Summary

The summary should give a short, non-technical overview of your project. You should also argue, based on your results, whether the machine learning approach is appropriate for this task and what is your expectation of its performance in real-life.

Problem 1:
The first thing we did was to analyze the dataset which consists of samples of 400-element long lists each representing a 20 x 20 greyscale image where each element is an integer between 0 (black) to 255 (white). Each sample represents an image of a number from 0 to 9 or a letter from A to F, labeled from 0 (0) to 15 (F) and also on top of that, the label 16, which is just a mostly black image. After dividing the dataset into training- and test-sets we had to tamper the training set in order to make it balanced. We then selected SVM and Random Forest as the two classifiers we wanted to try out. We figured that they were strong candidates due to their ability to handle high-dimensional data (like images). What we did was to tune each model on the training data in order to see their best possible performance on this dataset and then choose the best model based on the highest cross-validation accuracy. In our case, the **Support Vector Machine** was the strongest with a 2% higher cross-validation accuracy, and therefore we went for that model. We further tested its performance on test data and were satisfied with the overall score even for the labels that had significantly less occurrences in the original datasets before we oversampled the training data.

Problem 2:
In problem 2 we went further with our SVM model and tested its performance on different dimension reduction using PCA. We compared each dimension reduction on performance and computational efficiency. We found that when reducing the dimensions to 100 it gave the best result. And we chose this as our final classifier.

Problem 3:
For this problem we was given a new dataset that had images as before but now with images that were out of the distribution (clothes) and without labels. Our task was to try and find these images that was out of distribution in a clever way. We tried many ways to make the predictions better, but the idea that gave the best results was using both the training and test data as training data and then use only the corrupt dataset as test data Instead of not including the previous test-set in the training-set.

Results: Our final classifier had an average weighted f1-score of 0.93. Based on this we think that the machine learning approach we chose were appropriate for this project and we think it will perform well in real-life.

Technical report

The technical report should cover what you have actually done and why. It should contain detailed information on your design choices and experimental design. The list below is a guideline for what you should include in your report (but you are free to structure it as you see fit):

Our observations about the dataset.

First we tried to understand the data. To do this we printed out a random sample 5 in this case.
We can see that the label prints out 8 so we expect the image to look like an 8.

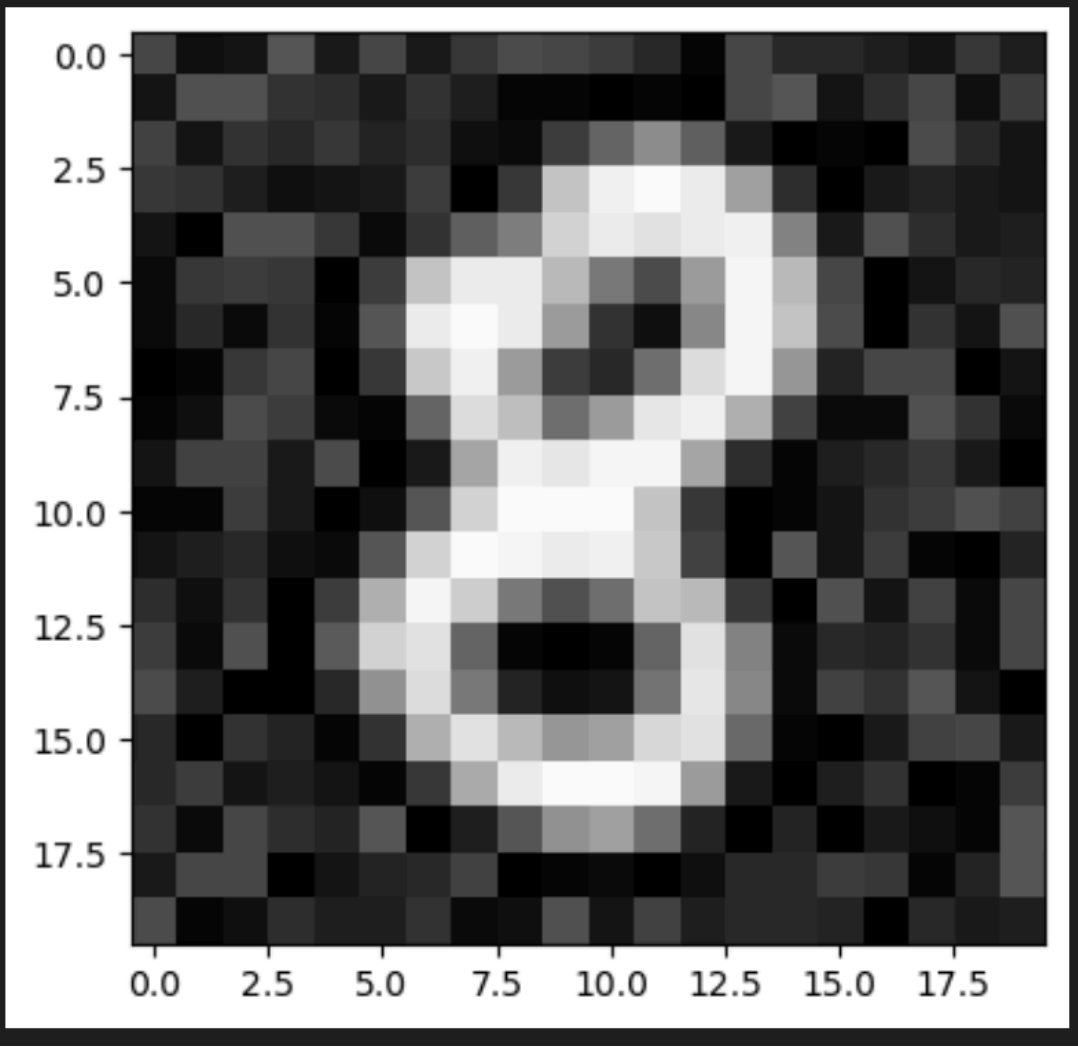
```
# Print a random sample to visualise it
print(y[5])
```

✓ 0.0s

```
8
```

```
# Here we can see more clearly that it is a 8
plt.imshow(X[5].reshape(20,20), vmin=0, vmax=255.0, cmap="gray")
plt.show()
```

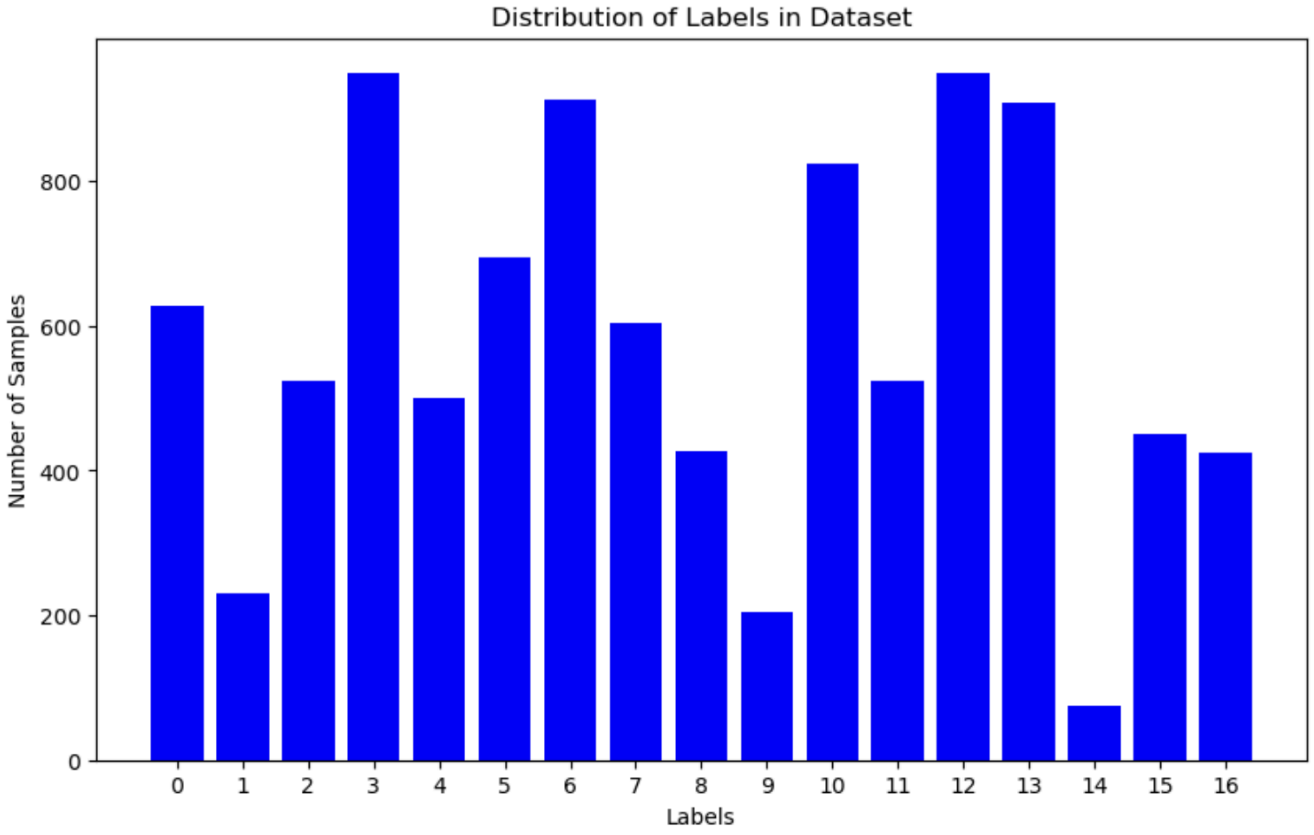
✓ 0.0s



We as humans can clearly see that this is an 8. Our job will be to make the machine see it too. We found this helpful when continuing on the project.

As we can tell on the histogram below, the data is not balanced. This is something we need to consider and we might want to do some pre-processing of the data. We don't know yet but we'll find out what's best. probably something like oversampling or undersampling is the way to go to make the data more balanced. We also know that the **randomforestClassifier** from sklearn has a way to take the imbalance in consideration with class weights. So now we

have a few things to try out.



Class 14 (E) has the lowest count with only 74 occurrences and the highest was class 3 with 950. So clearly there is a big imbalance in the dataset we need to pre-process.

Pre-processing steps (if any).

Weighted classes:
Our first approach was to use random forest with class_weight set to 'balanced'. This way the classifier should weight each class more appropriately so that for example class 14 doesn't get underrepresented when training.

When there is imbalance when training the model it can lead to the model almost never guessing the minority classes, that makes the model perform very badly when guessing on these classes. We did not want this.

To take account for the imbalance we first tried this:

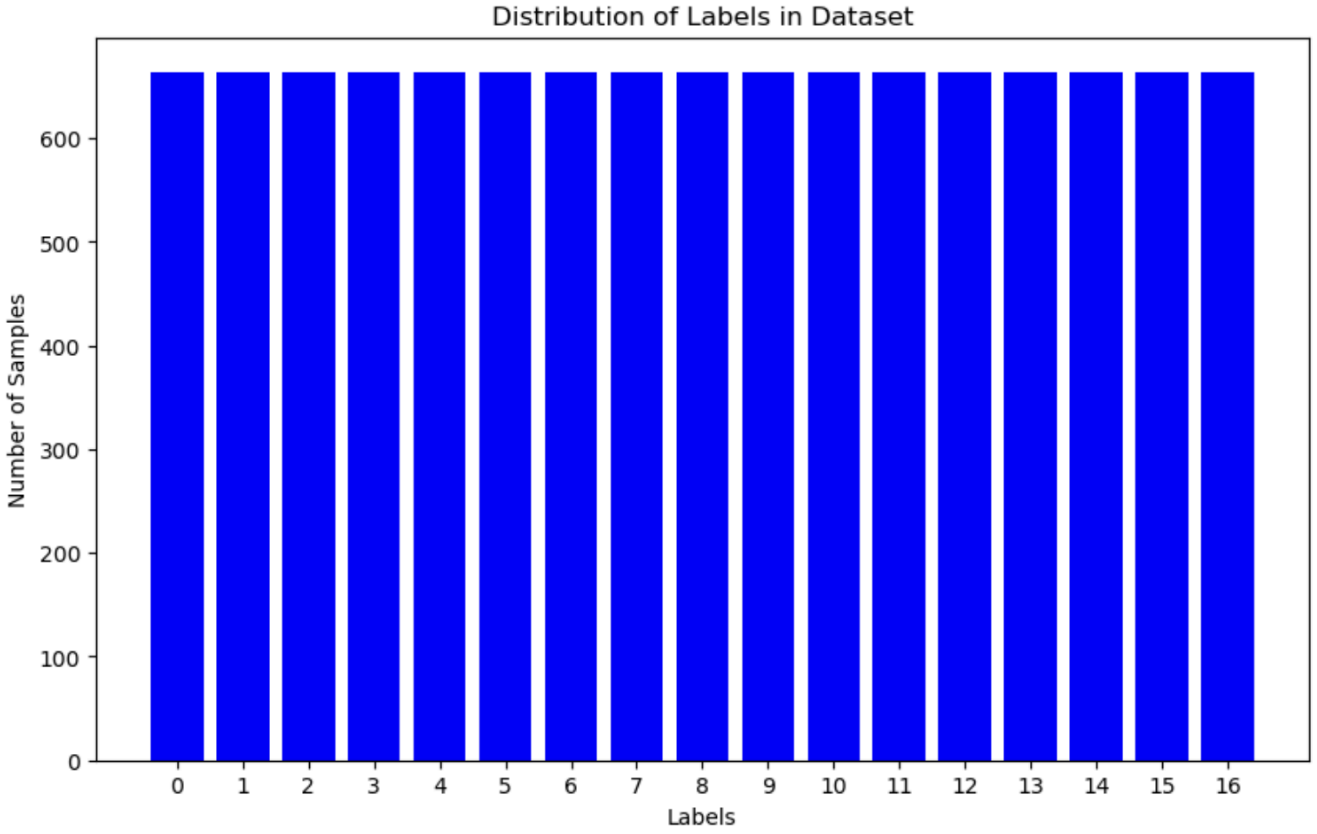
```
rf = RandomForestClassifier(random_state=seed, class_weight='balanced')
```

Oversampling:
But the results was not so impressive, therefore we wanted to try oversampling which gave better results. Oversampling is another way to account for imbalance in datasets. Oversampling creates synthetic data of the minority classes in order to make the amount level with the class that has the highest amount of occurrences and thus makes the data become balanced. We used SMOTE for this:

```
from imblearn.over_sampling import SMOTE
```

```
#Split the data into test and (training and validation)
X_train_val, X_test, y_train_val, y_test = train_test_split(X,y, test_size=0.3, random_state=seed, shuffle=True, stratify=y)
# We use stratify so that the distribution is equal in the test and the training + validation sets.


# We normalize so that whatever classifier we choose all features will contribute equally. (This is not needed for random for
# Normalize the data to range from 0 to 1
X_train_val = X_train_val / 255.0
X_test = X_test / 255.0

# Oversample our data with Smote
smote = SMOTE(random_state=seed)
X_train_val, y_train_val = smote.fit_resample(X_train_val, y_train_val)
# We only oversample the training and validation! Why?
# Because if we oversampled the test-data we would make the test data unrealistic and not represntative of real world data.
```

We split the data into training + validation and test (30% test and 70% training and validation). We normalized the data and then we used SMOTE to oversample our data. Now we can see that the data is well balanced in our training + validation data.



Splitting the data:

We split the data so that we can use the training data to train our model and also keep the test data untouched until we are satisfied with our model and trust it to do predictions on data it has never seen before. we make sure that the tampering only happens with the training data in order to make the test data as realistic as possible.

Normalized the data:

The SVM (Support Vector Machine) classifier is sensitive to feature scales. That means that if one feature ranges from 0 to 1000 and another ranges from 0 to 10, the feature that ranges from 0 to 1000 will have a greater influence since it can have higher values. Since every feature in this specific dataset are all ranging from 0 - 255, we don't necessarily need to normalize. However, normalizing is a good practice when training your model so we did it anyways. It may also improve the performance of the model.

We do not need to normalize when using random forest since it is not sensitive to feature scales, but it doesn't hurt either so we normalize before using either of the classifiers.

Stratify:
We use stratify so that the distribution is equal in the test and the training + validation sets. **Stratification** ensures that all classes are represented proportionally, improving the reliability of our model evaluation and making our test set more representative of the real-world scenario.

Choice of candidate models and hyperparameters. And why were the others omitted?

In our project we chose random forest and SVM.

Random forest

We chose random forest because it is well-suited for both classification and regression tasks. It has the advantage of handling high-dimensional data and preventing overfitting. And personally we were curious to see how well it could perform.

Why these hyperparameters for random forest:

```
param_grid = {
    'n_estimators': [10,15,20,40,100],
    'max_depth': [None, 10,15,20,30],
    'max_features': ['sqrt','log2'], #
    'criterion': ['entropy','gini']
}
```

We chose to tune **n_estimators** (number of trees) with 10, 15, 20, 40 and 100. We thought that a low number of trees may lead to underfitting and a higher one often lead to a better and accurate model. We thought this was a nice balance. We could add higher numbers in n_estimators but we think this will lead to little improvement compared to computational time.

We chose to tune **max_depth** with None, 10, 15, 20 and 30. We wanted to let the trees grow until they are finished (None) which may lead to overfitting. We do not include lower values since we think this will just underfit too much. We thought this was a good and relevant mix of max_depth.

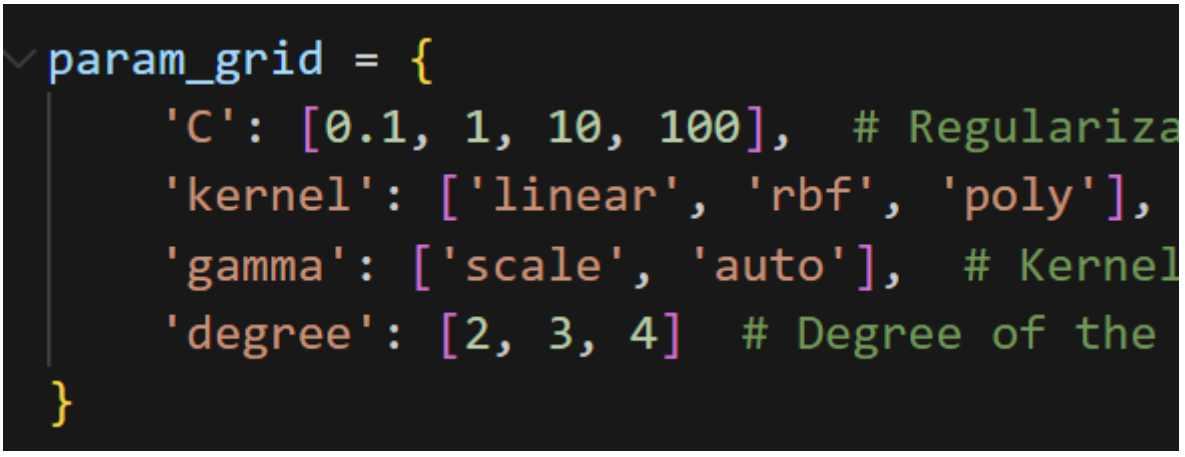
We chose **max_features** with sqrt and log2. We did not include None (consider all features) because this will often overfit. With sqrt and log2 we gave the model the option to choose either, which then may give us a little boost in performance.

We chose **criterion** with entropy and gini. We have entropy and gini because we wanted to determine which works better for our dataset. We do not include other criterions because these we have are relevant for classification.

We did not include other hyperparameters since we didn't want the tuning to take longer time.

SVM

We chose SVM (Support vector machine) since this classifier is suitable for a wide variety of datasets. The classifier focuses on maximizing the margin between classes, which often leads to better generalization in classification tasks which makes it very suitable for classification.



By choosing these hyperparameters to tune we balance generalization and overfitting by tuning C, gamma, and degree, which control regularization and model flexibility. Different kernels, including linear, rbf, and poly, were used to capture both linear and non-linear relationships in the data. We didn't add more hyperparameters to tune since we didn't want the tuning to use too much time.

We chose to tune **C** with 0.1, 1, 10, 100. Here having lower values will potentially underfit and having higher will potentially overfit. So we chose this range to balance this.

We chose to tune **kernel** with linear, rbf and poly. We chose rbf and poly because this is suitable for non-linear data. And we included linear in case the data could be linearly seperable, this may be possible after reducing the dimensions as we do in problem 2.

We chose to tune **gamma** with scale and auto because they automatically adjust based on the dataset, helping the model balance between fitting the data well and generalizing to new data. Scale uses the number of features, and auto uses the number of samples to decide gamma. We skipped fixed numbers like 0.1 or 1 to keep things simple and avoid making the process too slow.

We chose to tune **degree** with 2, 3, and 4 because these values let the model capture complex relationships without overfitting. A degree of 2 gives a simple quadratic boundary,

while 4 allows for more flexibility. We avoided higher degrees since they can lead to overfitting and make the model slower without adding much benefit. We don't include lower numbers for **degree** (like 1) because a degree of 1 is just a straight line, which is too simple to capture complex patterns in the data.

Why were the other classifiers omitted?

Classifiers and why we didn't choose them:

CNN (Convolutional Neural Networks): We wanted to use this but we didn't because of the complexity and unfamiliarity. And we thought that it would be sufficient enough with random forest and SVM.

KNN (K-Nearest Neighbors): Inefficient in high dimensions.

Boosting models: Computationally intensive and not optimized for image data.

Logistic regression: Limited to linear separability, so we didn't think it could perform as well.

Naive Bayes: Naive Bayes assumes feature independence, which is inappropriate for image data.

Chosen performance measure. Justify your choice.

In tuning:
When tuning our two different classifiers, we used accuracy_score and chose the parameters based on which fold that got the best average accuracy score. We used accuracy_score since we have oversampled the data and now it is not imbalanced anymore.

Here is example of where we use this. This is for when tuning random forest:

```
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           scoring='accuracy',
                           cv=5, n_jobs=-1, verbose=0)

grid_search.fit(X_train_val, y_train_val)


print("Best Hyperparameters:", grid_search.best_params_)
print("Best Cross-Validation accuracy:", grid_search.best_score_)

#Here we can use accruacy when tuning since we removed the imbalance
```

Here it will print out the best cross-validation accuracy and with this we choose our parameters.

In testing performance:
After tuning in problem 1 we then test the model that got the best cross-validation accuracy

score. In our case that was SVM.

To further test SVM performance on unseen test data we did the following:

```
# Evaluate the best model on the test set
best_svm = grid_search.best_estimator_

# Here we use classification report so that we can see how well
print(classification_report(y_test, best_svm.predict(X_test)))
```

We chose to evaluate SVM's performance on test data with classification_report since this gives us a good look at each class's f1-score. Here is a example of this:

```
SVM:
              precision    recall  f1-score   support

           0       0.90      0.88      0.89       188
           1       0.89      0.96      0.92        69
           2       0.91      0.95      0.93       157
           3       0.94      0.95      0.95       285
           4       0.89      0.89      0.89       150
           5       0.90      0.93      0.92       209
           6       0.97      0.95      0.96       274
           7       0.95      0.95      0.95       182
           8       0.90      0.88      0.89       128
           9       0.93      0.89      0.91        61
          10       0.91      0.95      0.93       248
          11       0.91      0.79      0.85       158
          12       0.96      0.95      0.96       285
          13       0.89      0.93      0.91       273
          14       0.80      0.55      0.65        22
          15       0.84      0.83      0.84       135
          16       1.00      1.00      1.00       127

    accuracy                           0.92      2951
   macro avg       0.91      0.90      0.90      2951
weighted avg       0.92      0.92      0.92      2951
```

We can for example see that class 14 has the lowest f1-score of 0.65. And the others are pretty high. We also see that class 16 has f1-score of 1.00 this is very good, that means that we achieved what was asked of us in the problem 1 description. In problem 1 it is written "Your system must be capable of detecting these empty labels, allowing the CEO of the production department to be notified and take necessary actions." we can say that our model can with very high confidence find the empty images.

Model selection scheme(s) that you used. Justify your choices

The model selection scheme we chose involved **hyperparameter tuning with cross-validation**, using **accuracy** as the evaluation metric. Hyperparameter tuning was performed using **Grid Search**, where various combinations of hyperparameters were tested to find the optimal configuration.

We used cross-validation because this will helps the model generalize and reduces the risk of overfitting.

Here is a example of this for SVM:

```
svm = SVC(random_state = seed)

param_grid = {
    'C': [0.1, 1, 10, 100],  # Regularization parameter
    'kernel': ['linear', 'rbf', 'poly'],  # Kernel type to be used in the algorithm
    'gamma': ['scale', 'auto'],  # Kernel coefficient
    'degree': [2, 3, 4]  # Degree of the polynomial kernel function ('poly')
}

grid_search = GridSearchCV(estimator=svm, param_grid=param_grid,
                           scoring='accuracy',
                           cv=5, n_jobs=-1, verbose=2)
```

Here the training + validation data was split into 5 folds, where the model was trained on 4 folds and validated on the remaining fold, repeating this process across all folds and averaging the accuracy scores.

We used accuracy since it is appropriate on our data since we balanced the data with oversampling.

We did the same procedure with random forest but with other hyperparameters.

What is your final classifier, and how does it work? Justify why you think it is the best choice.

We will walk through how we found our final classifier.

Firstly the comparison between the two models in problem 1:

Our **random forest** ended up with the best cross-validation accuracy of ~0.95 with criterion set to entropy, max_depth set to 20, max_features set to sqrt and n_estimators set to 100. Also it took random forest ~4m and 47seconds to tune this.

```
✓ 4m 46.5s

Best Hyperparameters: {'criterion': 'entropy', 'max_depth': 20, 'max_features': 'sqrt', 'n_estimators': 100}
Best Cross-Validation accuracy: 0.9473684210526315
```

Here is the what we tuned:

```
# Defining the hyperparameters we want
param_grid = {
    'n_estimators': [10,15,20,40,100],
    'max_depth': [None, 10,15,20,30],
    'max_features': ['sqrt','log2'], #
    'criterion': ['entropy','gini'] #
}
```

Our **SVM** ended up with the best cross-validation accuracy of ~0.97 with C set to 10, degree set to 2, gamma set to scale and kernel set to rbf. It took SVM ~13min to tune this.

```
✓ 13m 5.2s

Best Hyperparameters: {'C': 10, 'degree': 2, 'gamma': 'scale', 'kernel': 'rbf'}
Best Cross-Validation accuracy: 0.9701017249004865
```

Here is what we tuned:

```
param_grid = {
    'C': [0.1, 1, 10, 100],  # Regulariz
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto'],  # Kerne
    'degree': [2, 3, 4]  # Degree of the
}
```

We see that SVM uses more time but has a higher cross-validation accuracy then random forest. Hence we chose to go further with SVM.

Now comparing our original to the best result in PCA part:

Our original SVM got cross-validation accuracy of ~0.97. And an average weighted f1-score of 0.92 on testdata.

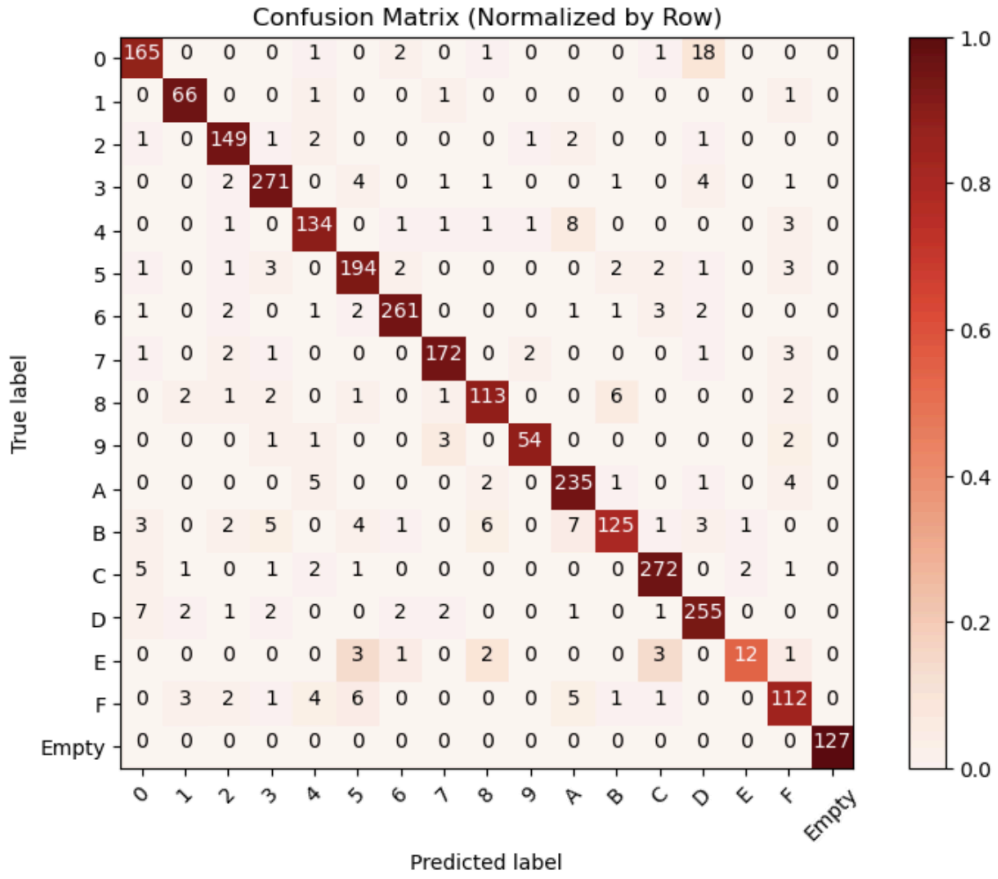
Here is the f1-score on testdata:

SVM:

	precision	recall	f1-score	support
0	0.90	0.88	0.89	188
1	0.89	0.96	0.92	69
2	0.91	0.95	0.93	157
3	0.94	0.95	0.95	285
4	0.89	0.89	0.89	150
5	0.90	0.93	0.92	209
6	0.97	0.95	0.96	274
7	0.95	0.95	0.95	182
8	0.90	0.88	0.89	128
9	0.93	0.89	0.91	61
10	0.91	0.95	0.93	248
11	0.91	0.79	0.85	158
12	0.96	0.95	0.96	285
13	0.89	0.93	0.91	273
14	0.80	0.55	0.65	22
15	0.84	0.83	0.84	135
16	1.00	1.00	1.00	127
accuracy			0.92	2951
macro avg	0.91	0.90	0.90	2951
weighted avg	0.92	0.92	0.92	2951

We see that it performs quite badly on class 14 with a f1-score of 0.65.

Here is a confusion matrix of these results:



This confusion matrix shows when it misses and what it misses with. The color gets darker as the point in the matrix gets bigger in relation to itself. If you look at the Empty class where it is predicted correctly we see that it recognizes this all 127 of 127 times, so that is why it is a dark red. If we look at class E where it correctly predicted we see that it has a much lighter red indicating that it does not correctly find all of the E's all the time. This is a nice way of looking at what the model misses with.

Now for the best PCA SVM model. We got the best result when n_components were 100.

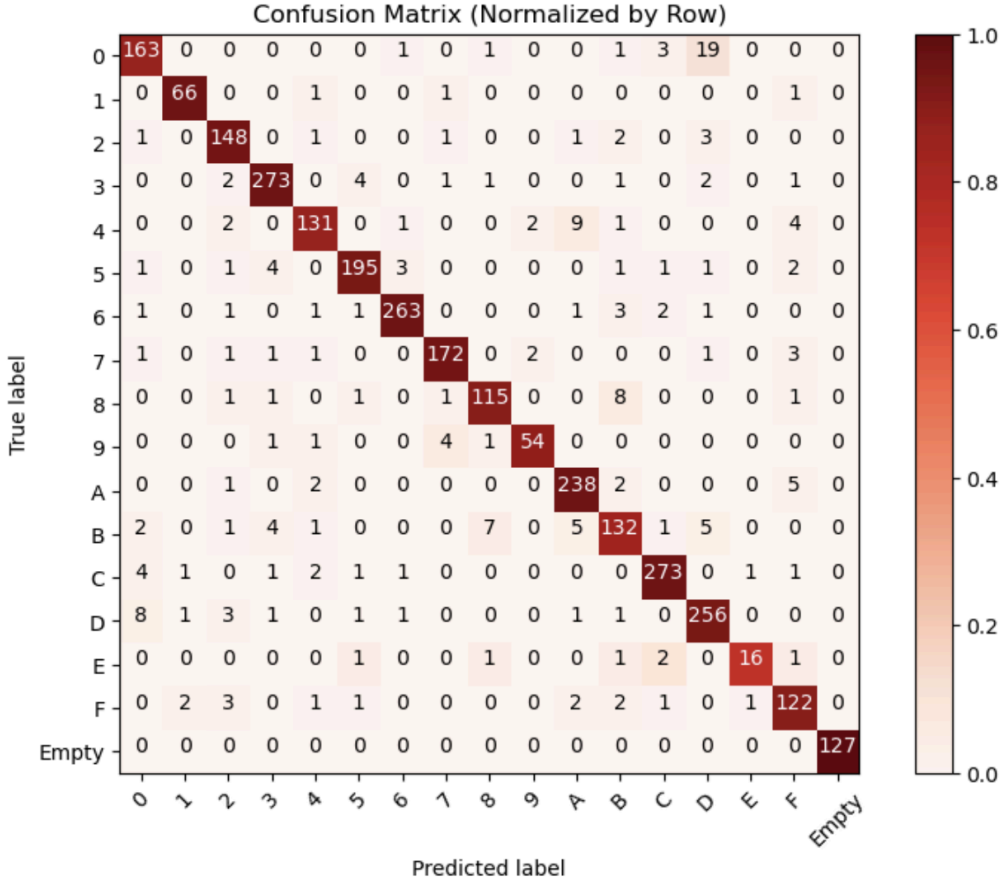
```
n_components: 100
total_variance_explained: 0.925265560636292
best_hyperparameters: {'C': 100, 'degree': 3, 'gamma': 'scale', 'kernel': 'poly'}
best_cv_accuracy: 0.9737284387439187
execution_time: 42.65602254867554
```

This had ~0.93 variance in the data and got the best cross-validation accuracy when C was set to 100, degree set to 3, gamma set to scale and kernel set to poly. It also had a execution time ~43 seconds (This is extra reduced because we tune on less hyperparameters than in problem 1).'

Here is the f1-score for this model on testdata:

	precision	recall	f1-score	support
0	0.90	0.87	0.88	188
1	0.94	0.96	0.95	69
2	0.90	0.94	0.92	157
3	0.95	0.96	0.96	285
4	0.92	0.87	0.90	150
5	0.95	0.93	0.94	209
6	0.97	0.96	0.97	274
7	0.96	0.95	0.95	182
8	0.91	0.90	0.91	128
9	0.93	0.89	0.91	61
10	0.93	0.96	0.94	248
11	0.85	0.84	0.84	158
12	0.96	0.96	0.96	285
13	0.89	0.94	0.91	273
14	0.89	0.73	0.80	22
15	0.87	0.90	0.88	135
16	1.00	1.00	1.00	127
accuracy			0.93	2951
macro avg	0.93	0.91	0.92	2951
weighted avg	0.93	0.93	0.93	2951

Here is the confusion matrix:



Comparison and choice:
The PCA one has a little better cross-validation accuracy then our original SVM. The execution time is dramatically reduced from ~13min to ~43seconds.

Looking at their f1-scores the PCA one has a better weighted average f1-score of 0.93 compared to 0.92. We also see that it performs better on class 14 with a f1-score of 0.80 compared to 0.65. Can also see this in the comparison of the two confusion matrices.

By these observations we chose SVM with dimensions reduced to 100 through PCA. Very practical that it seems we don't lose any performance by reducing the dimension this much.

We didn't consider any other values for n_components because they all had worse cross-validation accuracy then when n_components = 100.

□ How well it is expected to perform in production (on new data). Justify your estimate.

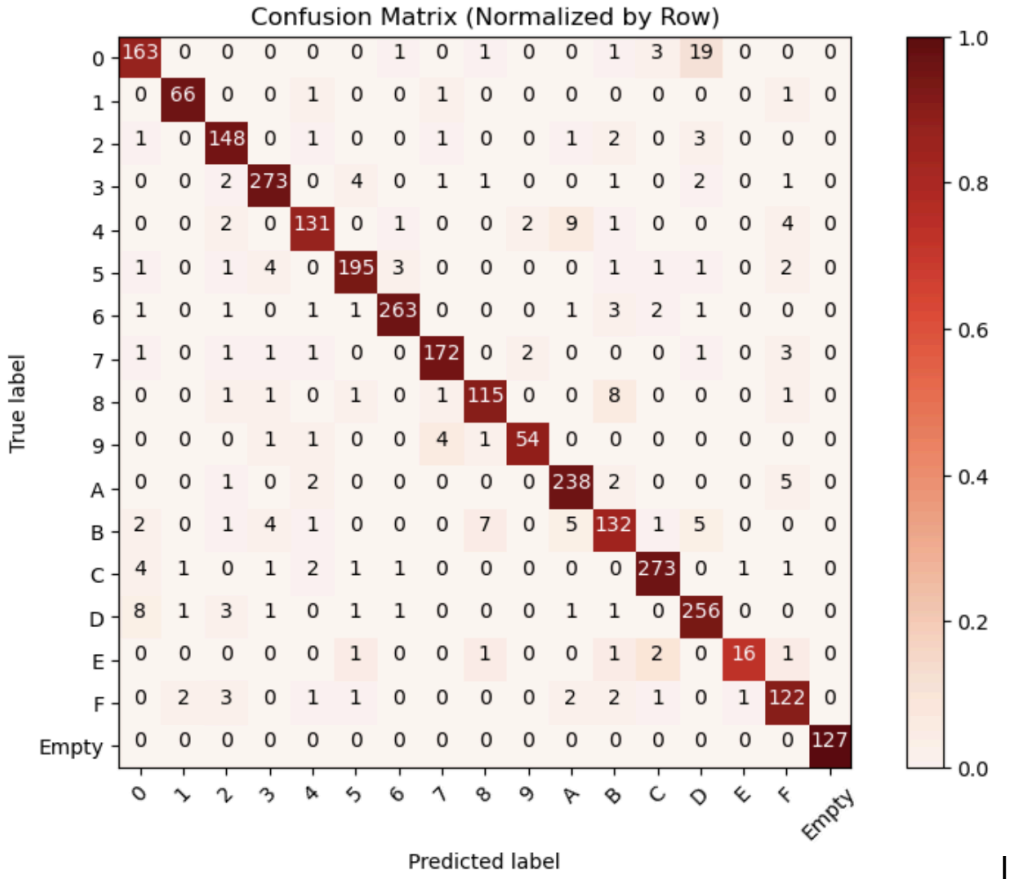
We interpret that problem 2 is separate from problem 1 in the sense that we could use the test-data again in problem 2 even though we used it in problem 1.

Our final classifier was SVM with dimensions reduced to 100 through PCA. Let us take a look at how well we expect it to perform on new data.

We can look at the f1-scores and the confusion matrix for this.

	precision	recall	f1-score	support
0	0.90	0.87	0.88	188
1	0.94	0.96	0.95	69
2	0.90	0.94	0.92	157
3	0.95	0.96	0.96	285
4	0.92	0.87	0.90	150
5	0.95	0.93	0.94	209
6	0.97	0.96	0.97	274
7	0.96	0.95	0.95	182
8	0.91	0.90	0.91	128
9	0.93	0.89	0.91	61
10	0.93	0.96	0.94	248
11	0.85	0.84	0.84	158
12	0.96	0.96	0.96	285
13	0.89	0.94	0.91	273
14	0.89	0.73	0.80	22
15	0.87	0.90	0.88	135
16	1.00	1.00	1.00	127
accuracy			0.93	2951
macro avg	0.93	0.91	0.92	2951
weighted avg	0.93	0.93	0.93	2951

We see that the model is worst at identifying class 14 (E). But it still has a pretty high f1-score of 0.80. And with a average weighted f1-score of 0.93 we think that the model will perform strongly on new data. But it can depend, if the data we trained on and tested on isn't representative of real world (new data) then our model may perform worse.



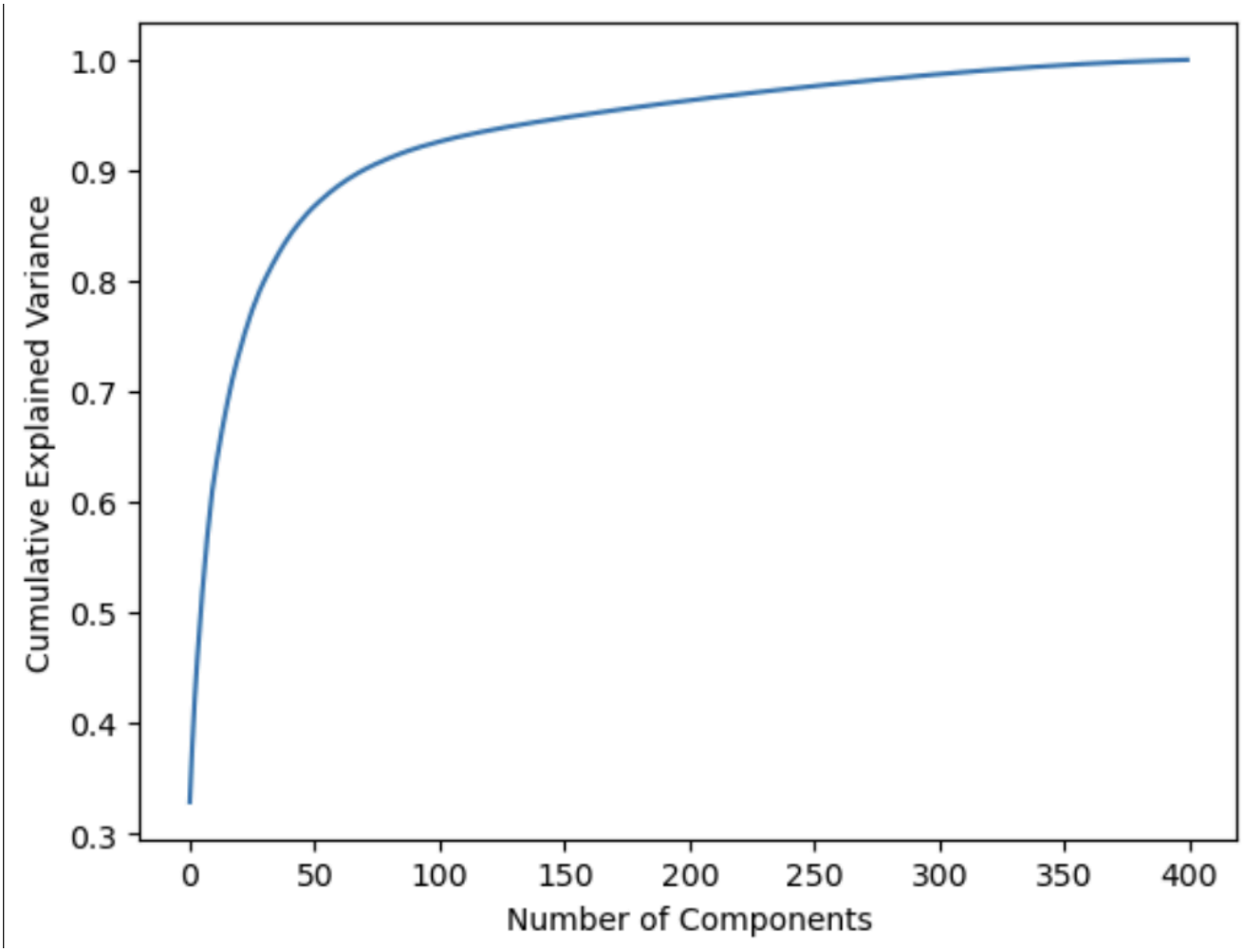
In the confusion matrix we see that our model will sometimes think that images of zeros are the letter D. This will likely also be reflected on with new real life data. We as humans can also relate to this example.

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

We tested reducing the dimensions to 10, 20, 50, 100 and 200. We wanted to test it with these values since this gives us insight of how well the model performs when the dimensions are reduced to a much lower dimension than it previous was (10), to just reducing the dimensions to half of the original (200). Here we see a graph of the variance of the data compared to the reduced dimensions.

We see that from 400 to 100 it doesn't lose that much variance. But it begins to dip around 50.

The **explained variance** in PCA measures how much information from the original dataset is captured by each principal component. So the task at hand is finding a sweet spot for when we reduce computational time at the same time as we don't lose too much information that it affects the performance.



We should mention that when tuning this we had fewer hyper parameters than in problem 1, we did this because of time restraint. Keep in mind that this will make the computational efficiency (execution time) even better than only reducing the dimensions. Here are the hyperparameters we tuned:

```
# Define the hyperparameters to tune
param_grid = {
    'C': [1, 10, 100],  # Regularization parameter
    'kernel': ['rbf', 'poly'],  # Kernel type to be used in the algorithm
    'gamma': ['scale', 'auto'],  # Kernel coefficient
    'degree': [2,3]  # Degree of the polynomial kernel function ('poly')
}
# Here we tune less parameters because we don't want it to take too long of time
```

Components set to 10

Performance and computational efficiency when number of components is set to 10:

```
n_components: 10
total_variance_explained: 0.6096659625927977
best_hyperparameters: {'C': 100, 'degree': 2, 'gamma': 'auto', 'kernel': 'rbf'}
best_cv_accuracy: 0.9172932330827068
execution_time: 15.35850191116333
```

Here we see that the variance is quite low at ~0.61 so we expect it to decrease the performance.

Execution time of ~15 seconds which is a lot faster then ~13min compared to the original's time.

For performance we have the best cross-validation accuracy at ~0.92 which is a lot better than expected when variance is as low as ~0.61.

Components set to 20

Performance and computational efficiency when number of components is set to 20:

```
n_components: 20
total_variance_explained: 0.7282308097030559
best_hyperparameters: {'C': 100, 'degree': 2, 'gamma': 'auto', 'kernel': 'rbf'}
best_cv_accuracy: 0.9576293675364882
execution_time: 13.467512845993042
```

Here we see that the variance is also lower at ~0.73.

Execution time of ~13 seconds which is a lot faster then ~13min compared to the original's time. Weird and peculiar to us that the execution time is faster here than with n_components set to 10 but it is not much.

For performance we have best cross-validation accuracy at ~0.96 which is a lot better than expected when variance is as low as ~0.73.

Components set to 50

Performance and computational efficiency when number of components is set to 50:

```
n_components: 50
total_variance_explained: 0.8653034195744087
best_hyperparameters: {'C': 100, 'degree': 2, 'gamma': 'scale', 'kernel': 'rbf'}
best_cv_accuracy: 0.9707209199469261
execution_time: 19.13480257987976
```

Here we see that the variance is increased to ~0.87.

Execution time of ~19 seconds which is a lot faster then ~13min compared to the original's time.

For performance we have the best cross-validation accuracy at ~0.97. We expected higher accuracy then ~0.97 because we had higher variance.

Components set to 100

Performance and computational efficiency when number of components is set to 100:

```
n_components: 100
total_variance_explained: 0.925265560636292
best_hyperparameters: {'C': 100, 'degree': 3, 'gamma': 'scale', 'kernel': 'poly'}
best_cv_accuracy: 0.9737284387439187
execution_time: 41.122032165527344
```

Here we see that the variance is increased to ~0.93.

Execution time of ~41 seconds which is a lot faster then ~13min compared to the original's time.

For performance we have the best cross-validation accuracy at ~0.97. A little increase in cross-validation in comparison to n_components = 50.

Final thoughts:
We were surprised by how good the cross validation accuracy was with such low numbers. We think that n_components between 50 to 150 is a good place to be in for this dataset. This way we can reduce the dimensions without losing too much performance.

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

Before we tried any approaches we normalized the data. And we used our final classifier for this task which uses PCA with dimension reduction to 100.

We tried some different approaches. We first thought that the best approach was to use predict.proba on the dataset and then sort the images by most uncertain to least uncertain. Predict.proba gives us a list for each image, in this list are the probabilities for each class it thinks the image is. We decided that most uncertain meant that it had a low difference between the highest and the second highest probability. In practice it means that our model is very unsure between two classes the image belongs to.

```
# Loading the previous dataset with labels
dataset = np.load("dataset.npz")
oldX, oldy = dataset["X"], dataset["y"]

# Normalize the old data
oldX = oldX / 255.0

# Oversample the old data with Smote
smote = SMOTE(random_state=seed)
oldX, oldy = smote.fit_resample(oldX, oldy)

# We want to use SVM with PCA set to 100 and with the following parameters, because this gave
svm = SVC(probability=True, random_state=seed, C=100, degree=3, gamma='scale', kernel='poly')

# Performing PCA with n_comp = 100
pca = PCA(n_components=100, random_state=seed)
oldXpca = pca.fit_transform(oldX) # PCA on old data
X_corrupt_pca = pca.transform(X) # PCA on unlabeled data

# Fit our classifier on old data
svm.fit(oldXpca, oldy)

# Then we predict the unlabeled data
predicts_proba = svm.predict_proba(X_corrupt_pca)
```

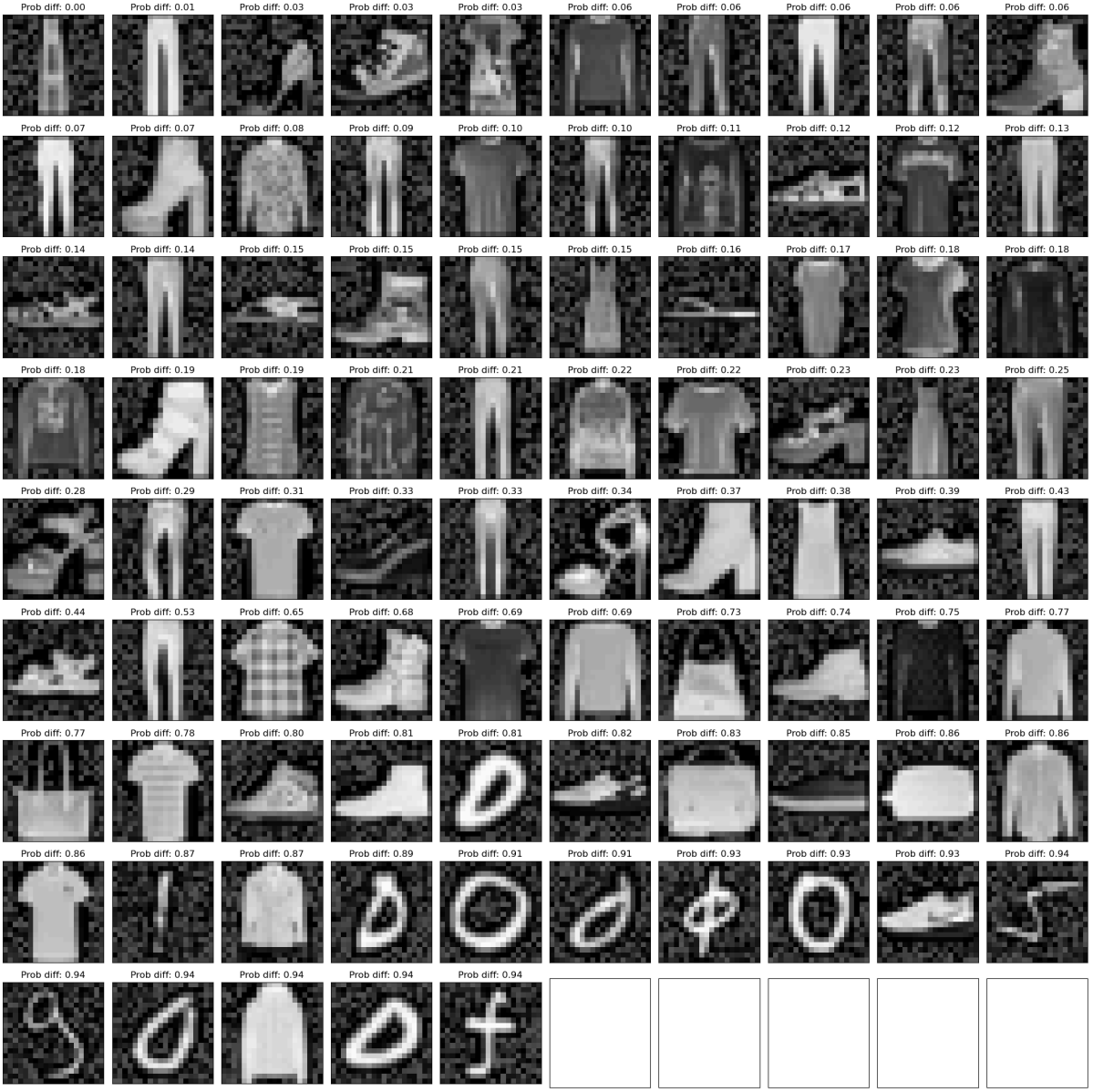
To predict on our new unlabelled data we need the old data with labels. Here we used all of the old data, since we thought that this would get better results. We normalized the old data,

oversampled it and then fit our final classifier on this data. Then lastly we predict the unlabelled data.

Here is how we coded approach 1 difference in top probability:

```
sorted_probs = np.sort(predicts_proba, axis=1)  # Sort the probabilit
prob_diffs = sorted_probs[:, -1] - sorted_probs[:, -2]
# Here we get the two highest probabilities and compute the differenc

# Sort by uncertainty (smallest difference = most uncertain)
most_uncertain_indices = np.argsort(prob_diffs)  # Sorted from smalle

# Plot the most uncertain images, from highest uncertainty to lowest
num_uncertain = 85  # Since there is 85 unlabeled images
images_per_row = 10  # Number of images per row in the plot
num_rows = 9 # Number of rows we need

fig, axes = plt.subplots(num_rows, images_per_row, figsize=(20, 20))

for i, ax in enumerate(axes.flat): # loop through the subplot's axes
    if i < num_uncertain:
        image_idx = most_uncertain_indices[i]  # Start with the most
        image = X[image_idx].reshape(20, 20)  # Get the image
        ax.imshow(image, vmin=0, vmax=1, cmap="gray") # Add the image
        ax.set_title(f"Prob diff: {prob_diffs[image_idx]:.2f}") # Set
    ax.set_xticks([])  # Remove x-axis ticks
    ax.set_yticks([])  # Remove y-axis ticks

plt.tight_layout()
plt.show()
```

This plots the first 85 images with the lowest diff between the top two differences. Here is the plot:



We counted and we counted 75/85 out of distribution images.

Further on approach 2 we tried to find the images by taking the entropy on each image's probabilities. Here is the code for that:

```
from scipy.stats import entropy


entropies = np.apply_along_axis(entropy, 1, predicts_proba)  # Appl

most_uncertain_indices = np.argsort(entropies)[::-1]  # Sort from h

num_uncertain = 85 # Number of most uncertain images to display
images_per_row = 10  # Number of images per row in the plot
num_rows = 9 # Number of rows needed

fig, axes = plt.subplots(num_rows, images_per_row, figsize=(20,20))

for i, ax in enumerate(axes.flat): # Loop through the subplot's axe
    if i < num_uncertain:
        image_idx = most_uncertain_indices[i]
        image = X[image_idx].reshape(20, 20)  # Get the image
        ax.imshow(image, vmin=0, vmax=1, cmap="gray") # Add image t
        ax.set_title(f"Entropy: {entropies[image_idx]:.2f}") # Set
    ax.set_xticks([])  # Remove x-axis ticks
    ax.set_yticks([])  # Remove y-axis ticks

plt.tight_layout()
plt.show()
```

And here is the plot:

Here we got 72 out of 85. We thought that maybe entropy would give us a better result but it did not.

Final results: We found 75 unlabelled images in the first 85 images by using the first mentioned approach. We are quite happy with this result.

We also tried to binarize the images to see if that improved our results, but it did not.

Measures taken to avoid overfitting.

K-Fold Cross-Validation: Instead of using a single train-test split, we used cross-validation with multiple folds (5 folds) to train and evaluate the model on different portions of the data. This helps us reduce the risk of overfitting by ensuring that the model performs well across various portions of the data.

Hyperparameter Tuning: We employed Grid Search to find the best combination of hyperparameters. Combined with cross-validation, this ensures that the model doesn't overfit by being over-optimized for one specific parameter set.

Reduce Dimensionality: In problem 2 used Principal Component Analysis (PCA) to reduce the dimensionality of the data. This technique retains most of the variance, which allows the model to focus on the most important aspects of the data while reducing the risk of overfitting.

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

If we had more time we would want to try CNN and to tune with more values. Since CNN are usually good at image classification we think that it could give us better results.

We would also want to tune with more values and to tune with the same amount of hyperparameters in problem 2 as we did in problem 1.