
def normalizer(X):
    # Input: X feature matrix
    # Output: Normalized X feature matrix

    # We know the maximum value a feature can have is 255.
    maxval = 255

    # Divide all the entries by this maximum value to get a number between 0 and 1
    X = X/maxval

    return X
```

data_splitter(X,y):

This function takes a feature matrix X and the target matrix y as inputs, and outputs a training, validation and testing set for both the features and their corresponding targets. It uses Sklearn's **train_test_split()** function with a set random seed for reproducibility. Ive also set the ratios such that the training data consists of 70% of the data, and the test and validation data consists of the remaining 15% + 15%. The datasets are also shuffled before splitting.

```
def data_splitter(X,y):
    # Input: X feature matrix, y target matrix
    # Ouput: Training, testing and validation data

    # I define a function which can split our dataset into training-, validation- and testdata.
    # It does NOT shuffle the data before splitting.

    # Shuffle and split the data into train and a concatenation of validation and test sets
    X_train, X_val_test, y_train, y_val_test = train_test_split(X, y, test_size=0.3, shuffle=True, random_state=0)

    # Shuffle and split the data into validation and test sets
    X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, shuffle=True, random_state=1)

    return X_train, X_val, X_test, y_train, y_val, y_test
```

1.1 Data

Our data consists of 70000 samples which are 28×28 grayscale images. Each image contains one handwritten digit 0 through 9. Each sample has 784 features, each corresponding to each pixel in the image, as $28 \times 28 = 784$. Each feature can range from 0 to 255, (0 to 1 after normalization), corresponding to either a fully white pixel or a fully black pixel respectively.

1.1.1 Preprocessing

I load the dataset features **dataset_imgs** and the dataset labels **dataset_labels** with the **pd.read_csv()** function. I then convert the features and labels into numpy arrays. The features are then normalized to values between 0 and 1 using the **normalizer()** function. This is just done to ensure all features have the same scaling. I then print the number of samples, features and labels to check we get the correct output.

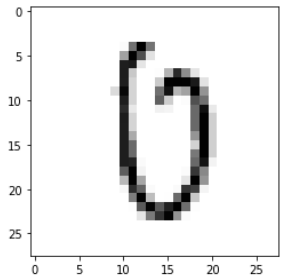
```
# Load the dataset as a pandas dataframe
dataset_imgs = pd.read_csv('handwritten_digits_images.csv', header=None)
dataset_labels = pd.read_csv('handwritten_digits_labels.csv', header=None)

# Extract features to a pandas dataframe X then convert to a numpy array
X = dataset_imgs.to_numpy()

# Extract labels to a pandas dataframe y then convert to a numpy array
y = dataset_labels.to_numpy()

# Normalize the features to values ranging from 0 to 1
X = normalizer(X)

print('Number of samples: ', X.shape[0])
print('Number of features: ', X.shape[-1])
print('Number of labels: ', y.shape[0])
```

```
Number of samples:  70000
Number of features:  784
Number of labels:  70000
```

As we expect, we have 70000 samples, and each sample contains 784 features. We are also able to plot the corresponding images by reshaping a sample to a 28×28 2D array, then using the pyplot's **.imshow()** method. I do this for the first sample in our data,

```
plt.imshow(X[0].reshape(28,28), cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x1b1f4e08cd0>
```



I also print the first sample's label,

```
print(y[0])
```

```
[0]
```

We see that the image is supposed to correspond to the number 0.

Finally, I split our dataset X and y into training, validation and testing datasets, using the **data_splitter()** function,

```
X_train, X_val, X_test, y_train, y_val, y_test = data_splitter(X,y)
```

```
print('Shape of training data:   ', X_train.shape)
print('Shape of validation data: ', X_val.shape)
print('Shape of testing data:    ', X_test.shape)
```

```
Shape of training data:    (49000, 784)
Shape of validation data:  (10500, 784)
Shape of testing data:     (10500, 784)
```

1.1.2 Feature selection

As each sample have 784 features, we are dealing with high dimensional data. High dimensional data require lots of computing to train with. Reducing the number of features will help us to train estimators more quickly to learn the datasets. Another reason is to reduce the likelihood of overfitting to the original data. By removing some features, the estimator is less likely to just 'memorize' the data it is trained on.

In this case, which features can we ignore? We should ignore features that stay constant throughout the dataset. These constant features will be useless for training anyway. In our case, this would most likely be the pixels near the boundary of the image. As the handwritten digits should stay relatively centered, the edge pixels are not likely to change at all (from white to black) between any of the samples. Another way to reduce the amount of features is by ranking each feature by how 'important' they are.

I first define **feature_selector_1** as an sklearn **VarianceThreshold()** feature selector, with the threshold set to 0. This feature selector will look at all the features and remove the ones which have 0 variance. In other words, those which are constant. I fit it on the training data, then i use this fitted feature selector to remove the exact same features from the validation and testing data aswell. Thus, this process might remove some features from the validation and test sets which were not actually constant, however we are treating validation and test data as 'unseen' data.

Then i define **feature_selector_2** as an sklearn **SelectPercentile()** feature selector, with the percentile set to 50 and the score function set to **chi2**. This feature selector will select the top 50% features with the highest chi-squared statistic. The chi-squared test measures the dependence of the features on the label, thus this feature selector will remove the least relevant features for classification.

```
# Define our first feature seletor as a VarianceThreshold feature selector
# Then fit it using the training data at the same time transforming the training data.
# Then use the fitted selector to remove features from the validation and test data
feature_selector_1 = VarianceThreshold(threshold=0)
X_new_train = feature_selector_1.fit_transform(X_train)
X_new_val = feature_selector_1.transform(X_val)
X_new_test = feature_selector_1.transform(X_test)

# Define our first feature seletor as a SelectPercentile feature selector
# Then fit it using the training data and training labels at the same time transforming the training data.
# Then use the fitted selector to remove features from the modified validation and test data
feature_selector_2 = SelectPercentile(chi2, percentile=50)
X_new_train = feature_selector_2.fit_transform(X_new_train, y_train)
X_new_val = feature_selector_2.transform(X_new_val)
X_new_test = feature_selector_2.transform(X_new_test)
```

```
print('Shape of training data after feature selection:   ', X_new_train.shape)
print('Shape of validation data after feature selection: ', X_new_val.shape)
print('Shape of test data after feature selection:       ', X_new_test.shape)
```

```
Shape of training data after feature selection:    (49000, 355)
Shape of validation data after feature selection:  (10500, 355)
Shape of test data after feature selection:        (10500, 355)
```

As we can see, we have reduced the datasets from containing 784 features, down to just 355. This is a reduction down to 45% of its original size.

I now also make a function which will remove features according to our feature selectors, for any other unmodified feature matrix. It takes some data, and two feature selectors **fs_1** and **fs_2**, as inputs, then outputs the data with some features removed.

```
def transform_features(data, fs_1, fs_2):
    # Input: Feature matrix, and two feature selectors
    # Ouput: Feature matrix with only selected features remaining

    # Remove certain features using the first feature selector
    modified_data = fs_1.transform(data)

    # Then remove other features using the second feature selector
    modified_data = fs_2.transform(modified_data)

    return modified_data
```

1.2 Code

The goal is to produce a classifier that predicts the labels of handwritten digits. I want to try atleast three different types of classifiers and vary the hyperparameters for each one. Then I select the model with the best performance on the validation data. Once I have a selected model, i will approximate it's performance on unseen data by evaluating it on the test data. I will then also make a function which takes some input sample (image of digit) then prints the image and the selected model's prediction.

The performance measure used during model selection and performance approximation is the sklearn's **.score()** method for each of the classifiers. This is the 'accuracy' of the model. In other words, the number of correctly predicted labels over total number of predictions. For example, out of 1000 predictions, if 800 were correctly predicted, then the accuracy would be 0.8 or 80%.

For each of the candidate models, I will also plot the confusion matrix. This will tell us what numbers the algorithms correctly predict, and which numbers the algorithms may struggle more with. In the confusion matrix, we want to have high values along the diagonal, and low values for any entry off the diagonal.

1.2.1 Model selector function

I create a function which takes some candidate model, its hyperparameters and training data as inputs, then outputs the best model, its hyperparameters, the average score during training and the elapsed time spent while doing model selection.

model_selector(model, parameters, X, y, verbose=0, n_jobs=1)

This function uses sklearn's **GridSearchCV()** model selector function. This function takes a candidate model, and a dictionary composing the different hyperparameters the user wishes to vary as inputs. Then, by using the **.fit()** method on the model selector, it trains candidate models by iteratively varying the given parameters.

The **GridSearchCV()** model selector selects the best hyperparameters by training the candidate model with the same hyperparameters 5 times. This is because it uses KFold cross validation for evaluating the performance of the given hyperparameters. The training data is split into 5 parts, then for each of the 5 iterations, one of the parts is kept as 'validation' data to approximate the performance of the model. Then the 5 different scores are averaged. Once the best or most optimal hyperparameters have been found, the best model, the best hyperparameters and its average score on the KFold set, will be returned. The elapsed time of model selection is also returned.

```
def model_selector(model, parameters, X, y, verbose=0, n_jobs=-1):
    # Inputs: A candidate model, its different parameters, training data
    # Outputs: The best model, its parameters, its score during training, the elapsed time of training

    # Create a GridSearch model selector with given estimator and parameter grid
    selector = GridSearchCV(estimator=model, param_grid=parameters, verbose=verbose, n_jobs=n_jobs)

    # Start timer, fit the selector to data. End timer when complete
    start = timer()
    selector.fit(X, y.ravel())
    end = timer()

    # The elapsed time in hours:minutes:seconds:milliseconds
    elapsed_time = timedelta(seconds=end-start)

    # Extract the best training score, the best parameters and the best model for this type of estimator
    best_score = selector.best_score_
    best_params = selector.best_params_
    best_model = selector.best_estimator_

    return best_model, best_params, best_score, elapsed_time
```

The variables **verbose** refers to how much feedback the user wants to get while model selection is running. A higher integer will mean more feedback. This was used during debugging to remove some parameters from the dictionary that would lead to elapsed times which were in the order of days. **n_jobs** refers to how many CPU cores the user wants to run the code with. A value of -1 will use all cores. The more cores will lead to a faster model selection. However if **n_jobs** is set to **None**, then it will only use one core, which runs quite slow, but it will give more feedback, which was useful for debugging.

1.2.2 Full model selector

I can now finally create the full model selector. This block of code will gather a list of the best models out of a set of candidate models, with their corresponding hyperparameters, scores, elapsed times and names. In my case I have selected a **Stochastic Gradient Descent Classifier (SGDC)**, a **Support Vector Machine Classifier (SVC)**, a **Random Forest Classifier (RNDFC)** and a **Multi-Layer Perceptron Classifier (MLPC)** as my candidate models.

I then create dictionaries containing the different parameters i wish to vary for each of the four candidate models.

SGDClassifier hyperparameters

For the Stochastic Gradient Descent classifier, i want to vary the loss function, the regularization and the constant value deciding the strength of regularization. Regularization will penalize the complexity, so in theory, a stronger regularization should be less prone to overfitting. I will vary the parameters as,

loss: [hinge, log, modified_huber]

penalty: [l2, l1, elasticnet]

alpha: [0.0001, 0.01, 0.1]

This leads to $3 \times 3 \times 3 = 27$ different models to train and fit, each 5 times.

SVC hyperparameters

For the Support Vector Machine classifier, i wish to vary the kernel used and the constant which decides how strongly misclassifications are penalized. The kernels will decide the shape of the decision boundaries, wheras the constant will decide how 'sharp' the decision boundaries will be. Here, a smaller value for the constant should be benefitial for avoiding overfitting. I will vary them as,

kernel: [linear, poly, rbf]

C: [0.01, 0.1, 1, 10]

This leads to $3 \times 4 = 12$ different models to train and fit, each 5 times.

Random Forest Classifier hyperparameters

For the Random Forest Classifier, i wish to vary the number of estimators used, the criterion for splitting, and the amount of pre-pruning. With a higher number of estimators, we will more likely get a more accurate result via the Condorcet's jury theorem. With some amount of pre-pruning, we could avoid overfitting. I will vary the parameters as,

n_estimators: [100, 200, 300]

criterion: [gini, entropy]

max_depth: [None, 5, 10, 25]

This leads to $3 \times 2 \times 4 = 24$ different models to train and fit, each 5 times.

Multi-Layer Perceptron hyperparameters

For the Multi-Layer Perceptron, i wish to vary both the number of neurons in each layer and the number of layers and the constant deciding the strength of regularization. These values have been arbitrarily selected. I read that the number of neurons should be some value between the number of input nodes, and the number of output nodes. We have (for the case when we use feature selected data) 355 input nodes, and we have 10 output nodes (corresponding to each class). Thus i have just arbitrarily selected 128 and 256 as two values between these two. I have also selected 2 to 3 hidden layers as this dataset is sure to not be linearly seperable. I also vary the parameter which determines the strength of regularization so that our model is less prone to overfitting.

hidden_layer_sizes: [(128,128), (128,128,128), (256,256), (256,256,256)]

alpha: [0.0001, 0.01, 0.1]

This leads to $4 \times 3 = 12$ different models to train and fit, each 5 times.

```
# Create a list of candidate models
model_list = [SGDClassifier(random_state=0), SVC(random_state=0),
              RandomForestClassifier(random_state=0), MLPClassifier(random_state=0)]

# Create a list of model names
model_names = ['Stochastic Gradient Descent Classifier', 'Support Vector Classifier', 'Random Forest Classifier'
              , 'Multi-Layer Perceptron Classifier']

# Create dictionaries of hyperparameters to vary for the respective candidate models
SGD_parameters = {'loss':['hinge', 'log', 'modified_huber'], 'penalty':['l2', 'l1', 'elasticnet'], 'alpha':[0.0001, 0.01, 0.1]
SVC_parameters = {'kernel':['linear', 'poly', 'rbf'], 'C':[0.01, 0.1, 1, 10]}
RNDF_parameters= {'n_estimators':[100, 200, 300], 'criterion':['gini', 'entropy'], 'max_depth':[None, 5, 10, 25]}
MLP_parameters = {'hidden_layer_sizes':[(128,128), (128,128,128), (256,256), (256,256,256)], 'alpha':[0.0001, 0.01, 0.1]}

# Create a list of respective hyperparameter dictionaries
parameter_list = [SGD_parameters, SVC_parameters, RNDF_parameters, MLP_parameters]

# Initialize variables
best_model_list = []
best_params_list = []
best_score_list = []
elapsed_time_list = []

# Start timer
start = timer()

# For each model in the model list, vary the hyperparameters and select the best model based on the score on the kFold
# training data. Then evaluate the best model on the validation data.
for i in range(len(model_list)):

    # Aquire the best model and its hyperparameters for each candidate model
    best_model, best_params, best_score, elapsed_time = model_selector(model_list[i], parameter_list[i], X_new_train, y_train,
                                                                       verbose=3, n_jobs=-1)

    # Append the information to the lists
    best_model_list.append(best_model)
    best_params_list.append(best_params)
    best_score_list.append(best_score)
    elapsed_time_list.append(elapsed_time)

# End timer
end = timer()

# Save the total elapsed time for model selection
total_elapsed_time = timedelta(seconds=end-start)
```

```
# Define our first feat
# Then fit it using the
# Then use the fitted s
feature_selector_1 = Va
X_new_train = feature_s
X_new_val = feature_sel
X_new_test = feature_se

# Define our first feat
# Then fit it using the
# Then use the fitted s
feature_selector_2 = Se
X_new_train = feature_s
X_new_val = feature_sel
X_new_test = feature_se
```

```
print('Shape of trainin
print('Shape of validat
print('Shape of test da
```

```
Shape of training data
Shape of validation dat
Shape of test data afte
```

This block of code loops through all the candidate models and selects the best hyperparameters for each via KFold cross validation. Then that information is saved in their corresponding lists. The total elapsed time for all the model selections is also recorded to get a feel for how long this block of code takes to run. When running the block, I get the following feedback,

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed: 12.0min
[Parallel(n_jobs=-1)]: Done 135 out of 135 | elapsed: 13.1min finished
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed: 55.1min
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed: 111.5min finished
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:  2.8min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 20.0min finished
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed: 10.7min
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed: 53.4min finished
```

As we can see, for the **SGDC** and the **RNDFC** we have more candidates than for the **SVC** and the **MLPC**. This is because the two latter estimators take orders of magnitude more time to train. The **SGDC** and the **RNDFC** take about 20 minutes to train about 24-27 models, however **SVC** and the **MLPC** take about 1-2 hours to train 12 models. This is why i do not have as much variation in the hyperparameters for these slow models, as the code is already slow enough as it is.

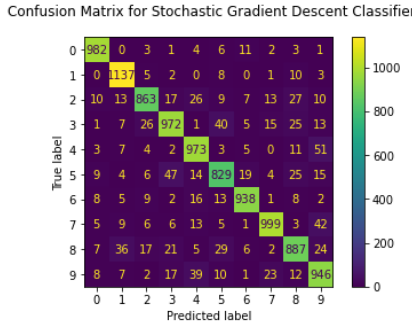
1.2.3 Performance of candidate models & the final selected model

Now we want to print all the relevant information needed to select our final model. I will loop over all the discovered models from the previous block of code, then plot their performances, their hyperparameters and their confusion matrices. Then I will find the model which had the very best performance on the validation set.

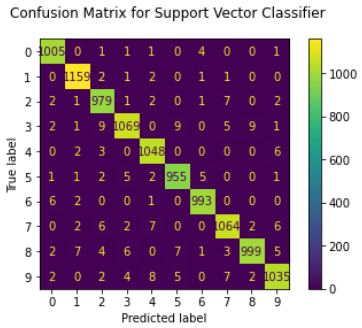
```
# Initialize the best validation score
best_val_score = 0

print('The total time spend doing model selection was: ', total_elapsed_time)

# For each of the best models, compute the validation score. Print model name, elapsed time, the optimal parameters and
# the models score on validation data. Then plot the confusion matrix for the validation data with the model.
# Then score whatever model achieved the highest validation score as the selected_model
for i in range(len(best_model_list)):

    val_score = best_model_list[i].score(X_new_val, y_val.ravel())

    print('The learning time for the best ', model_names[i], ' was: ', elapsed_time_list[i])
    print('The best ', model_names[i], ' scored: ', best_score_list[i], ' on the kFold training data.')
    print('The best parameters were: ', best_params_list[i])
    print('The validation score of the best ', model_names[i], ' is: ', val_score)

    # Plot the confusion matrix for the current model on the validation data
    disp = metrics.plot_confusion_matrix(best_model_list[i], X_new_val, y_val)
    disp.figure_.suptitle('Confusion Matrix for ' + str(model_names[i]))

    plt.show()

    # Find the best model by comparing current validation score to the previous best
    if val_score > best_val_score:

        selected_model = best_model_list[i]
        selected_model_params = best_params_list[i]
        selected_score = val_score
        selected_time = elapsed_time_list[i]
        selected_model_name = model_names[i]
        best_val_score = val_score
```

This block of code has the following output,

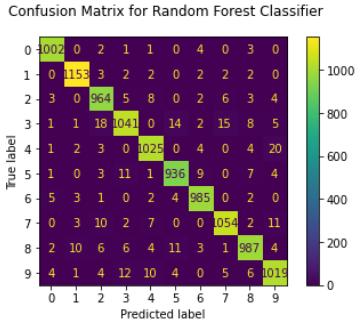
```
The total time spend doing model selection was:  3:24:02.099158
The model selection time for the  Stochastic Gradient Descent Classifier  was:  0:13:21.166263
The best  Stochastic Gradient Descent Classifier  scored:  0.9048979591836733  on the kFold training data.
The best parameters were:  {'alpha': 0.0001, 'loss': 'log', 'penalty': 'elasticnet'}
The validation score of the best  Stochastic Gradient Descent Classifier  is:  0.9072380952380953
```



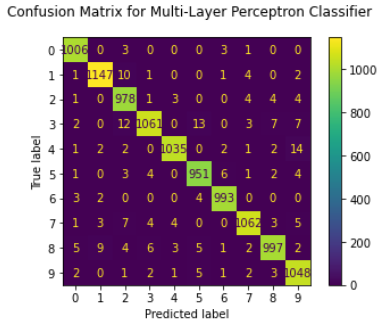

```
The model selection time for the  Support Vector Classifier  was:  1:53:21.981255
The best  Support Vector Classifier  scored:  0.9811020408163265  on the kFold training data.
The best parameters were:  {'C': 10, 'kernel': 'rbf'}
The validation score of the best  Support Vector Classifier  is:  0.9815238095238096
```



```
The model selection time for the  Random Forest Classifier  was:  0:21:17.163235
The best  Random Forest Classifier  scored:  0.964938775510204  on the kFold training data.
The best parameters were:  {'criterion': 'gini', 'max_depth': 25, 'n_estimators': 300}
The validation score of the best  Random Forest Classifier  is:  0.9681904761904762
```




```
The model selection time for the  Multi-Layer Perceptron Classifier  was:  0:56:01.787849
The best  Multi-Layer Perceptron Classifier  scored:  0.9780612244897959  on the kFold training data.
The best parameters were:  {'alpha': 0.01, 'hidden_layer_sizes': (256, 256)}
The validation score of the best  Multi-Layer Perceptron Classifier  is:  0.9788571428571429
```



It might be helpful to collect it all in a table.

Model	Validation Score	Best parameters
SGDC	0.9072	'alpha': 0.0001, 'loss': 'log', 'penalty': 'elasticnet'
SVC	0.9815	'C': 10, 'kernel': 'rbf'
RNDFC	0.9682	'criterion': 'gini', 'max_depth': 25, 'n_estimators': 300
MLPC	0.9789	'alpha': 0.01, 'hidden_layer_sizes': (256, 256)

We see that we get really good performance on both the Support Vector Classifier and the Multi-layer Perceptron Classifier. The leading edge goes to the SVC by 0.003 points. Thus the SVC with the Gaussian kernel (**rbf**) and $\mathbf{C} = 10$, is my selected model. However, the difference in performance between the two is likely within the margin of error and if one would want a model which can train on new data more quickly, one should consider the **MLPC**.

```
print('The selected model is: ', selected_model_name)
print('The parameters of the selected model are: ', selected_model_params)
```

```
The selected model is:  Support Vector Classifier
The parameters of the selected model are:  {'C': 10, 'kernel': 'rbf'}
```

By looking at the different confusion matrices, we can see which numbers the estimators struggle with the most. In the case of our selected model, we can see that we have 8 instances where the number 9 was mislabled as a 4. Similarly we have 9 cases where the number 3 was mislabled as an 8 and so on.

1.2.4 Selected model accuracy on unseen data

Now that we have a final selected model, we can estimate its performance on unseen data by evaluating it on the test set.

```
# Get predicted performance of selected model on unseen data
test_score = selected_model.score(X_new_test, y_test.ravel())

# Print the results
print('The final selected model is: ', selected_model_name)
print('With the hyperparameters: ', selected_model_params)
print('This model had the score of: ', selected_score, ' on the validation data')
print('This model has a score of: ', test_score, ' on unseen test data')
```

```
The final selected model is:  Support Vector Classifier
With the hyperparameters:  {'C': 10, 'kernel': 'rbf'}
This model had the score of:  0.9815238095238096  on the validation data
This model has a score of:  0.9833333333333333  on unseen test data
```

Here we see that our selected model has an expected performance of 0.9833 on unseen data.

1.2.5 Predict and plot

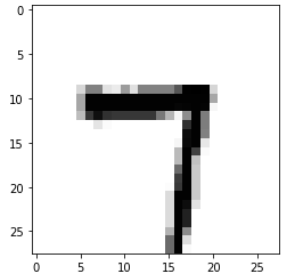
We can now use this model to predict labels to unseen images. I create a function **predict_plot()** which takes the selected model, and a feature sample as inputs. Then outputs the estimators prediction and the image corresponding to the feature sample.

It is important to note that here i have inputed the unmodified test set **X_test** as the feature to look at. NOT **X_new_test**. This is because **X_test** contains all of the $28 \times 28 = 784$ features, and this is how the real world data will be feed to the algorithm. Inside the **predict_plot()** function, I will transform this raw unmodified sample using the **transform_features()** function, thus making it compatible with our selected model. The data will also be normalized by default since we assume the raw data ranges from 0 to 255.

```
def predict_plot(model, data_row, normalize=False):
    # Input: A model, a data sample (image), and wether or not to normalize the data.
    # Output: Label for the image, the models prediction of the label, a plot of the image
    # Used to predict unseen data we dont have the label to.

    # If we need the data to be normalized first, set normalize to True. If the data is already normalized,
    # set normalize to False.
    if normalize == True:
        data_row = normalizer(data_row)

    # Do feature selection on raw data. Then make prediction.
    data_row_mod = transform_features(data_row.reshape(1, -1), feature_selector_1, feature_selector_2)
    prediction = model.predict(data_row_mod)

    print('The estimator has predicted the number: ', prediction)

    print('The actual number looks like,')
    plt.imshow(data_row.reshape(28,28), cmap='Greys')
```

```
predict_plot(selected_model, X_test[0])
```

```
The estimator has predicted the number:  [7]
The actual number looks like,
```



Here we see that our selected model has correctly predicted that the handwritten digit is a 7.

1.2.6 Implementation

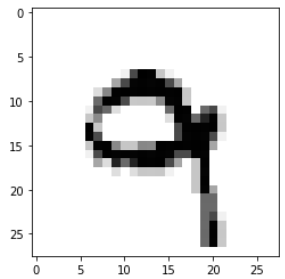
Now we can finally run the final model as we would in a real implementation. Given that we have our selected model (**selected_model**), and we have our two feature selectors **feature_selector_1** and **feature_selector_2**, we can now take any 28×28 image of a digit, and get a prediction out. To define our selected model and the feature selectors, we need to run all the cells in this Jupyter notebook in sequence first!

I now load the **handwritten_digits_images.csv** file again as a toy dataset. Then i feed an arbitrary sample from this dataset through the **predict_plot()** function with our selected model and the sample. Notice now that our raw data ranges from 0 to 255, thus we set **normalize** to True. The function then plots the image and predicts the label, as we wanted out of our algorithm.

```
# Load the data again as toydata
toydata_imgs = pd.read_csv('handwritten_digits_images.csv', header=None)
X_toy = toydata_imgs.to_numpy()
```

```
# Predict and plot any of the samples in the toydata

sample_idx =  63456

predict_plot(selected_model, X_toy[sample_idx], normalize=True)
```

```
The estimator has predicted the number:  [9]
The actual number looks like,
```



1.2.7 Further comments

Given enough time and computing power i would have changed a lot of things. One of the main issues is that the model selection process takes several hours to complete, thus making debugging quite tedeous. In some cases the model selection would get stuck on a specific hyperparameter because it would take more than 8 hours to complete. That is why my hyperparameters do not vary all that much. Ideally i would have a wide range of hyperparameters to try. I would also wish to vary the threshold in **feature_selector_2** as a part of the model selection process. I found that a percentile of 50 was a nice balance of accuracy and model selection time. Anything lower than 30 would severly degrade the accuracy of the estimators. However if i made this a part of the model selection process, it would likely increase the model selection time from 3.5 hours to more than 35 hours if I change the threshold in increments of 10.