Predicting digits on images with Random Forest Classifier

Filip Holmberg

EC Utbildning

Kunskapskontroll 2, Machine Learning

202403

Abstract

In the work for the current report a Streamlit application was build capable of making live stream predictions of handwritten images, using the web camera and a Random Forest Classifier model. The model was set to default hyperparameters and trained on a set containing (A) 784 features and (B) 56 custom features. As expected, the model with most features (A) had sufficient accuracy (97 %) and was chosen for the Streamlit application. It was an unexpected result that the model reaches 91 % accuracy on the other set (B), which correspond to only 7 % of the amount in used in set A. This result emphasize that feature selection is an important aspect in ML projects.

Table of contents

1 Introduction

In recent years artificial intelligence has received increasing attention and is believed to play more important role in our day-to-day lives, in near future (Haenlein, 2019). The impact of this technology has been considered more profound than the invention of electricity and fire (Sundar Pichai, 2016). A central concept in AI is machine learning (ML). This concept has been described as follows,

"Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed" (Samuel, 1959).

In practice a ML model is trained on a set of data and deployed in production. The aim is to solve specific tasks, such as forecasting weather conditions or stopping spam to enter the email box. To successfully solve these tasks, the data scientist developing the model must consider many aspects and make design choices. To name a few, aspects can be exploration of data, feature selection, suitability of models. Design choices involve judging what is reasonable for the task at hand. Perfection is rarely coupled with efficiency in ML projects, and experience in consideration of aspects and design choices become important. Each ML task has its own eases and challenges which may not be obvious from start.

To meet the future increasing needs of developing AI (and ML) systems it is required that those working with these technologies are ready for the tasks at hand. Therefore, data scientists need to prepare themselves and keep building experience, for the sake of satisfying future demands.

The current report considers the MNIST dataset (LeCun, 2024), which is a collection of handwritten digits with corresponding true values (e.g. labels) often used for practicing ML. Trial experiments showed that many classifiers (Random Forest, K-Nearest Neighbors, SVC) could make good models (> 90% accuracy), even with default parameters. However, we noticed that some classifiers took long time (2-3h) to model. It was therefore decided to explore the aspect of feature selection, to tackle this time intense issue. In order to build experience of setting a model in production (with associated issues), the final model would also be incorporated into a Streamlit application.

The main purpose of this report is to create a Streamlit application capable of predicting handwritten digits. To achieve this objective the following list of tasks and research questions are relevant,

1. Develop a Random Forest model that achieves an accuracy > 95 %.
2. With fewer and customized features, is it possible to achieve the same accuracy as in (1) using the same model and within a margin of 5 %?
3. Can handwritten and photographed digits be predicted accurately (> 90 %)?
4. Identify and create necessary preprocessing steps to make "live stream predictions" feasible.
5. Construction of a Streamlit application with appropriate functionalities, allowing the user to submit images taken with live stream web camera.

2 Theory

2.1 Data Exploration

The MNIST data set contain 70000 handwritten digits with 784 features with a corresponding label. These features describe the intensity of pixels in a 28 x 28 image (see Figure 1). The following list describe important preprocessing characteristics:

- Pixel intensity ranges from 0-255, where the background is 0.
- The position of the image has been centered.
- Aspect ratio of digits have been preserved.
- The distribution of dead space (background) above/below the digit.

Preprocessing and formatting are generally not necessary when training models on this data set. However, preprocessing custom images according to some of the above-mentioned characteristics may be important. Alternatively, one can transform both the MNIST data set and custom images through a customized preprocessing pipeline. All in all, the custom images need to be alike the training set, for the model to make prediction optimally.

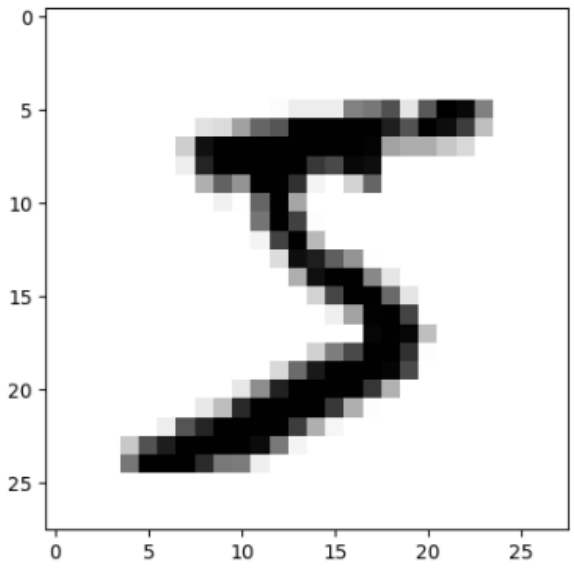

Figure 1: Plot of first image in the MNIST dataset.

2.2 Additional features deduced from MNIST dataset (i.e. custom features)

The practical interpretation of the MNIST dataset is a 2-dimensional image. This is the common way of portraying digits. However, these numbers can also be represented in other dimensions. To clarify the difference of such perspectives, Figure 2 show the same number “4” portrayed from the “in front”, “above” and “side” view. The above and side views are simply the sum of column and row values (respectively) from the in front view. In this manner a total of 56 features (28 “above” plus 28 “side”) can be deduced from the original 784 features found in the MNIST dataset.

The following two aspects of these additional features are worth pointing out:

- Handwritten digits are not like their digital equivalents. For example, a digital “5” would not be distinguishable to digital “2” using both the above and side view, but handwritten digits perhaps would be.
- The 56 features would correspond to only 7 % in amount of the original 784 features. This should surely decrease time spent to train and test models.
- It is a tedious task to predict if these 56 features are better than the original features, if even useful at all. This task is better suited for a machine learning algorithm.

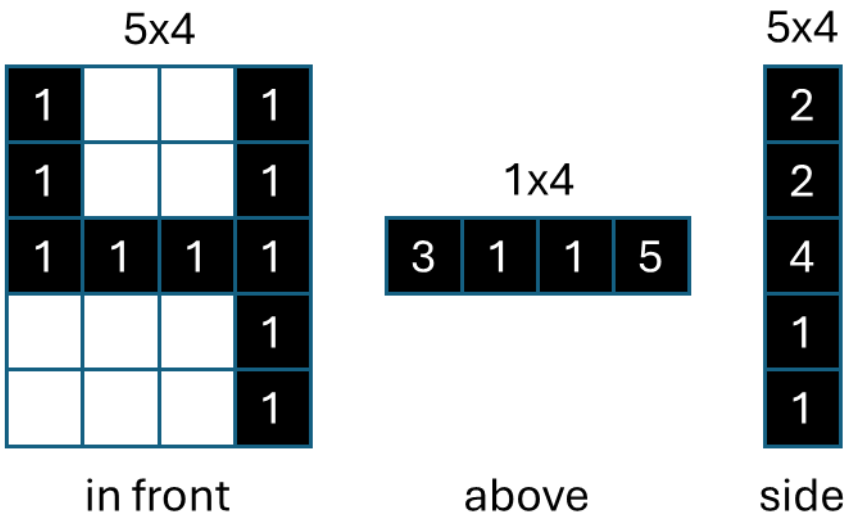

Figure 2: Number "4" schematically shown "in front" (left), "above" (center) and "side" (right).

2.3 Confusion Matrix and Accuracy

The confusion matrix is useful for evaluating classification models (see Figure 3). From this matrix one can deduce metrics for evaluating a model, such as the accuracy (Ajay Kulkarni, 2020):

$$Accuracy = \frac{TN+TP}{TN+TP+FN+FP} \qquad \text{(eq. 1)}$$

Benefits of using the accuracy as metric for evaluation is that it easy to understand and gives an overall idea how correct the model is.

		Predicted	
		Negative	Positive
Actual	Negative	True Negative (TN)	False Negative (FN)
	Positive	False Positive (FP)	True Positive (TP)

Figure 3: Schematic representation of a confusion matrix.

2.4 Gini impurity

The Gini impurity describes the purity of a class in a node (Géron, 2019),

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \qquad \text{(eq. 2)}$$

Gini impurity ranges from 0-1, where 0 is the purest state.

2.5 Classification and Regression Tree (CART) for growing Decision Trees

The Classification and Regression Tree (CART) algorithm is used by Scikit-Learn for constructing (e.g. "growing") a Decision Tree (Géron, 2019). This algorithm works by splitting the training set from top-bottom. The goal is to produce as pure pairs of subsets as possible, from the training set. This results in binary trees (two children per node).

To find optimal split the CART uses a cost function,

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right} \qquad \text{(eq. 3)}$$

From Eq. 3 it can be seen that the split depends on the purity ($G_{right/left\ subset}$ from eq. 2), number of instances (m). Feature (k) and threshold value (t_k) are found at which purest subsets of data are produced. That is to say, for a single feature and threshold value, a boundary is found where subsets are as pure as possible.

2.6 K-fold cross validation

K-fold cross validation is a method for splitting the training set into a subset of train and validation set (see Figure 4). The original training data is randomized and split according to the k parameter. The mean of resulting k numbers of the metric can be used to evaluate the model.

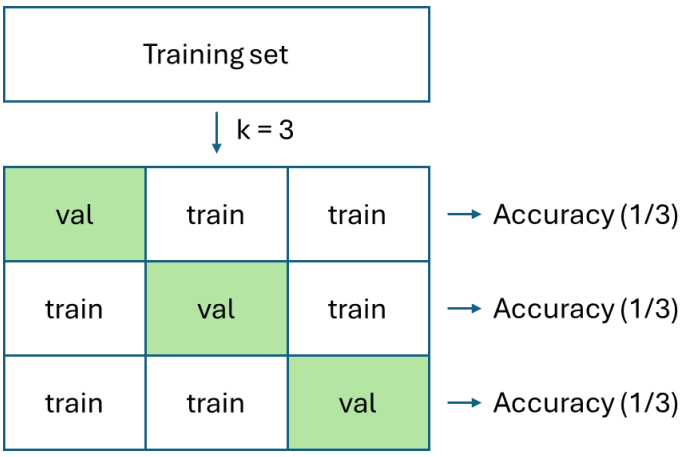

Figure 4: Schematic representation of K-fold cross validation using k = 3.

2.7 Random Forest, an ensemble of Decision Trees

A Random Forest is an ensemble of Decision Trees. The aggregated prediction (soft voting) of the ensemble of trees is the prediction of the Random Forest (scikit-learn documentation, 2024). To understand the Random Forest model it is therefore necessary to understand the Decision Tree model. Hence the following short description focuses on the Decision Tree, and further reading is recommended elsewhere (Géron, 2019).

The iris dataset has been chosen to describe how a Decision Tree make predictions, as the current dataset containing 784 nameless features is less easily visualized. As shown in Figure 5 the features of an observations are filtered by traversing the tree: starting with the root node, followed by child(s) nodes and finally reaching the leaf node. It is at the final leaf node where the prediction of a certain class takes place and according to highest probability of class in the leaf.

The Decision Tree is "grown" by splitting the training set node by node (see section 2.5). Each split results in a boundary perpendicular to an axis. For instance, the left thick vertical line in Figure 5 was the optimal split at depth 0, corresponding to the root node. The dashed horizontal line represents the subsequent split at the right child node. The purity of a node is quantified by Gini impurity (eq. 2).

There are two important differences between Random Forest and Decision Tree. First, Random Forest (like any ensemble of trees) become less sensitive to singular observations when growing. This is because of the soft voting mechanism, where one tree only account for a fraction of the final result. Second, the Random Forest algorithm introduce extra randomness when growing the trees. This is achieved by searching for the best split in a random subset of features, as to compared to searching all features (i.e. Decision Tree).

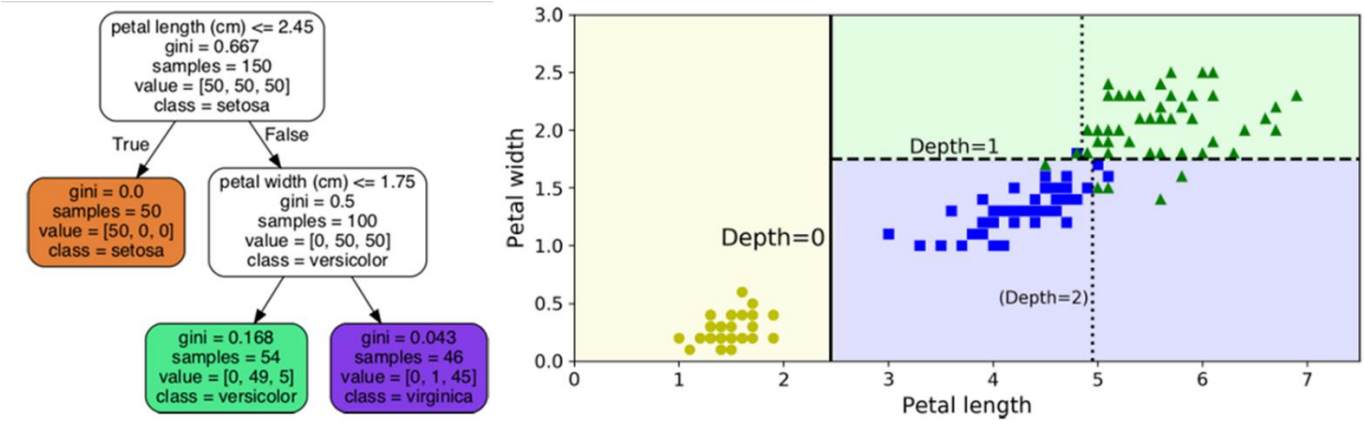

Figure 5: Visualization of a Decision Tree (left) and corresponding decision boundaries (right). These figures have been adapted (Géron, 2019).

3 Method

To fulfill the objectives of this report the practical work was made in part 1-3. Details of design choices can be found in the code (see Appendix A). In all instances the default model hyperparameters were used. Accuracy (eq. 1) was primarily used as metric for evaluation. as Custom functions were often created to solve tasks, and the reader is encouraged to inspect them in.

3.1 Collection and exploration of data (Part 1)

The first part considered data exploration according to the following list,

1. Downloading the MNIST database (LeCun, 2024).
2. Reading the MNIST description.
3. Inspecting the data and exploring the connection between arrays and image.
4. Reviewing attributes of handwritten digits (aspect ratio and dead space above/below the digits). This review was achieved by for-looping the dataset and collect the necessary information. Finally, the attributes were estimated by the mean of observations.

3.2 Random Forest model performance on two different datasets (Part 2)

Random Forest Classification was used to model two different datasets, one having the original 784 features (set A) and the other containing 56 features (set B). Constructing set B was made accordingly:

1. For-looping set A and extracting the necessary 56 features. This was practically achieved using a combination of the numpy.sum() and numpy.transpose() methods.
2. Set B was constructed by creating a pandas dataframe with the 56 features, including the labels from set A.

The steps taken to model set A and B were the same and as follows:

3. The entire dataset were split into training and test.
4. Instancing and model.
5. K-fold cross validation (k=5) was used to evaluate the model.
6. Accuracy and confusion matrix were inspected and used to evaluate the model.

After evaluation the best model was chosen, and two steps followed:

7. The model was retrained on the entire training set.
8. The model was deployed on the test set.

These final two steps were made to inspect that no overfitting had occurred and estimate the generalization error.

3.3 Preprocessing of custom images (Part 3)

One objective of this report was to successfully predict custom handwritten images, taken by smartphone images and snapshots using the web camera. One smartphone image were taken and loaded into the computer (See Figure 6). The image was cropped into 10 digits and used without further processing. Images using web camera was made on a local Streamlit application.

Preprocessing both the MNIST dataset and custom images was necessary for prediction of custom images, by making the data compatible (i.e. normalized). The preprocessing steps followed the below listed order:

1. Identifying the background by looking at the top and lowest 2 rows. Quantification was made by selecting the highest value. Pixels with intensities < 1.05 intensity, compared to the background, was set to 0 (custom images only).
2. Replace the digit in the top right corner of the 28x28 pixel image (see Figure 7).
3. Make the digit thicker by "smearing" it on the canvas (custom images only).
4. Normalize the intensity of pixel so that it ranges from 0-1.

These steps were made using custom function which operated dynamically. To emphasize, no hard-coding was made to fit the instances of certain images.

After preprocessing it was necessary to repeat step 7 and 8 in part 2 (section 3.2), for the sake of creating a model fit to the task.

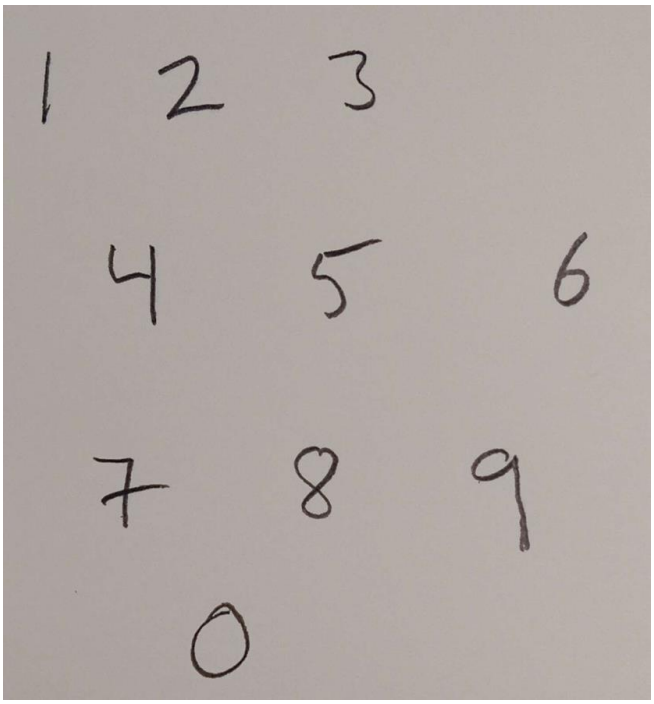

Figure 6: Image created by smartphone containing 10 digits. Single digits were created by cropping this image.

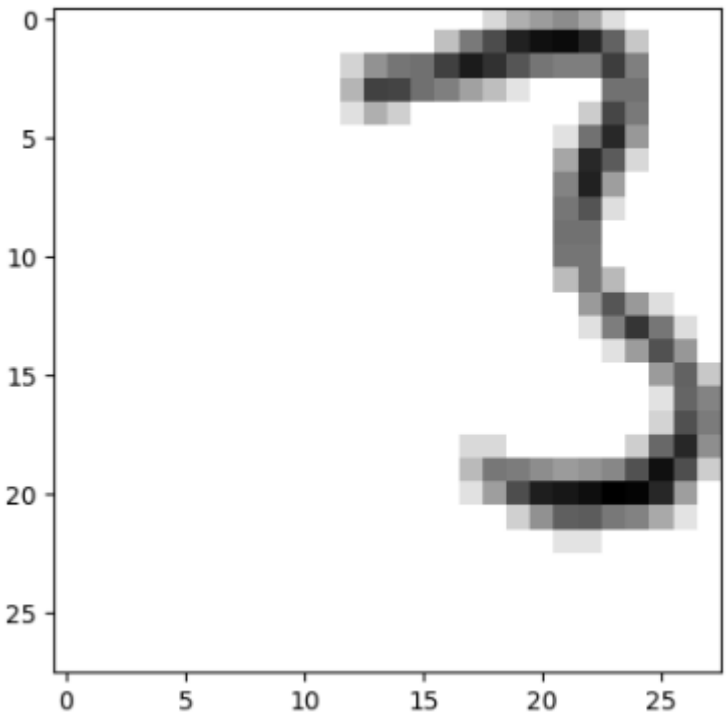

Figure 7: Custom Image reprocessed by removing background and moving the digit to right top corner using custom functions.

3.4 Construction of a Streamlit application

A Streamlit application was built with the following list of user allowed functionalities:

- Take a snapshot using the web camera live feed.
- Re-crop and positioning the snapshot, with zoom-in capabilities.
- Real time displaying of the formatted snapshot to help user.
- Guiding lines to make the user stay out of areas used for background analysis.
- Button to use the cropped image and predict value.

The code for the streamlit application was created in PyCharm. Preprocessing steps were using the same code as for smartphone images. Functionalities were created using both ChatGPT and streamlit documentation (Sam Altman, 2022), (Streamlit documentation, 2024).

4 Results and discussion

4.1 Data Exploration

The MNIST data was explored by analyzing the width and height of digits. From these observations a mean was calculated from the entire dataset. The following results were obtained,

- White “dead space” above and below digits make up 29 % of the image height.
- Aspect ratio (length/width) of the digits are approximately 1.3.

These two results have no effect modelling of MNIST data. However, it would surely affect prediction capabilities of models trained on this dataset. Consider presenting an image with skewed aspect ratio. An extremely wide and short “8” would be considered a horizontal line, not being recognizable by the model. In fact, such observation matches none in the dataset. Thus, these results are import to consider as “restrictions” when predicting custom images. Similar discussion can be made for the position of the digit.

4.2 Accuracy of Random Forest model using 2 different datasets

The Random Forest Classifier was used to model the MNIST data using 2 different datasets. The accuracy scores of tests are shown in Table 1 and confusion matrixes are presented in Figure 8.

As expected from trial experiments, the model was able to achieve an accuracy > 95 % using the original 784 features. Interestingly, viewing the digits from “above” and “side” (i.e. 56 features) the model was able to score an accuracy of 91.3 %. This was not expected, as features were reduced to only 7 % of the original amount, and most would agree this way of viewing digits are indeed odd. There is a difference of 5.4 % accuracy between the two datasets, therefore the objective stating “within 5 % margin” was unsuccessful. Still, this result demonstrates the power of feature selection.

Confusion matrixes show similar behavior of the model on the two datasets. There are few faulty predictions made on the 784 feature dataset. This was an expected result. For the 56 feature dataset it is interesting to consider the true/predicted labels in more detail. For example, in 92 instances a “5” was predicted as “3”. By comparing the two confusion matrixes created by the same model, we can draw the following conclusion: it is more difficult to distinguish handwritten numbers by looking from “above” and “side”. This conclusion is arguably not possible to draw without using machine learning.

From the 5-Fold Cross Validation test it was decided to use the 784 features for prediction of handwritten digits. Once refit on entire training set, the model was finally evaluated on the test set. The Random Forest Classifier scored 97.0 % accuracy. By comparison of previous score (96.7 %) there is no evident overfit of the model.

Table 1: Accuracy scores of Random Forest Classifier on 2 different datasets.

Test	784 features (original)	56 features (custom)
5-Fold Cross Validation (mean)	96.7 %	91.3 %
Refit on entire train set and validated on test set	97.0 %	n/a

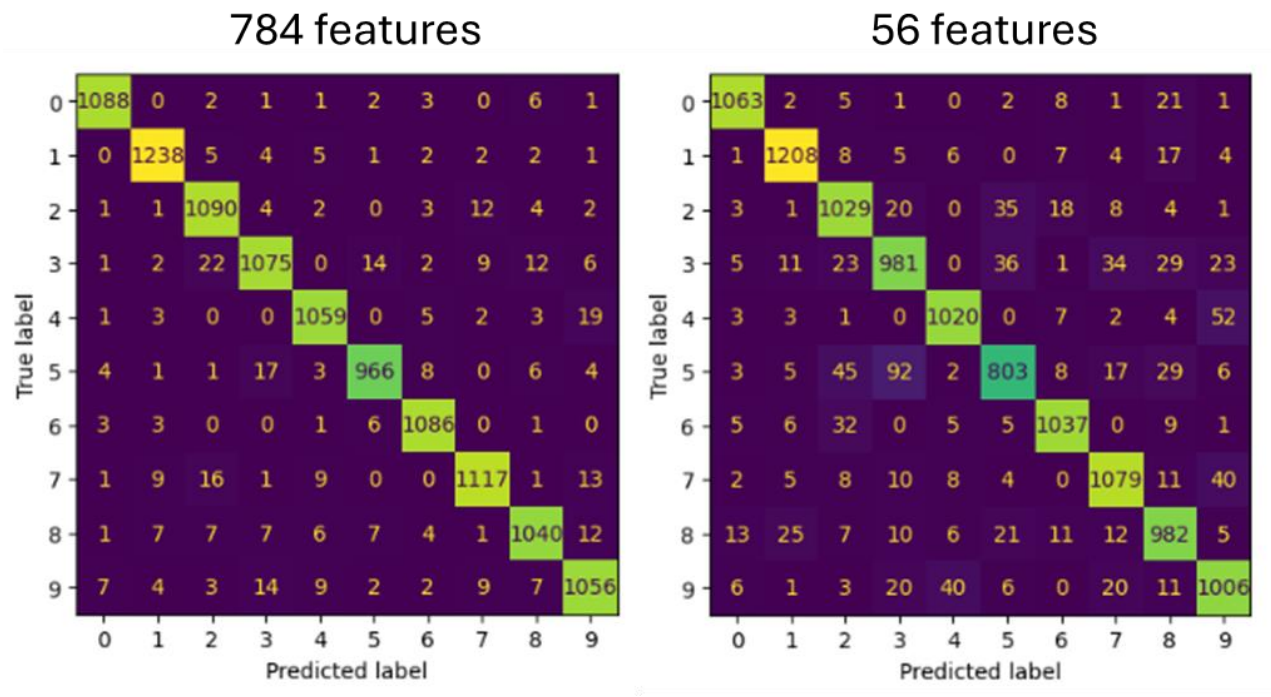

Figure 8: Confusion matrix for Random Forest Classifier on data with 784 features (left) and 56 features (right).

4.3 Predicting custom images from smartphone and identifying preprocessing steps

A total of 10 handwritten digits, from images taken with a smartphone, was predicted using the Random Forest Classifier. Without any preprocessing, the model mostly failed (70 %) to make accurate prediction. Approaching this challenge involved a few trials with errors, and it was believed that the issue was due to poor custom image compatibility with the MNIST dataset. Therefore, preprocessing steps (mentioned in section 3.3) were taken for both the MNIST and custom images. As a result, the model was able to yield 100 % accuracy (see Figure 9) which satisfies the objective of achieving > 90 % accuracy. This was an unexpected result, which prove that the preprocessing step(s) were necessary. We believe all steps to be necessary, as each step would intuitively improve the success rate of the model. However, with only one experiment such conclusion cannot be made.

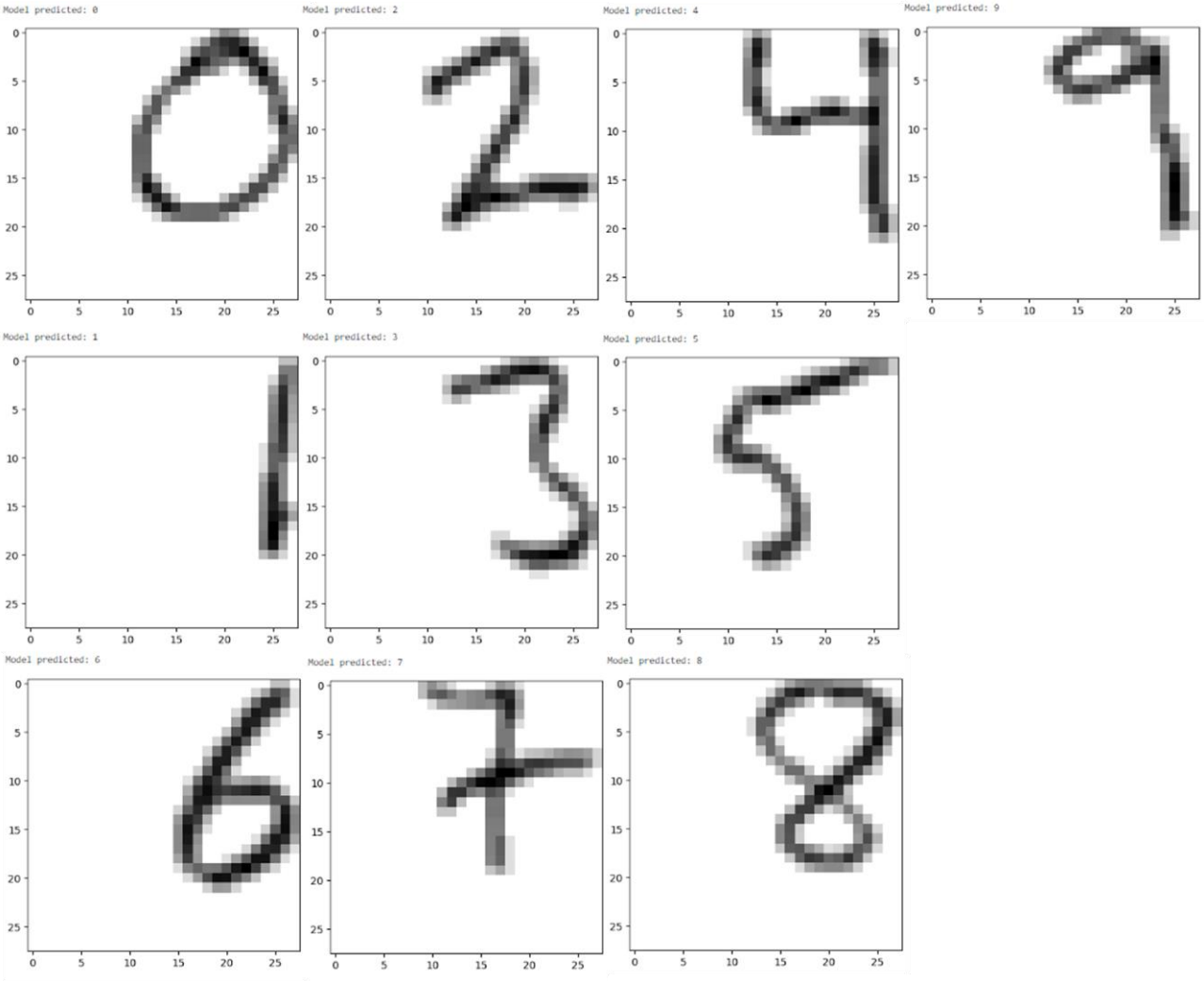

Figure 9: Predictions of handwritten images (smartphone). Images have been preprocessed.

4.4 Prediction of images using web camera (Streamlit application)

A streamlit application (local) was created and a screenshot of its functionality can be seen in Figure 10. Here, the application is successful in its prediction of a “4”. The user is able to take a photo using the web camera, re-crop and see the adjustments live. Horizontal lines are used as guides. The height above and below these lines make up 28 % of the image height which help data compatibility with the MNIST dataset. This deadspace is used to identify the background intensity. In the backend side, the image is reprocessed and the result is shown as an image. Finaly, a trained model is uploaded and predicts the handwritten digit. With these functionalities, the Streamlit application offer flexibility regarding some constraints (e.g. centered digit, uneven background and other objects in the picture). However, the ability to predict was closely related to the thickness of the image. If too thin digit, the Streamlit application was not able to predict the value accurately. Some adjustments were made by smearing the digit onto the image, making in thicker. However, the amount of required smearing is unique to each person’s handwriting style. Though not verified, regards to aspect ratio (length/width) of digits are believed to influence model performance. These issues remains and needs to be resolved.

Figure 10: Screenshot of Streamlit application created in this project.

5 Conclusions

The following conclusions regarding set out objectives, tasks and research questions can be drawn,

1. A Random Forest Classifier model was created, which achieved an accuracy > 95 %.
2. With 56 custom features it was not possible to achieve the same accuracy within a margin of 5 %, using the same model. It was concluded that it is more difficult to distinguish between digits by looking from “above” and “side”. However, it is an interesting result that with only 7 % of the original amount of features, the model could achieve 91 % accuracy.
3. Handwritten and photographed digits could be predicted accurately (100 %). This was owed entirely to necessary preprocessing of images. Without such steps, the model performed poorly.
4. The following preprocessing steps were considered and adjusted for:
    - Placement of digit in the image
    - Background intensity
    - Normalization of intensity
    - Thickness

    With these adjustments it was feasible to make “live stream predictions” using a web camera.
5. A Streamlit application was constructed with appropriate functions (re-crop, zoom in and positioning). These functions made it possible for a user to access the intended function of accurate “live prediction” of handwritten digits. However, it was noticed that the application was sensitive digits drawn too thin. This issue was not resolved.

Appendix A

The Jupyter notebook code for this report can be found below,

Collection and exploration of data (Part 1)

```
mnist = fetch_openml('mnist_784', version=1, cache=True, as_frame=False)
print(mnist.DESCR)

X = mnist["data"]
y = mnist["target"].astype(np.uint8)

# inspect data
print(X.shape)
print(y.shape)
print(X[0])
print(y[0])

# Plotting the features of the X data, it looks like a 5.
some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
print(some_digit_image)

# Count feature attributes

def count_feature_attribute(my_matrix, control):
    new_matrix = my_matrix.reshape(28, 28)

    empty_matrix = []
    count_length_deadspace = 0
    count_length_digit = 0

    for idx, num in enumerate(new_matrix):
        if num.sum() < 1:
            count_length_deadspace += 1
        if num.sum() > 0:
            empty_matrix.append(num)

    empty_matrix = np.array(empty_matrix)

    diff = 28 - empty_matrix.shape[0]
    zeros = np.zeros((diff, empty_matrix.shape[1]))


    if control == True:

        new_empty_matrix = np.concatenate((empty_matrix, zeros), axis=0)
    else:
        new_empty_matrix = np.concatenate((zeros, empty_matrix), axis=0)

    new_empty_matrix =new_empty_matrix.transpose()

    count_length_digit = 28-count_length_deadspace

    return new_empty_matrix, count_length_deadspace, count_length_digit

heights_dead_space = []
heights_digits = []
width_dead_space = []
width_digits = []

for item in X:
```

```
    output_1 = count_feature_attribute(item, True)
    new_item = output_1[0]
    heights_dead_space.append(output_1[1])
    heights_digits.append(output_1[2])

    output_2 = count_feature_attribute(new_item, True)
    width_dead_space.append(output_2[1])
    width_digits.append(output_2[2])

print(np.mean(heights_dead_space))
print(np.mean(heights_digits))
print(np.mean(width_dead_space))
print(np.mean(width_digits))
print()
print(f"sum of dead_space above and below digit should be approx: {int(100*(np.mean(heights
_dead_space)/28))} % of image height")
print()
print(f"aspect ratio (length/width) digit should be approx: {np.mean(heights_digits)/np.mea
n(width_digits)}")
```

Random Forest model performance on two different datasets (Part 2)

```
# For-looping set A and extracting the necessary 56 features.

X_new_concat_final = []

# for item in X_new:
for item in X:

    item = item.reshape(28, 28)

    row_sum = []
    column_sum = []

    for array in item:
        row_sum.append(np.sum(array))

    new_matrix_transposed = item.transpose()

    for array in new_matrix_transposed:
        column_sum.append(np.sum(array))

    row_sum = np.array(row_sum)
    column_sum = np.array(column_sum)

    row_sum = row_sum.reshape(1, 28)[0]
    column_sum = column_sum.reshape(1, 28)[0]

    # X_new_concat = np.concatenate((item, row_sum), axis=None)
    # X_new_concat = np.concatenate((X_new_concat, column_sum), axis=None)
    X_new_concat = row_sum
    X_new_concat = np.concatenate((X_new_concat, column_sum), axis=None)
    # X_new_concat = np.concatenate((X_new_concat, row_sum[::-1]), axis=None)
    # X_new_concat = np.concatenate((X_new_concat, column_sum[::-1]), axis=None)

    X_new_concat_final.append(X_new_concat)

X_new_concat_final = np.array(X_new_concat_final)

# Dumping data into dictionary

dict = {}
```

```
for i in range(0,(28+28)):

    current_list = []

    for index,number in enumerate(X_new_concat_final):

        current_list.append(number[i])  #första frame, ta första element, ny frame, ta förs
ta element.

    dict[f"x_{i}"] = current_list

dict["label"] = y

df = pd.DataFrame(dict)

df.head()

X_to_model = df.drop(['label'], axis=1).values
y_to_model = df['label'].values
```

Splitting data into train and test

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, s
tratify = y)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

X_train_56, X_test_56, y_train_56, y_test_56 = train_test_split(X_to_model, y_to_model, tes
t_size=0.2, random_state=42, stratify = y)
print(X_train_56.shape)
print(X_test_56.shape)
print(y_train_56.shape)
print(y_test_56.shape)
```

Instancing and fit Random Forest Classifier models

```
# 784 features
random_forest_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
# 56 features
random_forest_clf_56 = RandomForestClassifier(n_jobs=-1, random_state=42)

scores_random_forest_clf = cross_val_score(random_forest_clf, X_train, y_train, cv=5)
scores_random_forest_clf_56 = cross_val_score(random_forest_clf_56, X_train_56, y_train_56,
cv=5)

print(np.mean(scores_random_forest_clf))
print(np.mean(scores_random_forest_clf_56))
```

Splitting and retraining model for the sake of Confusion Matrix

```
X_train_cm, X_val_cm, y_train_cm, y_val_cm = train_test_split(X_train, y_train, test_size=0
.2, random_state=42, stratify = y_train)
X_train_56_cm, X_val_56_cm, y_train_56_cm, y_val_56_cm = train_test_split(X_train_56, y_tra
in_56, test_size=0.2, random_state=42, stratify = y_train_56)

random_forest_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
random_forest_clf.fit(X_train_cm, y_train_cm)
random_forest_clf_56 = RandomForestClassifier(n_jobs=-1, random_state=42)
random_forest_clf_56.fit(X_train_56_cm, y_train_56_cm)

random_forest_clf_pred = random_forest_clf.predict(X_val_cm)
random_forest_clf_56_pred = random_forest_clf_56.predict(X_val_56_cm)

# These names were obtained above
target_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
cm1 = confusion_matrix(y_val_cm, random_forest_clf_pred)
cm2 = confusion_matrix(y_val_56_cm, random_forest_clf_56_pred)

fig, axs = plt.subplots(1, 2, figsize = (10, 9), layout='constrained')
ConfusionMatrixDisplay(cm1, display_labels = target_names).plot(ax=axs[0])
ConfusionMatrixDisplay(cm2, display_labels = target_names).plot(ax=axs[1])

random_forest_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
random_forest_clf.fit(X_train, y_train)

random_forest_clf_pred = random_forest_clf.predict(X_test)
print(classification_report(y_test, random_forest_clf_pred, target_names=target_names))

joblib.dump(random_forest_clf, "model.pkl")
```

Preprocessing of images (Part 3)

prepocessing of MNIST dataset

```
# Custom function to dynamically find background of image

def find_background(my_matrix):
    new_matrix = my_matrix.reshape(28, 28)

    first_max = np.max(new_matrix[:2])
    last_max = np.max(new_matrix[-2:])

    return np.max([first_max,last_max])

# Custom function to replace digit in top right corner

def remove_dead_space(my_matrix, control):
    new_matrix = my_matrix.reshape(28, 28)

    empty_matrix = []

    for idx, num in enumerate(new_matrix):
        if num.sum() > 0:
            empty_matrix.append(num)

    empty_matrix = np.array(empty_matrix)

    diff = 28 - empty_matrix.shape[0]

    zeros = np.zeros((diff, empty_matrix.shape[1]))

    if control == True:

        new_empty_matrix = np.concatenate((empty_matrix, zeros), axis=0)
    else:
        new_empty_matrix = np.concatenate((zeros, empty_matrix), axis=0)

    new_empty_matrix =new_empty_matrix.transpose()

    return new_empty_matrix


# Looping through MNIST data to put digit in top right corner
X_new = []

for item in X:
#     print(len(item))
```

```
    new_item = remove_dead_space(item, True)
    X_new.append(remove_dead_space(new_item, False))

# Normalize MNIST data
X_new = np.array(X_new)
X_new = X_new.reshape(-1,784)
X_new = X_new / 255 #highest datapoint is always 254-255 in MNIST.

# Checking the results after
some_digit = X_new[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
# plt.axis("off")

print(X_new[0])
```

Split the preprocessed MNIST dataset, train the model and check generalization error. Finally saving the model locally.

```
X_train_pp, X_test_pp, y_train_pp, y_test_pp = train_test_split(X_new, y, test_size=0.2, ra
ndom_state=42, stratify = y)

random_forest_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
random_forest_clf.fit(X_train_pp, y_train_pp)

random_forest_clf_pred = random_forest_clf.predict(X_test_pp)
print(classification_report(y_test_pp, random_forest_clf_pred, target_names=target_names))

joblib.dump(random_forest_clf, "model.pkl")
```

Preprocessing of custom images (images taken with mobile phone)

```
import os

X_test_images = []

directory = "./bilder/egna_bilder/"  # Specify the directory where the files are located

# List the files in the directory
files = os.listdir(directory)

for filename in files:
    if os.path.isfile(os.path.join(directory, filename)):
        # Read the image file using OpenCV and convert it to grayscale
        test_image = cv2.imread(os.path.join(directory, filename), cv2.IMREAD_GRAYSCALE)
        img_resized = cv2.resize(test_image, (28,28), interpolation=cv2.INTER_LINEAR)
        img_resized = cv2.bitwise_not(img_resized) #invert image
        img_resized = img_resized.reshape(-1,784)
        X_test_images.append(img_resized.reshape(-1,784))
        print(filename)

some_digit = X_test_images[3]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
print(X_test_images[3])

# Custom function to find background dynamically

def find_background(my_matrix):
    new_matrix = my_matrix.reshape(28, 28)

    first_max = np.max(new_matrix[:2])
    last_max = np.max(new_matrix[-2:])
```

```
    return np.max([first_max,last_max])

# Set background to 0 and normalize

my_X_new = []
background_list = []

for item in X_test_images:

    background = find_background(item)
    top_number = np.max(item)

    my_X_new.append(item.flatten()/np.max(item))

    background_list.append(background/top_number)

X_test_images = my_X_new

for index,item in enumerate(X_test_images):

    item[item < background_list[index]*1.05] = 0 #works equally well for smartphone and web
cam images. May not work using other peoples equipment?

print(len(X_test_images))
some_digit = X_test_images[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)

print(X_test_images[0])

# Replace smart phone images to the top right

my_new_X = []

for item in X_test_images:
    new_item = remove_dead_space(item.reshape(-1,784), True)
    my_new_X.append(remove_dead_space(new_item, False))

plt.imshow(my_new_X[0], cmap=mpl.cm.binary)

# Custom function for smearing a digit onto the "canvas". To make the image thicker.

def make_thicker(my_matrix):

    thick_matrix = []

    for item in my_matrix:
# for item in my_new_X[:1]:

    # print(item)

        zeros = np.zeros((1,28))

        move_down = item[:-1]
        one_move_down = np.concatenate((zeros,move_down), axis=0)
        together_down = item+one_move_down

        together_down = together_down.transpose()
        # print(len(together_down))
        move_down = together_down[1:]
        # print(len(move_left))
```

```
        # print(len(zeros))

        one_move_left = np.concatenate((move_down,zeros), axis=0)
        # print(len(one_move_left))
        together_left = together_down+one_move_left

        thick_matrix.append(together_left.transpose())

    # print(together_left.shape)
    # print(together_left)

    # plt.imshow(together_left, cmap=mpl.cm.binary)

    return thick_matrix

# make digits thicker

my_thick_X = make_thicker(my_new_X)

my_X_new = []

for item in my_thick_X:

    my_X_new.append(item.flatten()/np.max(item))

my_thick_X = my_X_new

plt.imshow(my_thick_X[3].reshape(28,28), cmap=mpl.cm.binary)
```

Predicting Smartphone images

```
my_clf = joblib.load("model.pkl")
print()

for item in my_thick_X:

    item = item.reshape(-1, 784)
    print(f"Model predicted: {my_clf.predict(item)[0]}")
#     print(item[0])
#     print()
    some_digit_image = item.reshape(28, 28)
    plt.imshow(some_digit_image, cmap=mpl.cm.binary)
    plt.show()
```

8 Referenser

Ajay Kulkarni, D. C. (2020). Foundations of data imbalance and solutions for a data democracy. *Data Democracy*, 83-106.

Géron, A. (2019). *Hands-on Machine Learning with Scikit-Learn Keras & TensorFlow.* Canada: O'Reilly.

Haenlein, M. &. (2019). A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. *California Management Review*, 61.

LeCun, Y. (2024). Hämtat från https://yann.lecun.com/exdb/mnist/

Sam Altman. (2022). Hämtat från https://chat.openai.com/

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 210-229.

scikit-learn documentation. (2024). *scikit-learn.org.* Hämtat från scikit-learn.org: scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Streamlit documentation. (2024). Hämtat från https://docs.streamlit.io/

Sundar Pichai . (den 4 10 2016). *Town Hall Event Google.* San Francisco.