INF264 Obligatory Assignment 2

Brage L. Aasen

October 2024

1 Introduction

In this project I designed and built a classifier that can take input grayscale 20x20 images, and identify its contents based on the numeric, letter or empty visualizations of 0-9 (numbers), A-F (letters), or empty contents. The class labels are numbered from 0-16, where 0-9 is the numbers, 10-15 is the letters, and 16 is Empty.

In addition to this, I tested dimensionality reduction by the use of PCA. I also tried detecting out of distribution images (OOD) from a similar but corrupt dataset, of which had no labels, and new image contents in the form of clothing.

2 Summary

2.1 Problem 1: Digit Recognizer

To solve the problem of identifying the grayscale images to their respective label, I trained two different classifiers and performed model selection with hyperparameter tuning. By doing this, I found a suitable classifier for the problem at hand. The classifiers I decided to try out for image recognising was SVM (Support Vector Machine) and Random Forest. After performing model selection, I found that the SVM outperformed the Random Forest classifier in terms of F1, precision, and recall across various classs.

The results from each respective classifier on the test set were as follows (from task 1, e.i, pre-PCA):

Metric	**SVM**	**Random Forest**
F1 Score (Macro)	`0.888533303881354`	`0.8349082617662811`
F1 Score (Weighted)	`0.9119598921495807`	`0.8513775494642909`
Precision (Weighted)	0.91	0.85
Recall (Weighted)	0.91	0.85

Table 1: Comparison of Performance Metrics for SVM and Random Forest

The SVM classifier performed well in distinguishing between the different image classifications because of its ability to find optimal decision boundries in high-dimensional feature space. Since the images of our dataset were grayscale 20x20 images, the feature space was 400. Hence, the SVM was a suitible choice for this image recognition task. This was further supported by the F1 scores observed in the classification report, and in the table above. The weighted F1 score (taking the number of samples for each class into account) for SVM performed significantly better than its counterpart Random Forest. In the SVM I observed a score of ≈ 0.912, and in Random Forest a score of ≈ 0.851.

While Random Forest also performed reasonably well, it was slightly less effective compared to the SVM. I suspect that this has to do with the complexity and high dimensionality of the feature values.

Given the high-dimensional nature of the data and the goal of accurately classifying handwritten letters and digits, SVM proved to be an appropriate and effective machine learning approach for this task, demonstrating better performance compared to Random Forest.

I expect the SVM to generalize well to real-life data, especially when similar grayscale images are presented to the model. But, this is provided that the given images are of similar quality and distribution as the training data. This means that the same pre-processing steps made on the dataset used during model development, should be made on the new datasets used for testing real life data. E.i, scaling and normalizing the grayscale images before predicting the data's classifications. Without such pre-processing on new real life data, it will likely compromise the model's success and accuracy of its predictions.

Even though I would expect good performance from the model in practise, there are still potential challenges for the model when faced with new real life data. Given some data of handwritten letters and numbers, the model's performance might be reduced significantly due to some of the following reasons:

- The captured images have varying resolutions.
- Incomplete or badly written characters are present in the dataset.
- The images contain noise or are distorted.

These challenges could lead to further decreased performance in real-life scenarios if not addressed beforehand, as the model's ability to recognize letters and digits may be compromised.

2.2 Problem 2: Dimensionality Reduction

For this task I used Principal Component Analysis (PCA) to perform dimensionality reduction on the pre-processed (scaled and normalized) dataset.

I defined various values of k (the number of components for PCA), and reduced the dimensionality of the dataset on these k's. Then for each value of k, I chose the best hyperparameters by using a grid search, with accuracy as its performance measure. Here I chose accuracy instead of F1 Score since the data had been applied SMOTE in the pre-processing, e.i, the dataset is not unbalanced. For each value of k, I also tracked its computation time taken. The results are visualized to further analyze the impact of the PCA dimensionality reduction on the model efficiency and performance. From this analysis, I derived an optimal k (number of components) to use for my model.

2.3 Problem 3: Detecting Out-of-Distribution Images

In the last problem of the project, detecting Out-of-Distribution (OOD) images, I used a uncertainty probability based approach. This approach involved using a model that was uncertainty based, combined with PCA for dimensionality reduction.

I trained the best model on the entirety of the labeled data (normal dataset), and then applied it to the unlabled data (corrupt dataset) to detect the OOD images. The OOD images that my model detected was the images with the highest uncertainties in its predictions.

3 Technical report

3.1 My observations about the dataset

For the analysis of data, I used matplotlib.pyplot and numpy to visualize the datasets.

```
import numpy as np
import matplotlib.pyplot as plt

# Load the dataset
dataset = np.load("../dataset.npz")
X, y = dataset["X"], dataset["y"]
```

To understand the dataset better, I started by visualizing its class distributions

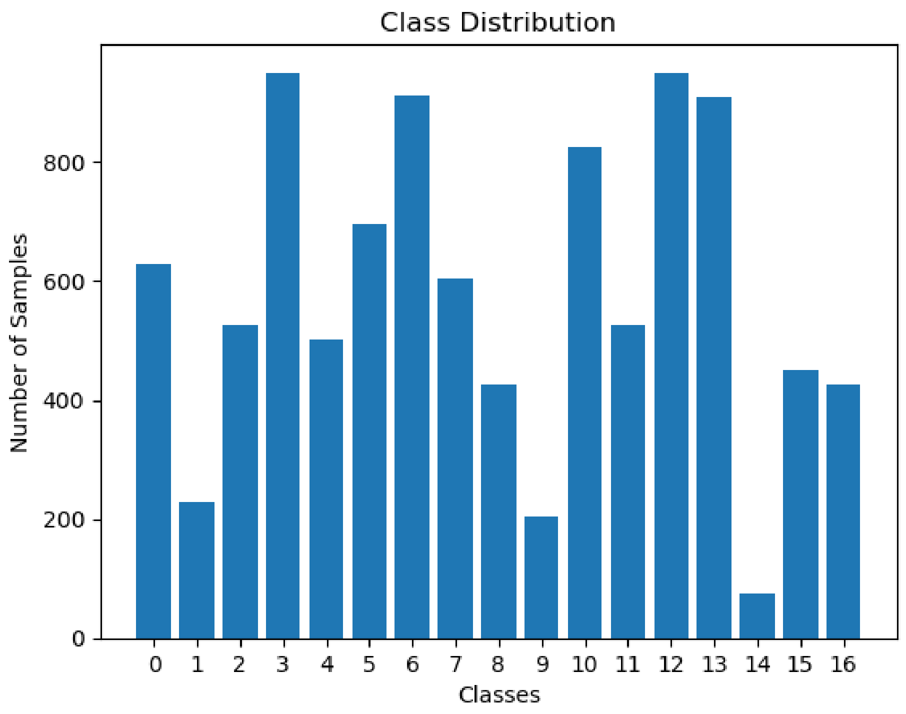

Figure 1: Class Distributions

The plot clearly shows that the dataset has an imbalanced class distribution, particularly in classes such as 1, 9, and 14, which have significantly fewer samples compared to other classes.

Furthermore, I wanted to visualize the distribution of pixel values (the features) across all of the images in the dataset. I created a plot of histograms that shows the frequency of different gray-scale values (0-255) across every sample. This plot lets me observe which features contains large values, e.i, many images may have the gray-scale value 0 (black) due to the nature of the dataset.

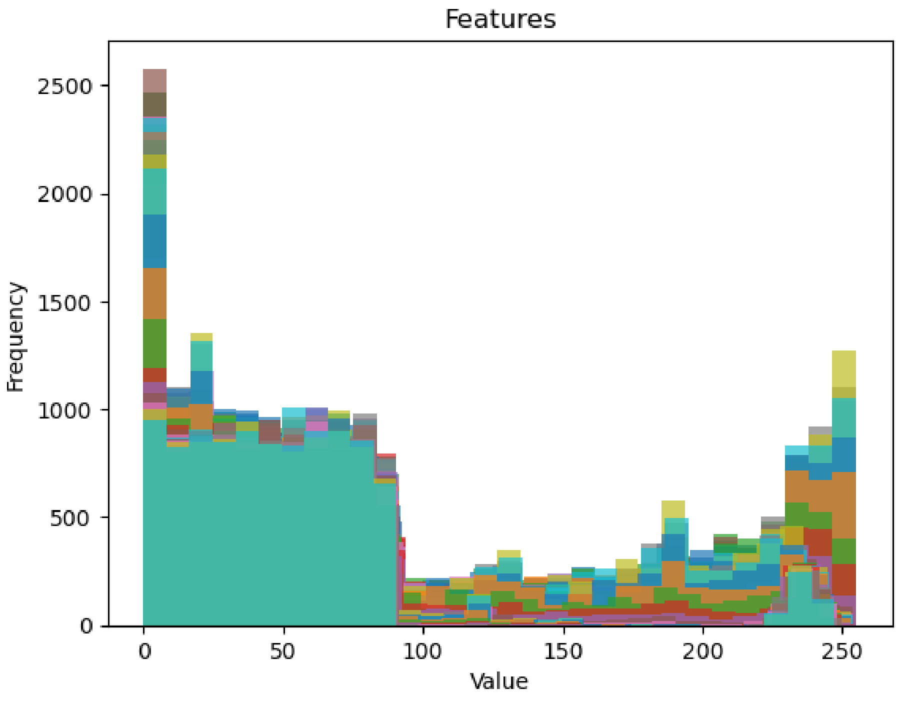

Figure 2: Feature Distributions

From this plot we can see that there are primarily two dominant peaks. We can see a large peak at 0, which tell us that many of the pixels in the dataset are black. This means that many of the sample images has dark regions. We can also note from the second peak around 255, that there are also many white parts. Lastly, we can observe that around the middle regions of the gray-scale (gray), there are a large dip of frequency. This means that our sample images are primarily dark or bright.

3.2 Pre-processing steps

With the observations from the previous section in mind, I chose to first normalize the dataset based on the gray-scale values. The pixel color values in the dataset range from 0 to 255, where 0 represents black and 255 represents white. To ensure that all of the features are on a consistent scale for model training, I normalized the pixel values by dividing them by 255.

```
# Pre-processing
grayscale = 255.0
X = X / grayscale
```

Now, the features are in a range of [0, 1] instead of [0, 255]. This is an important pre-processing step in my case, because machine learning algorithms such as SVM and Random Forest tend to perform better when the input features are normalized to a standard range. By normalizing the data, we can prevent certain features from dominating others, e.i, our model can effectively learn from all features.

Next, I split the data into training and test sets before doing any more pre-processing. This is because scaling after splitting the datasets prevents data leakage from the test set into the training set, which would in turn artificially boost the performance measure of my model on the test set.

```
# Split the data, 70% for training and 30% for test
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=seed, shuffle=True, stratify=
    y)
```

When I split the dataset into training and test sets, I set the parameter stratify=y. I did this to ensure that the class distribution would remain the same, both in the training and test sets. By doing this, the proportions of samples for each class in y would remain consistent across both the training and test sets.

Furthermore, I scaled the data with StandardScaler to calculate the mean and standard deviation of the features in the training set, and then applied those same scaling parameters to the training and test set.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

By doing this, the training data will not see the test data during the model training process, and will eliminate any potential bias from the test data.

The results of normalizing and scaling the data can be seen when I plot the feature frequencies again:

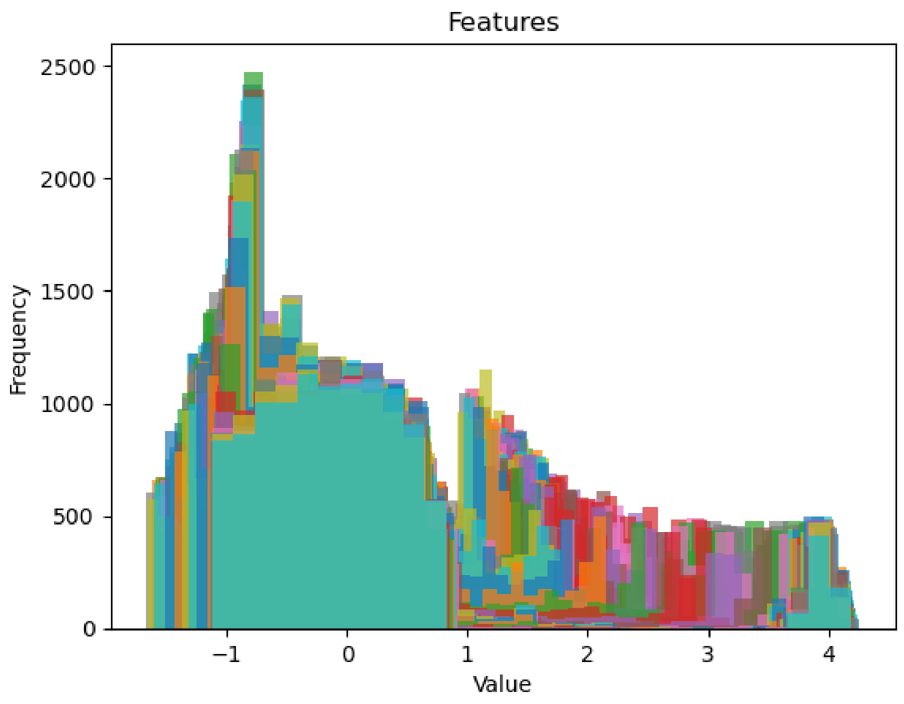

Figure 3: New Feature Distributions

The next pre-processing step I performed was to oversample the training data to reduce my model's bias towards the majority classes.

```
smote = SMOTE(random_state=seed)
X_train_smote, y_train_smote = smote.fit_resample(
    X_train_scaled, y_train)
```

To oversample the training data, I applied SMOTE. This generated synthetic samples for the minority classes, which helped balance out the class distributions in the training set. Now the model will be improved when it comes to learning equally from all of the classes. An important factor is that I only applied SMOTE to the training data. By doing so, I can assess the model's true performance in conditions that reflect the actual class distribution it would encounter in practice with real world data.

The results of oversampling the data can be seen when I plot the class distribution again:

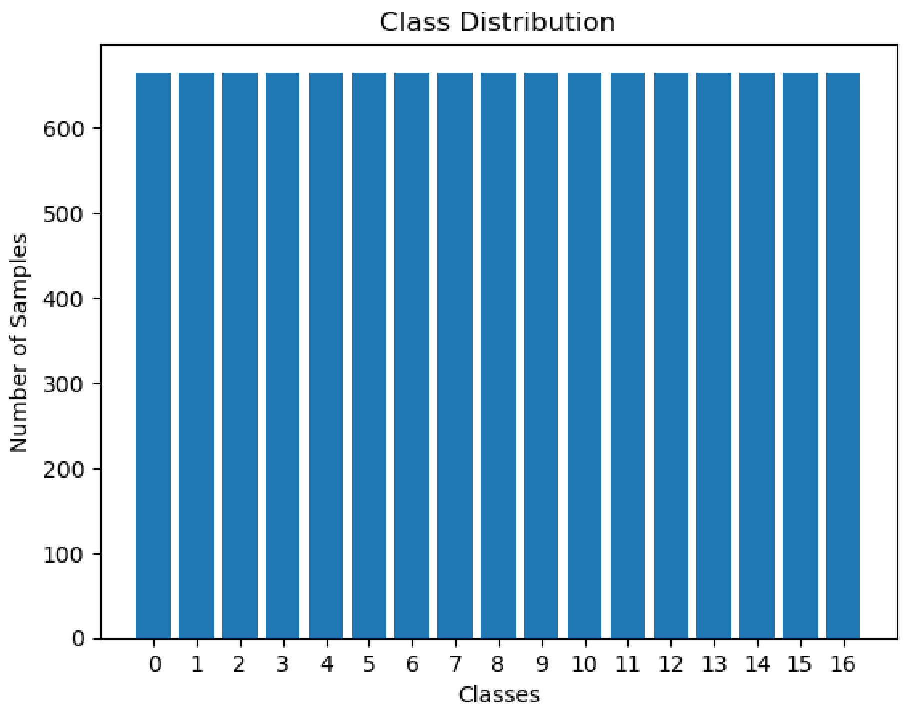

Figure 4: New Class Distributions

3.3 Choice of candidate models and hyperparameters

To solve the problem issued of classifying grayscale images, I decided to go for the two candidate models Support Vector Machine (SVM) and Random Forest. I chose these candidate models because they both perform well in classification tasks such as high-dimensional data like images, and I wanted to test them out further in practise.

3.3.1 SVM

As stated in my summary of the project, SVM is very effective in high-dimensional spaces, which makes it a good choice for our specific problem where each image is 20x20 pixels (each image is represented by 400 features). SVM excels at finding an optimal hyperplane to separate classes, even when faced with complex class boundaries, making it a generally excellent choice for image classification tasks.

The hyperparameters I chose to tune for the SVM were as follows:

- **C (regularization)**: This controls the trade-off between minimizing the classification errors and maximizing the margin.

- **Kernel**: Influences how well the decision boundary can handle non-linear separations in our data.
- **Gamma**: This determines the influence of each training sample for the RBF kernel.

I decided to remove both the polynomial kernel and the degree hyperparameter to keep the tuning of the SVM within 40 minutes, while still exploring a substantial number of other hyperparameters. I also had to limit myself to rather few values hyperparameter grid, for the same reason as stated above.

I found the following hyperparameter grid for SVM to work well for me (taken computation time into account):

```
# Support Vector Machine Classifier
svm_clf = SVC(random_state=seed)
svm_params = {
    "C": [0.1, 1, 10],
    "kernel": ["linear", "rbf"],
    "gamma": ["scale", "auto"],
}
```

3.3.2 Random Forest

I chose Random Forest as my second candidate model firstly because it is a versitile and robust ensemble machine learning method that works by aggregates predictions from many different decision trees. Random Forest is also able to handling complex, non-linear relationships in the data and can provide a strong baseline performance for classification tasks such as the one we are faced with in this project. And secondly, because I became very familiar with it from the first project, and wanted to explore its capabilities in an image recognition task.

The hyperparameters I chose to tune for the Random Forest were as follows:

- **Number of trees (n_estimators)**: The number of decision trees that are built.
- **Maximum depth (max_depth)**: The max depth of the trees, which can help control overfitting.
- **Criterion**: This determines the function that is used to measure the quality of a given split.
- **Max features**: This defines how many features we want to consider for each split in the tree.

I found the following hyperparameter grid for Random Forest to work well for me (taken computation time into account here as well):

```
# Random Forest Classifier
rf_clf = RandomForestClassifier(random_state=seed)
rf_params = {
    "n_estimators": [10, 30],
    "max_depth": [10, 50],
    "criterion": ["gini", "entropy"],
    "max_features": ["sqrt", "log2", None],
}
```

I determined that it was necessary for my use case to keep the number of estimators low and avoid making the grid search space too large. Even with this specific grid, using 5-fold cross-validation in the grid search meant performing 5 folds for each of the 24 hyperparameter candidates, resulting in 120 total fits. Performing this many fits for the Random Forest classifier significantly increased the overall computational time.

But, even with a reduced grid search space, I was happy with the performance of the model.

3.3.3 Why Other Models Were Omitted

I initially considered using K-Nearest Neighbors (KNN), but I quickly decided against it. While KNN is simple and effective for small datasets, it becomes computationally expensive and slow when applied to high-dimensional datasets like the one we are faced with in this project.

3.4 Chosen performance measure

For this project, I chose accuracy and F1 score as the primary performance metrics, where I selected the appropriate metric based on whether the dataset was SMOTEd or not.

3.4.1 Accuracy for Datasets with applied SMOTE

In the cases where the training dataset was balanced with the use of SMOTE, I used accuracy as the chosen performance measure. By using SMOTE on the training dataset, the dataset's classes will be artificially balanced by new synthetic samples for minority classes as discussed previously. With the dataset no longer imbalanced, accuracy is a suitable metric because each class is equally represented in the dataset.

Accuracy is an effective performance metric when dealing with balanced datasets. So, in our case where SMOTE has mitigated the class imbalance in the training set, accuracy will provide a good and simple measure of the overall performance. Examples of where accuracy is used as the performance metric, can be seen for example in the Grid Search Cross-Validation, where the SMOTEd dataset is used.

```
# SVM
svm_clf_grid = GridSearchCV(svm_clf, svm_params, cv=
    cross_validation, scoring="accuracy", verbose=1)
svm_clf_grid.fit(X_train_smote, y_train_smote)
...
# Random Forest
rf_clf_grid = GridSearchCV(rf_clf, rf_params, cv=
    cross_validation, scoring="accuracy", verbose=1)
rf_clf_grid.fit(X_train_smote, y_train_smote)
```

3.4.2 F1 Score for Datasets without applied SMOTE

In the cases where the datasets were not balanced, e.i, without SMOTE applied, I chose F1 score as the performance metric. F1 score, as the harmonic mean of precision and recall, is a well-suited metric for evaluating performance on imbalanced datasets. I did not use accuracy in the cases of imbalanced datasets, because it can be misleading by favoring the majority class. By the use of F1 Score, I can account for both the false positives and false negatives, which in turn will give me a better, and more balanced view of the overall performance of the model. I decided to focus on the weighted F1 score, which accounts for class imbalance by the use of weights, e.i, it assigns weights to each class based on their frequency in the dataset.

F1 Score penalizes the model for misclassifying the minority classes, which provides a more reliable measure of the performance in cases where specific classes have fewer samples than others. As mentioned earlier, such classes in our case are: 1, 9, and 14.

Examples of where F1 Score is used as the performance metric:

```
## Evaluate the best model on the test data
# If SVM was chosen as the best model:
y_pred_svm_clf = svm_clf_grid.predict(X_test_scaled)
f1_weighted_svm_clf = f1_score(y_test, y_pred_svm_clf,
    average="weighted")
...
# If Random Forest was chosen as the best model:
y_pred_rf_clf = rf_clf_grid.predict(X_test_scaled)
f1_weighted_rf_clf = f1_score(y_test, y_pred_rf_clf, average
    ="weighted")
```

Here I used F1 Score to test the best model's performance after it had been trained, tuned, and generated predictions on the test set. As mentioned earlier, F1 scored was chosen in this case because I evaluated on the test sets, which are not SMOTEd, e.i, imbalanced.

3.5 Model selection scheme(s)

For the model selection part of this project, I chose to go with GridSearchCV with K-Fold Cross-Validation to find the model with the best performance and hyperparameter combination.

GridSearchCV was used to automate the process of searching over a range of hyperparameter combinations for both the SVM and Random Forest classifiers. I chose to set my K to 5, and apply 5-fold Cross-Validation during the grid search. I chose K to be 5 because it is a standard choice for K, and yet, not too large (with computational time in mind). This split the dataset into 5 equally sized folds, and for each of the hyperparameter combinations, the model was then trained on 4 of the folds, and validated on the remaining fold. This process was repeated K (5) times, each with a different fold for validation. The result from the 5 folds were averaged to give a reliable estimate of the model's performance for each given set of hyperparameters. This entire process was repeated for every combination in the grid. Once the grid search was completed, the model with the highest Cross-Validation score was selected as the best model, and saved for further evaluation.

The use of K-fold Cross-Validation helped me reduce the variance in the model evaluation, and also helped to prevent overfitting by ensuring that the models were evaluated across different data folds. It also gave me a more accurate estimate on how well the models would generalize to new and unseen data, by training and validating on different subsets of the data.

3.6 What is your final classifier, and how does it work?

After performing model selection and evaluating the performance of both SVM and Random Forest, the SVM was selected as the final classifier for this project. I set up my code such that the model with the best validation accuracy after the GridSearchCV with K-Fold Cross-Validation got chosen like this:

```python
# Choose the best model based on the highest cross-
   validation accuracy achieved for each model
if svm_clf_grid.best_score_ > rf_clf_grid.best_score_:
    best_model = "SVM"
    model_to_use = svm_clf_grid.best_estimator_
    best_model_f1_macro = f1_macro_svm_clf
    best_model_f1_weighted = f1_weighted_svm_clf
    best_params = svm_clf_grid.best_params_
else:
    best_model = "Random Forest"
    model_to_use = rf_clf_grid.best_estimator_
    best_model_f1_macro = f1_macro_rf_clf
    best_model_f1_weighted = f1_weighted_rf_clf
    best_params = rf_clf_grid.best_params_
```

Like mentioned in the summary in the start of the report, the SVM classifier works by finding the optimal hyperplane that separates classes in the feature space. The classifier tries its best to maximize the margin between the closets points from each class to the hyperplane. By doing this, the SVM proves to be effective in the high-dimensional feature space such as in this project where each image has 400 features. In addition to this, the hyperparameter "kernel" in the SVM was in my case tuned to be "RBF", which gave the best performance. This kernel helps SVM handle non-linear relationships in the data by mapping input features into a higher-dimensional space. Mapping the input into a higher-dimensional space helps the classifier to separate classes that are not linearly separable in the original feature space.

The SVM classifier proved itself to be the superior model because of its performance during the model evaluation phase of the project. It got the highest performance over several metrics such as accuracy on the SMOTEd training set, and F1 score on the not SMOTEd test set.

Here is the performance results of the final classifier (SVM):

Metric	**SVM**
F1 Score (Macro)	`0.888533303881354`
F1 Score (Weighted)	`0.9119598921495807`
Precision (Weighted)	0.91
Recall (Weighted)	0.91

Table 2: Results of SVM

With the best parameters for the SVM model being the following:

- **C**: 10
- **Gamma**: `auto`
- **Kernel**: `rbf`

3.7 How well it is expected to perform in production (on new data)

I would expect the final classifier of SVM to generalize well to real-life data, especially when the data is similar to the dataset of which was used during the model development, as discussed in the summary. The final model showed to perform very well on the test set, and achieved an F1 score (weighted) of ≈ 0.912. Based on this performance, I would estimate that the final model would maintain a good performance with new real-world data, but, provided that certain conditions are met.

1. **Data Quality**: Whether or not the model will perform well on new real data, will depend on the similarity between the new data, and the data of which it was trained on. In this project, the model was trained on

pre-processed gray-scale images of handwritten characters. This data was normalized and scaled for training. Given that the new real-world data is pre-processed similarly as the training data, I would assume that the model should maintain its high performance in terms of classification accuracy. If the real-life data is not pre-processed at all, the model's predictions would most likely suffer, and result in a large drop in its accuracy and generalization ability.

2. **Image Quality**: In the summary I also discussed cases where the real-world data differs significantly from the training data in terms of image quality.

Quality differences such as the ones listed below may impact the model's performance:

- The captured images have varying resolutions.
- Incomplete or badly written characters are present in the dataset.
- The images contain noise or are distorted.

However, the SVM model which was built in this projects makes use of the RBF kernel, which allows it to handle non-linear separations in the feature space. This makes the model somewhat robust towards minor variations in the real-life data taken as input. But this doesn't guarantee us a good performance of the model. Larger variations in the data quality or new styles of characters that were not in the training set, could very possibly impact the model's ability to perform well.

3.7.1 A deeper look into the final model's performance

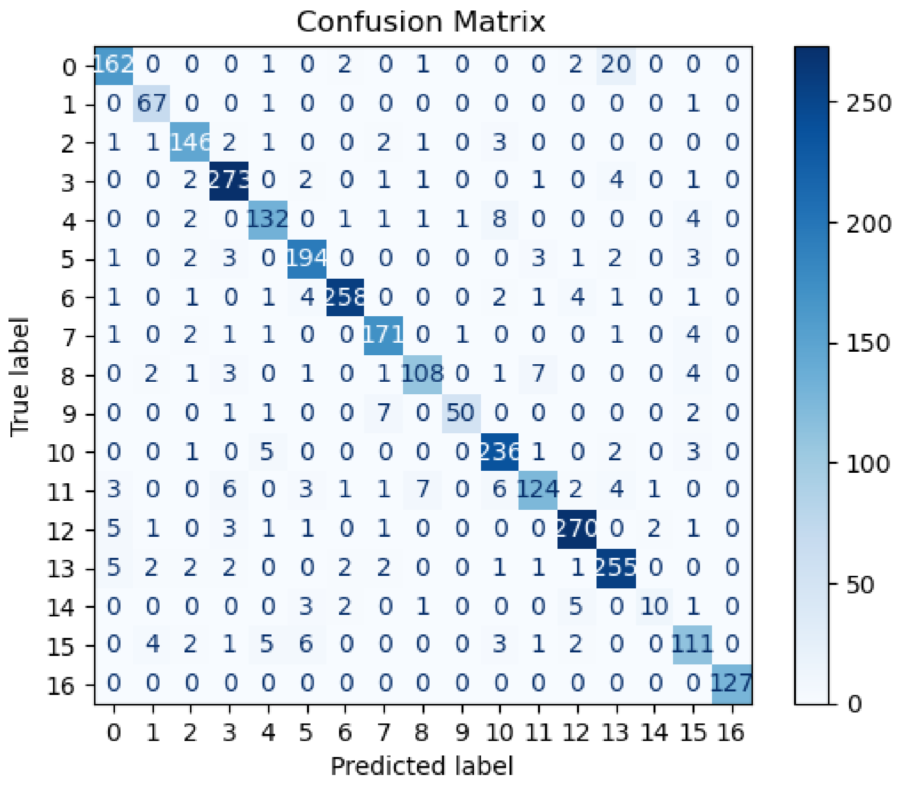

Figure 5: Confusion Matrix of the final model

This confusion matrix shows us how often the predicted labels matches with the true labels. In the plotting, the diagonal entries shows us the number of times the model correctly classified the respective class. On the contrary, the entries on the off-diagonal shows us the number of mis-classifications between the classes.

We can for example see from the confusion matrix that the label 16 is predicted perfectly, which indicates that empty images will in most cases perform very well in real-data as well.

3.8 How did dimensionality reduction (PCA) affect the performance and computational efficiency of your classifier?

I iterated through a predefined list of K's to perform dimensionalty reduction of the feature space. These were the values of K I tested out on best classifier:

```
ks = [25, 50, 100, 150, 300]
```

For each value of K, I performed model selection on the reduced feature space for the best classifier. For the model selection process, I timed the model, and saved various results for each k to use for plotting as visualization.

Here is how I did this:

```
results = {}
best_k = None
highest_accuracy = 0

for k in ks:
    pca = PCA(n_components=k, random_state=seed)
    X_train_pca = pca.fit_transform(X_train_smote)
    X_test_pca = pca.transform(X_test_scaled)

    if best_model == "SVM":
        # Train SVM
        svm_clf_pca = GridSearchCV(SVC(random_state=seed),
            svm_params, cv=cross_validation, scoring="
            accuracy", verbose=1)

        # Time and fit the model
        start_time = time.time()
        svm_clf_pca.fit(X_train_pca, y_train_smote)
        end_time = time.time()

        best_accuracy = svm_clf_pca.best_score_

        results[k] = {
            "X_test_pca": X_test_pca,
            "best_estimator": svm_clf_pca.best_estimator_,
            "best_params": svm_clf_pca.best_params_,
            "best_accuracy": svm_clf_pca.best_score_,
            "time": end_time - start_time
        }

        print(f"Best parameters for SVM with n_components={k
            }:", svm_clf_pca.best_params_)
        print(f"Best accuracy for SVM with n_components={k}:
            ", svm_clf_pca.best_score_)
        print()
        if best_accuracy > highest_accuracy:
            highest_accuracy = best_accuracy
            best_k = k

    elif best_model == "Random Forest":
        # Train Random Forest
        rf_clf_pca = GridSearchCV(RandomForestClassifier(
            random_state=seed), rf_params, cv=
            cross_validation, scoring="accuracy", verbose=1)

        # Time and fit the model
        start_time = time.time()
        rf_clf_pca.fit(X_train_pca, y_train_smote)
        end_time = time.time()

        best_accuracy = rf_clf_pca.best_score_

        results[k] = {
            "X_test_pca": X_test_pca,
            "best_estimator": rf_clf_pca.best_estimator_,
            "best_params": rf_clf_pca.best_params_,
            "best_accuracy": rf_clf_pca.best_score_,
            "time": end_time - start_time
        }

        print(f"Best parameters for Random Forest with
            n_components={k}:", rf_clf_pca.best_params_)
        print(f"Best accuracy for Random Forest with
            n_components={k}:", rf_clf_pca.best_score_)
        print()

        if best_accuracy > highest_accuracy:
            highest_accuracy = best_accuracy
            best_k = k
```

The results of the dimensionality reduction proved to be very good, and actually outperformed my previous model.

Here is the performance results of the final classifier after dimensionality reduction (SVM PCA):

Metric	**SVM PCA**
F1 Score (Macro)	0.8996412216100728
F1 Score (Weighted)	0.9201696564601234
Precision (Weighted)	0.92
Recall (Weighted)	0.92

Table 3: Results of SVM PCA

3.8.1 Understanding the different values of K (number of components)

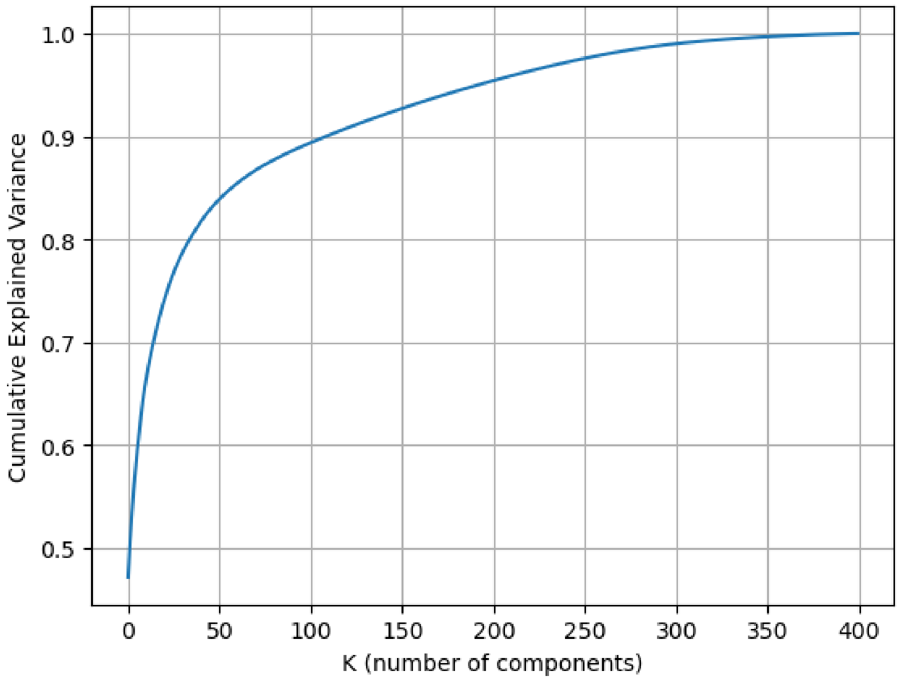

Figure 6: Explained Variance vs K (number of components)

From this graph, I could get an understanding of how much information was lost by the amount the feature space was reduced. Here we can see that a very low value of K, e.g, 25, would result in a large loss of information. But by choosing a larger value, such as 150, the information loss would not be as damaging. From the plotting we can see that further components after 150, contributes less and less to the overall variance, e.i, including more components will not provide substantially benefits of captured variance.

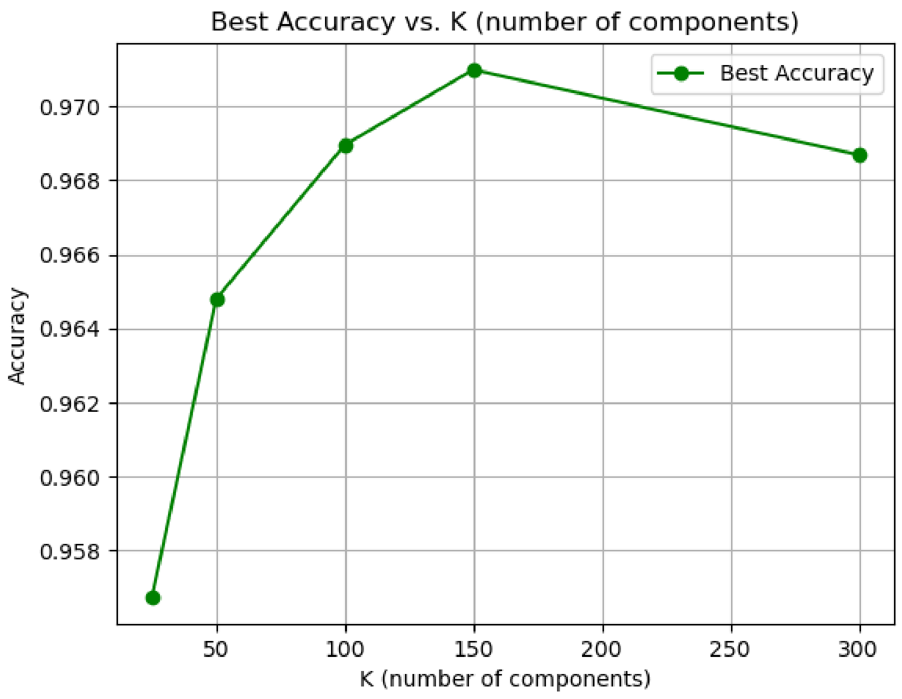

Figure 7: Accuracy vs K (number of components)

Next, we can see that the accuracy of 150 components proved to be very good. This was a large contributor as of why I chose 150 components to be the most suitable K. However, the computation time taken is also a valuable information to consider for what the best K is.

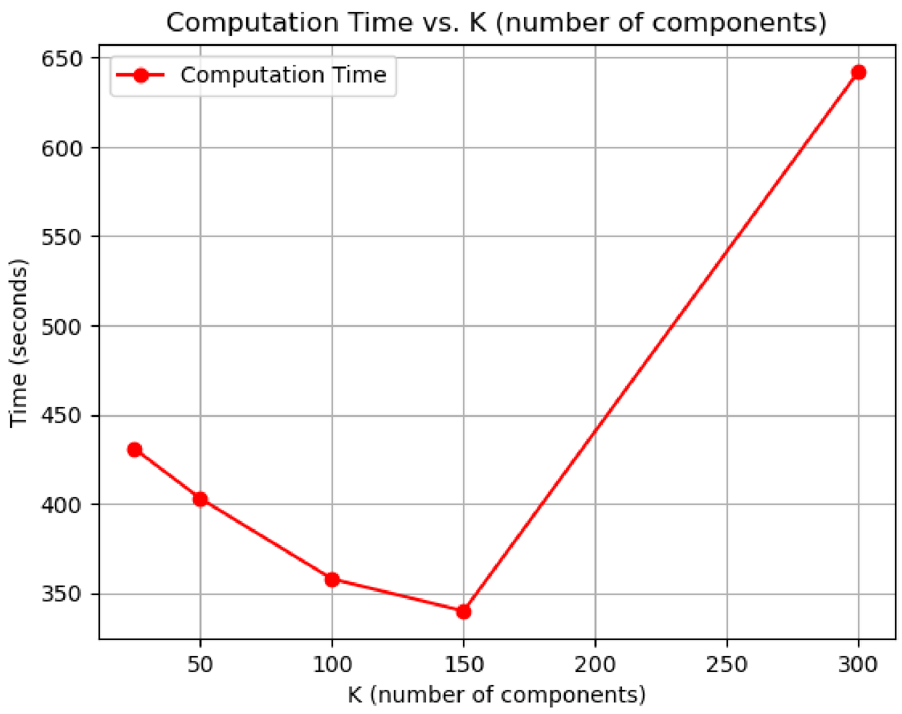

Figure 8: Time taken vs K (number of components)

Surprisingly, using 150 as the number of components proved to be a good choice when it game to computational time as well. Hence, reducing the dimensionalty of the feature space to 150 components significantly improved the final model it terms of both performance and computational efficiency.

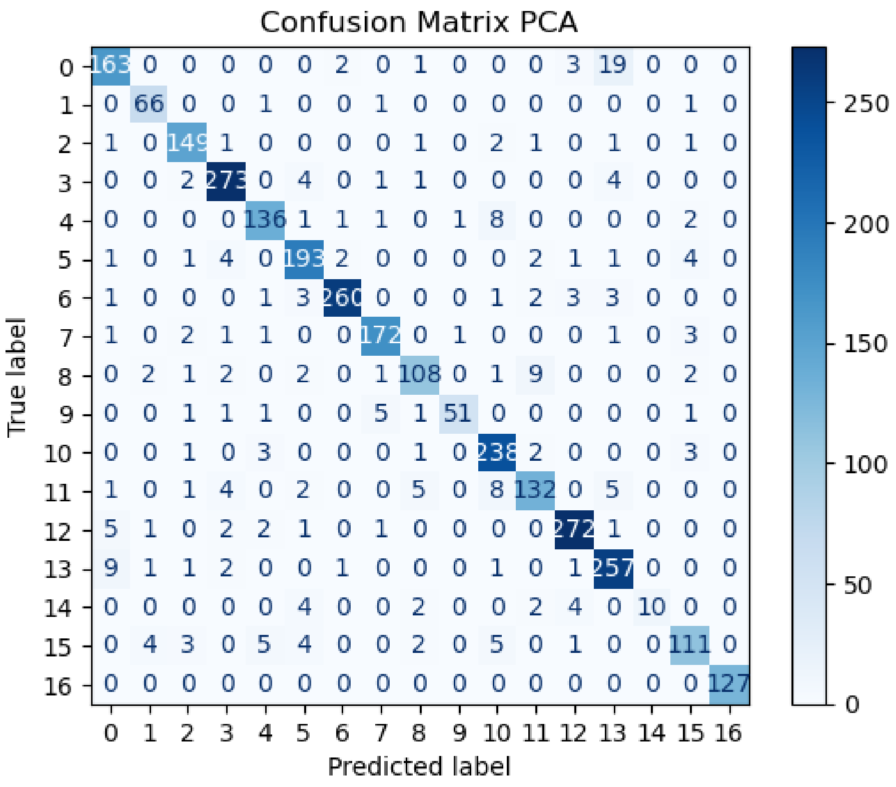

Figure 9: PCA Confusion Matrix of the final model

3.9 How did you find corrupt images in the unlabeled dataset? What was your approach and did it work as expected?

To find the corrupt images in the unlabeled dataset (the corrupt dataset), I tried an approach that involved using more of the available data for training. In addition, I employed a model uncertainty-based approach combined with PCA for dimensionality reduction, using the optimal number of components (best K) determined in the previous task. The idea I had was to train my model on the labeled data, and then apply it to the unlabeled data to detect the OOD images based on the model's uncertainty in its predictions.

I started off by loading the datasets again, such that I could treat this task as a separate task from the others. Then, I pre-processed both the normal dataset and the corrupt dataset, and applied PCA with the best K (150).

After the pre-processing and PCA steps, I trained the best model classifier (SVM) on the full transformed PCA labeled data (e.i, not split into training and test sets), after it had been applied with SMOTE to handle the class imbalance discussed earlier.

When the model was trained to correctly classify the gray-scale images with probabilities, I used it to predict the class probabilities for the unlabeled corrupt dataset.

I did this as such:

```
# Make predictions on the corrupt dataset using the best
    model
best_clf_pca.fit(X_pca, y_smote)
probabilities = best_clf_pca.predict_proba(corrupt_X_pca)
...
```

With these probabilities, I could see how confident the modle was in classifying each image.

To see which of the images in the corrupt dataset was out-of-distribution (OOD), i calculated the uncertainty scores for every image in dataset.

If the uncertainty score is high for a given image, it indicates that the model I trained is less confident about its classification prediction. Hence, the image may be OOD.

I sorted the images based on their given uncertainty score, such that I get the images that were most likely OOD.

Here is how I did this:

```
...
uncertainty_scores = 1 - np.max(probabilities, axis=1)
uncertainty_with_indices = list(enumerate(uncertainty_scores
    ))
uncertanty_sorted = sorted(uncertainty_with_indices, key=
    lambda x: x[1])
indicies_sorted = [index for index, _ in uncertanty_sorted]
corrupted_images_sorted = X_corrupt[indicies_sorted]
```

The results of the images with high uncertainty visualized turned out like this:

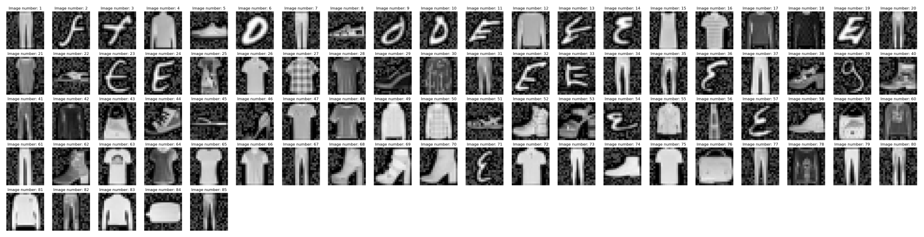

Figure 10: High Uncertainty Images (corrupt dataset)

I counted that I got 68 out of 85 OOD images from the corrupt dataset, which is a number I am very happy about.

3.10 Measures taken to avoid overfitting

In this project I employed several different strategies to avoid overfitting. Such strategies was for example Cross-Validation, hyperparameter tuning, SMOTE and dimensionality reduction with PCA.

The following steps were taken to help prevent overfitting during model development:

- **Cross-Validation (K=5)**: Cross-Validation with 5 as the K helped ensure that the model's performance was evaluated across different subsets of the data. Training the model on different folds, helped me reduce the risk of overfitting to a particular train-test split.
- **Hyperparameter Tuning**: Hyperparameter tuning also helped prevent overfitting by tuning on the regularization parameter C for the SVM, and the number of trees plus max depth for Random Forest. Tuning on such parameters helped control the model's complexity, and prevent the models from becoming too complex and overfit.
- **SMOTE**: SMOTE helped me prevent overfitting to the majority classes by balancing the training set. Hence, the models could learn equally from all of the classes.
- **Principal Component Analysis (PCA)**: PCA being applied to reduce the dimensionality of the data before giving it to the models, helped in removing noise and irrelevant data that may have been present in the dataset. This in turn helped simplify the models, and reducing the overall risk of overfitting.
- **Data Pre-processing**: Lastly, by pre-processing the data such as with StandardScaler, I could ensure that a single feature would not dominate in the learning process of the models (like discussed earlier).

3.11 Given more resources (time or compute), how would you improve your solution?

3.11.1 Model selection and hyperparameter tuning

Unfortunately, I feel like the model selection part of this project could be performed much better if I had been given more time and resources. I would tune many more hyper-parameters, and extended my grid much larger given the chance. Even though I am happy with the results of my current model, it would be great to see its full capabilities when being tuned to a respectable amount of hyperparameters with a large grid.

In addition to this, I would also have liked to experiment with machine learning techniques outside of SKLEARN, for tasks such as finding the OOD images from the corrupt dataset.

3.11.2 The use of pipeline to prevent data leakage

Lastly, I noticed an improvement that could be made upon my code. In my code, I initialize the scaler and SMOTE, and apply them to the data myself. Instead of this, I should've created a pipeline like such:

```
svm_pipeline = Pipeline([
    ("scaler", StandardScaler()),
    ("smote", SMOTE(random_state=seed)),
    ("svm", SVC(random_state=seed))
])
```

With this pipeline, I can make sure that no data leakage takes place. This is because SMOTE is applied only to the training folds in each Cross-Validation, and hence, prevents an over optimistic performance estimate in the evaluation of the models. In my current code without the pipeline, I SMOTE both the validation set and the training set, where the validation set should've not have been SMOTEd. This is an error that I am happy I found out about, and are eager to create a better model for my next machine learning project.