Machine Learning - Final Project

Frederik Predut - StudieNr: 201502305

Magnus Hauge Kyneb - StudieNr: 201807279

Sebastian Laczek Nielsen - StudieNr: 201806678

Group 22 - December 2020

Table of Contents

1 Introduction

This report covers the final project O4 and concludes the groups work on the course ITMAL at Aarhus University. Every participent has contributed equally in the forming of this project/rapport and has equally been engaged in the disscussion regarding issues revolving the various problems and subjects depicted in this report.

The final project is centered around Google's QuickDraw! browser game. This challenges players to doodle an object based on a word and have Google's ML-model guess what the object is. The player attempts are then collected added to a huge dataset that is made public. On figure 1 you can see the various categories of doodles.

Figur 1: Various doodle categories.

2 Problem

In this project, we attempt to solve the problem of classifying a drawn object. In this case, the classification is limited to one of two categories, a donut-doodle or house-doodle.

3 Dataset

Each category in the QuickDraw!-dataset contains over 100.000 examples of doodles of that category's object. On figure 2, a snippet of the donut-category dataset i shown.

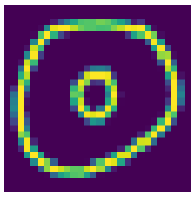
Figur 2: Some doodles from the donut-category.

To maximize the chance of our machine learning methods working, two very different looking doodle object categories are chosen. The house drawings are generally contain a lot of straight lines, while the donuts doodles mostly contain curves. Hopefully, a machine learning method might be able to pick up on this difference. On figure 3, a snippet of the donut-category dataset i shown.

Figur 3: Some doodles from the house-category.

Google has made it esay to work with the doodle-data. They've preprocessed the data in various and are providing various formats to choose from. Some versions of the dataset has timing-data in addition of the drawing in addition to the final doodle. The timing-data describes how the doodle was drawn.

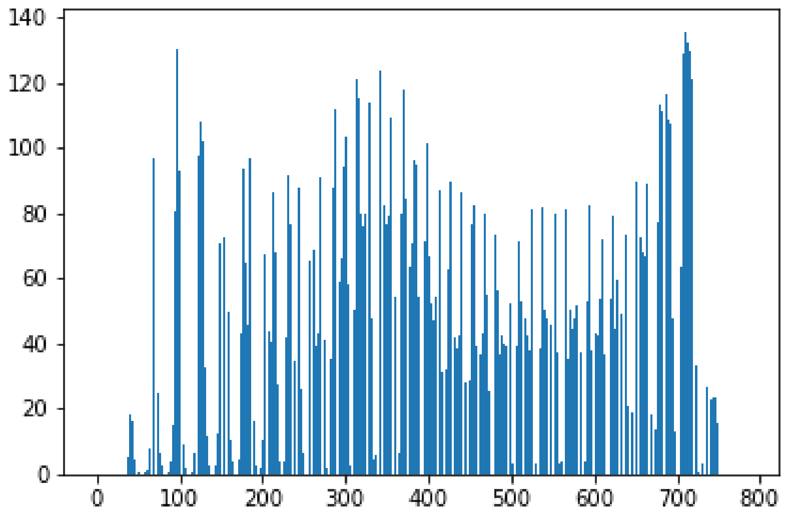
Format:
In this project the data format used is the numpy bitmap versions of the final doodles. These bitmap contains 784 values between 0 and 255. Each value corresponds to a pixel with a grayscale value of 0 to 255. The 784 values can be reshaped into a matrix showing a downscaled 28x28 resolution complete doodle. A plot of a reshaped donut sample can be seen on figure 4.


Figur 4: A donut sample visualized by plotting with the matplotlib.pyplot library.

Size and features:
In total, the donut-dataset contains 140751 donut-doodle samples, each with 784 features with values ranging from 0 to 255. The house-dataset contains 135420 houses, again with 784 features with values ranging from 0 to 255.

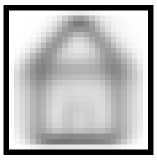
4 Data Analysis

After loading the data, it was concluded the best way to represent all images was, taking each pixel position and average all pixel values, then plotting it. This method would give what the average dataset would look like. The bar graph produced for the house dataset can be seen on figure 5

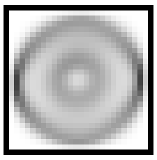


Figur 5: Bar graph of the average house

Converting this new data to an image gives us exactly what the average dataset looks like. In this case on figure 7 is how the average house looks like.



Figur 6: Average house as on a per pixel basis.



Figur 7: Average donut on a per pixel basis.

This was done to both our house and donut dataset, where both the image and graphs was compared to ensure that the analysis was different enough from each other.

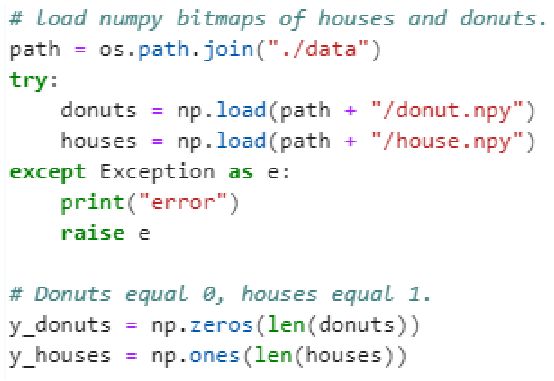
5 Machine Learning Pipeline

To get from data to model can be broken down to four main steps:

- Data Preparation - Labelling and splitting into test and training sets.
- Data Processing - Dimensionality reduction Principal Component Analysis (PCA).
- Training the Model(s) - Setting up learning algorithms and training them.
- Testing the Model(s) - Predicting output from new input, assessing performance.

6 Data Preparation

Labelling:
Because data is loaded by category, labelling is easy. Donuts are labelled 0 and houses are labelled 1. This is simply done by creating a ground-truth output-vector with size equal to the number of samples of each category, see figure 8. Using the labels and using input/output-pairs to train a model, categorizes the learning method as ***supervised learning***.

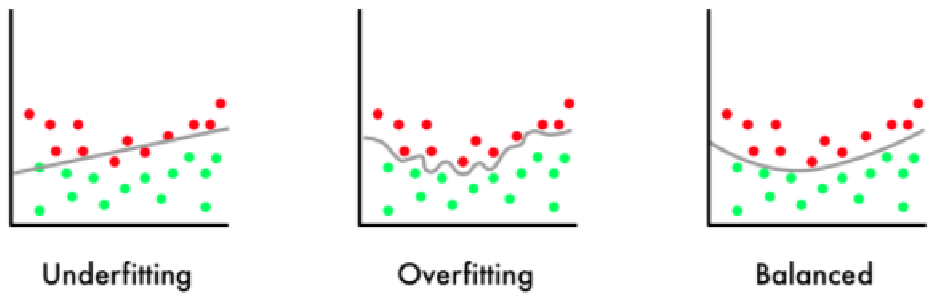


```
# Load numpy bitmaps of houses and donuts.
path = os.path.join("./data")
try:
    donuts = np.load(path + "/donut.npy")
    houses = np.load(path + "/house.npy")
except Exception as e:
    print("error")
    raise e

# Donuts equal 0, houses equal 1.
y_donuts = np.zeros(len(donuts))
y_houses = np.ones(len(houses))
```

Figur 8: Creating labelling for donuts and houses.

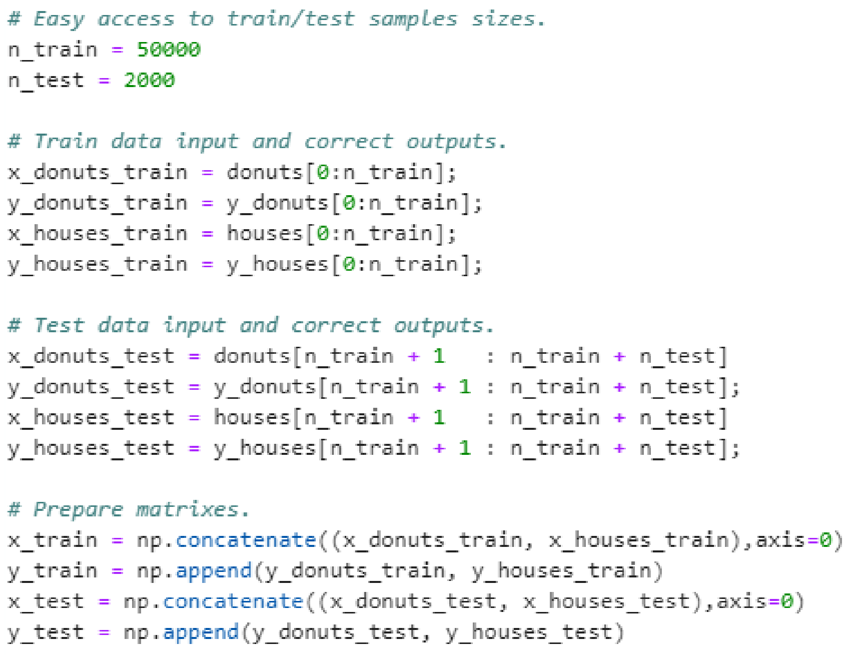
Splitting into training and test sets:
Data is split into a training-set and a test-set. The training-set is available when training the algorithm of choice, while the test should be kept completely seperated from the training process. This later allows for more valid performance metrics. Training on the entire dataset and then testing on that same data could initially show extremely good performance, but wouldn't be representative model's actual ability to predict on new data. Somewhat like if students were examined on the same assignment that they've been training on all semester.

The initial extremely good performance seen from training models on the entire dataset, then testing on that same data, is often due to ***overfitting***. As the model knows every data point, it shapes the model that is too specific to that set of data. We intuitively believe this becomes less of a problem the larger the dataset is, as it becomes impossible for the model to fit every data point and it instead settles for something more average and balanced. On figure 9, underfit, overfit and a balanced fit is visualized. To achieve a balanced fit, it important that the capacity of the model fits the complexity of the dataset. Having a large capacity but a simple dataset, makes the model overfit as it assumes the problem to be too complex in relation to reality. And having a low capacity in relation to the complexity of the data, results in underfitting, as the model predicts the dataset to be too simple compared to reality.



Figur 9: Under-, over- and balanced fit visualized. Source: https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d (8/12/2020)

Despite the very large amount of data in the QuickDraw!-dataset, we decide to split the data. It's good practice to split the data into test and training set and also gives a better estimate of how well the model would predict new data. Figure 10 shows how the data split is performed. Both houses and donuts are split, and each train-split and test-split is concatenated. The same is done for the output data.



```
# Easy access to train/test samples sizes.
n_train = 50000
n_test = 2000

# Train data input and correct outputs.
x_donuts_train = donuts[0:n_train];
y_donuts_train = y_donuts[0:n_train];
x_houses_train = houses[0:n_train];
y_houses_train = y_houses[0:n_train];

# Test data input and correct outputs.
x_donuts_test = donuts[n_train + 1   : n_train + n_test]
y_donuts_test = y_donuts[n_train + 1 : n_train + n_test];
x_houses_test = houses[n_train + 1   : n_train + n_test]
y_houses_test = y_houses[n_train + 1 : n_train + n_test];

# Prepare matrixes.
x_train = np.concatenate((x_donuts_train, x_houses_train),axis=0)
y_train = np.append(y_donuts_train, y_houses_train)
x_test = np.concatenate((x_donuts_test, x_houses_test),axis=0)
y_test = np.append(y_donuts_test, y_houses_test)
```

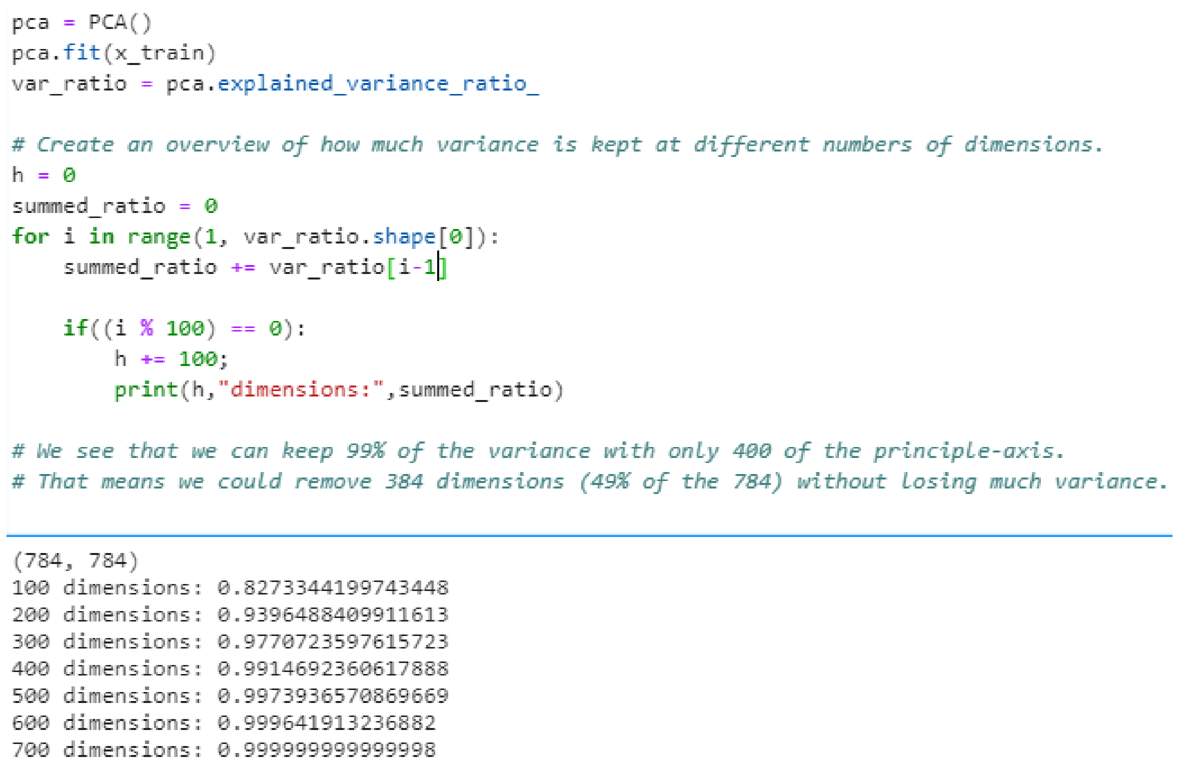
Figur 10: Creating labelling for donuts and houses.

7 Data Processing

Principel Component Analysis (PCA):
Each sample in the dataset has 784 features, one for each pixel, and therefore 784 dimensions. To simplify the data, we use to PCA. PCA is in itself not a dimensionality reduction method, but is a stepping stone to reduce dimensionality while maintaining variance in the features. Simplifying the data and getting more variance per dimension can improve the result of some machine learning algorithms. In addition, it also speeds up the training process, which can quickly become a limiting factor with large datasets with many dimensions. PCA can help us avoid the "curse of dimensionality".

Fitting the scikitlearn's PCA to the training data generates 784 new dimensions (also known as principal components) based on the original dimensions. The new dimensions have been optimized for variance and are ordered from most variance to least variance. The ratio between each new dimension's variance and the total variance can be extracted through the explained-variance-ratio attribute. These ratios are ordered from highest ratio to lowest.

By summing up the ratio of each new dimension, starting from the dimension with most variance going down, we get an estimate for how much variance is preserved in relation to how many dimensions we include. You see this summation on figure 11.



```
pca = PCA()
pca.fit(x_train)
var_ratio = pca.explained_variance_ratio_

# Create an overview of how much variance is kept at different numbers of dimensions.
h = 0
summed_ratio = 0
for i in range(1, var_ratio.shape[0]):
    summed_ratio += var_ratio[i-1]

    if((i % 100) == 0):
        h += 100;
        print(h,"dimensions:",summed_ratio)

# We see that we can keep 99% of the variance with only 400 of the principle-axis.
# That means we could remove 384 dimensions (49% of the 784) without losing much variance.
```

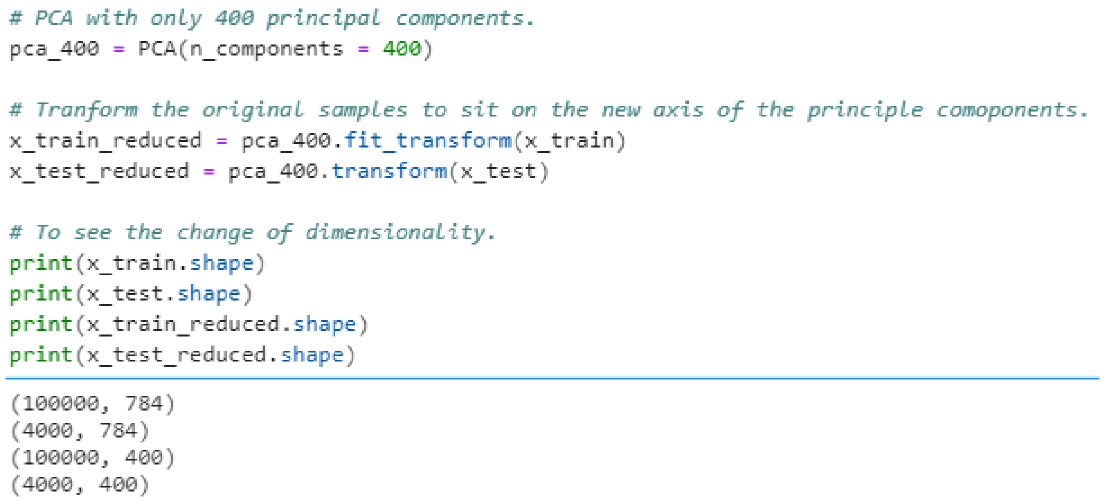
```
(784, 784)
100 dimensions: 0.8273344199743448
200 dimensions: 0.9396488409911613
300 dimensions: 0.9770723597615723
400 dimensions: 0.9914692360617888
500 dimensions: 0.9973936570869669
600 dimensions: 0.999641913236882
700 dimensions: 0.999999999999998
```

Figur 11: Finding the sum of variance ratios at increasing amounts of principle components.

The summation shows that using 400 of the principle components would preserve 99 percent of the variance of the original 784 dimensions. It's also striking that 83 percent of the variance is kept with only 100 principle components. This suggests that a *massive* dimensionality reduction is possible.

The actual reduction happens when we transform the data. This can be done through the PCA's fit-transform function. We first specify how many principle components that we want to keep, then the function creates new data with that number of components, of course using the components with the most variance.

Under the hood, the PCA generates 784 temporal features for each component. Each temporal feature describes the relation between the new component and one of the original dimensions of the data. The sum of all these relations between the original dimension and the new principle component outputs the sample's value on the new principle component. If we choose to keep 400 components, this is done for each of the new 400 principle components. On figure 12 the transformation-process is shown.



```
# PCA with only 400 principal components.
pca_400 = PCA(n_components = 400)

# Tranform the original samples to sit on the new axis of the principle comoponents.
x_train_reduced = pca_400.fit_transform(x_train)
x_test_reduced = pca_400.transform(x_test)

# To see the change of dimensionality.
print(x_train.shape)
print(x_test.shape)
print(x_train_reduced.shape)
print(x_test_reduced.shape)
```

```
(100000, 784)
(4000, 784)
(100000, 400)
(4000, 400)
```

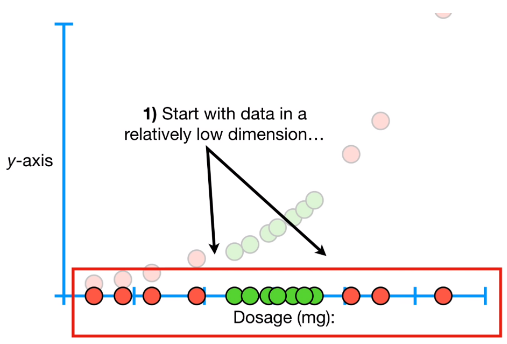
Figur 12: Dimensionality reduction from 784 dimensions to 400 while keeping 99+ percent variance.

The fit-transform is applied to the training data, while a transform *without fit* is run on the test data. The whole point of the test data is that it has not affected the model in any way and therefore becomes useful for measuring performance of the model in unseen situations. Even fitting nothing but the PCA using test data, would deminish the validity of future performance claims based on that test data.

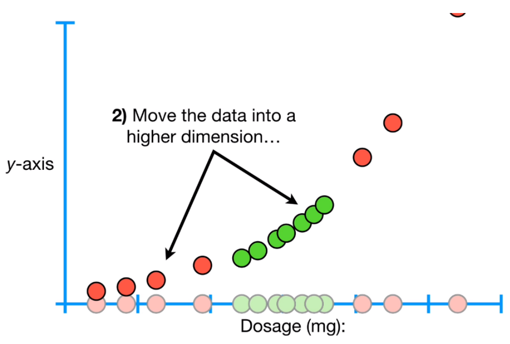
8 Choice of Algorithm(s)

Going through the documentation for the various machine learning methods, it was decided to continue with three learning algorithms.

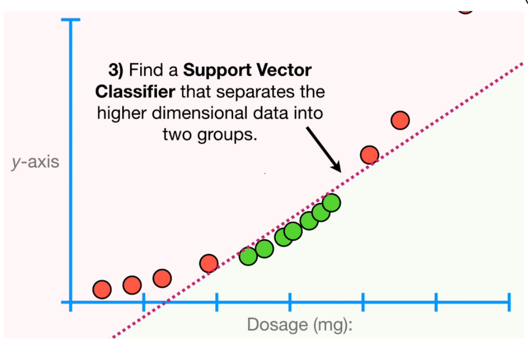
C-Support Vector Classifier:
C-Support Vector Classifier (SVC), a type of SVM. SVC comes with parameters for regulization, maximum iterations, kernel types and more. SVM's work by moving the data to a higher dimension in order to create better decision boundaries. The process is explained on figure 13.



(a) Step 1



(b) Step 2


3) Find a **Support Vector Classifier** that separates the higher dimensional data into two groups.
y-axis
Dosage (mg):

(c) Step 3

Figur 13: SVM classification explained. Source: https://youtu.be/efR1C6CvhmE (11/12/2020)

According to the documentation, these have the following advantages:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

SVM's effectiveness in high dimensional spaces and memory efficiency seems like important advantages when dealing with 400 dimensions, a large data set and very limited compute. Various disadvantages were also listed, such as not working well when the features far outnumber the amount of samples, which is not the case with the QuickDraw!-dataset.

Stochastic Gradient Descent Classifier:
SGD is strictly speaking only an optimization technique, that uses a random subset of samples for each iteration. This technique comes with a few advantages according to documentation:

- Efficiency.
- Ease of implementation.

We as a group have previously used the SGD-Classifier, which was the primiary reason for tring out the algorithm again. And, as can be infered from the name, it's a classifier, which is what we need. All this while being relatively efficient to train.

Multi-layer Perceptron Classifier:
This algorithm also uses stochastic gradient descent, but results in a neural network. The QuickDraw! website reveals that Google is using a neural network to guess the drawing. It also make some sense intuitively, that the problem could be solved by having each pixel correspond to at least one node in a network and get outputs based on the weight of those. Therefore, MLP was chosen as the third and final contender.

9 Performance Metrics

When training the chosen algorithms some way of comparing performance was needed. Accuracy is a useful performance metric, especially when training a binary classifier with an equal amount of donuts and houses. On figure 14 the formula for accuracy is shown.

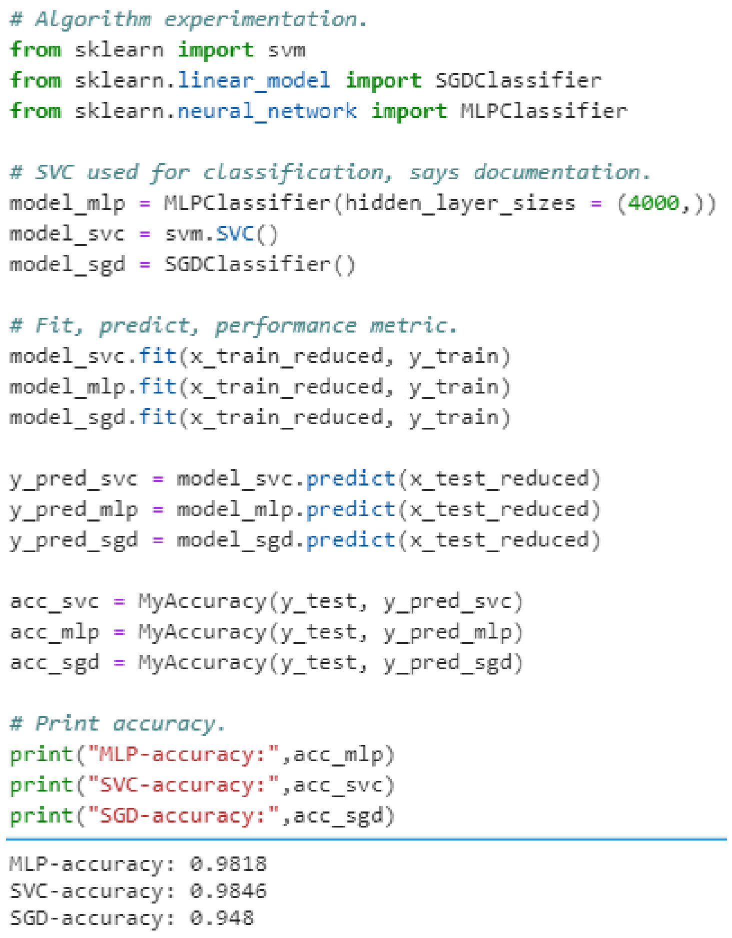
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Figur 14: Calculation of accuracy for binary classification.

Accuracy can be misleading in some cases, where the ground truth output of the classification is assymmetric. For example if training on 95 percent donuts and 5 percent houses, simply classifying every image as a donut will result in 95 percent accuracy. This sounds like a solid performance despite having a model with no actual predicitive power. However, this is not a problem with the symmetric QuickDraw! training and test data used.

Experimenting with the algorithms:
With limited computational-power and an relatively large dataset, training just one algorithm with one set of hyperparameters on the entire dataset takes a few hours to complete, even after PCA-assisted reduction.

In an attempt to get some idea of different learning models' performance on the data within a realistic timeframe, a smaller section of the data set is used for training. 15000 donuts and houses were shuffled and used for training, while 5000 houses and donuts. About 5 percent of the dataset is used. On figure 15 the experimentation is shown, training the three choosen algorithms on a smaller set of data.


```python
# Algorithm experimentation.
from sklearn import svm
from sklearn.linear_model import SGDClassifier
from sklearn.neural_network import MLPClassifier

# SVC used for classification, says documentation.
model_mlp = MLPClassifier(hidden_layer_sizes = (4000,))
model_svc = svm.SVC()
model_sgd = SGDClassifier()

# Fit, predict, performance metric.
model_svc.fit(x_train_reduced, y_train)
model_mlp.fit(x_train_reduced, y_train)
model_sgd.fit(x_train_reduced, y_train)

y_pred_svc = model_svc.predict(x_test_reduced)
y_pred_mlp = model_mlp.predict(x_test_reduced)
y_pred_sgd = model_sgd.predict(x_test_reduced)

acc_svc = MyAccuracy(y_test, y_pred_svc)
acc_mlp = MyAccuracy(y_test, y_pred_mlp)
acc_sgd = MyAccuracy(y_test, y_pred_sgd)

# Print accuracy.
print("MLP-accuracy:",acc_mlp)
print("SVC-accuracy:",acc_svc)
print("SGD-accuracy:",acc_sgd)
```

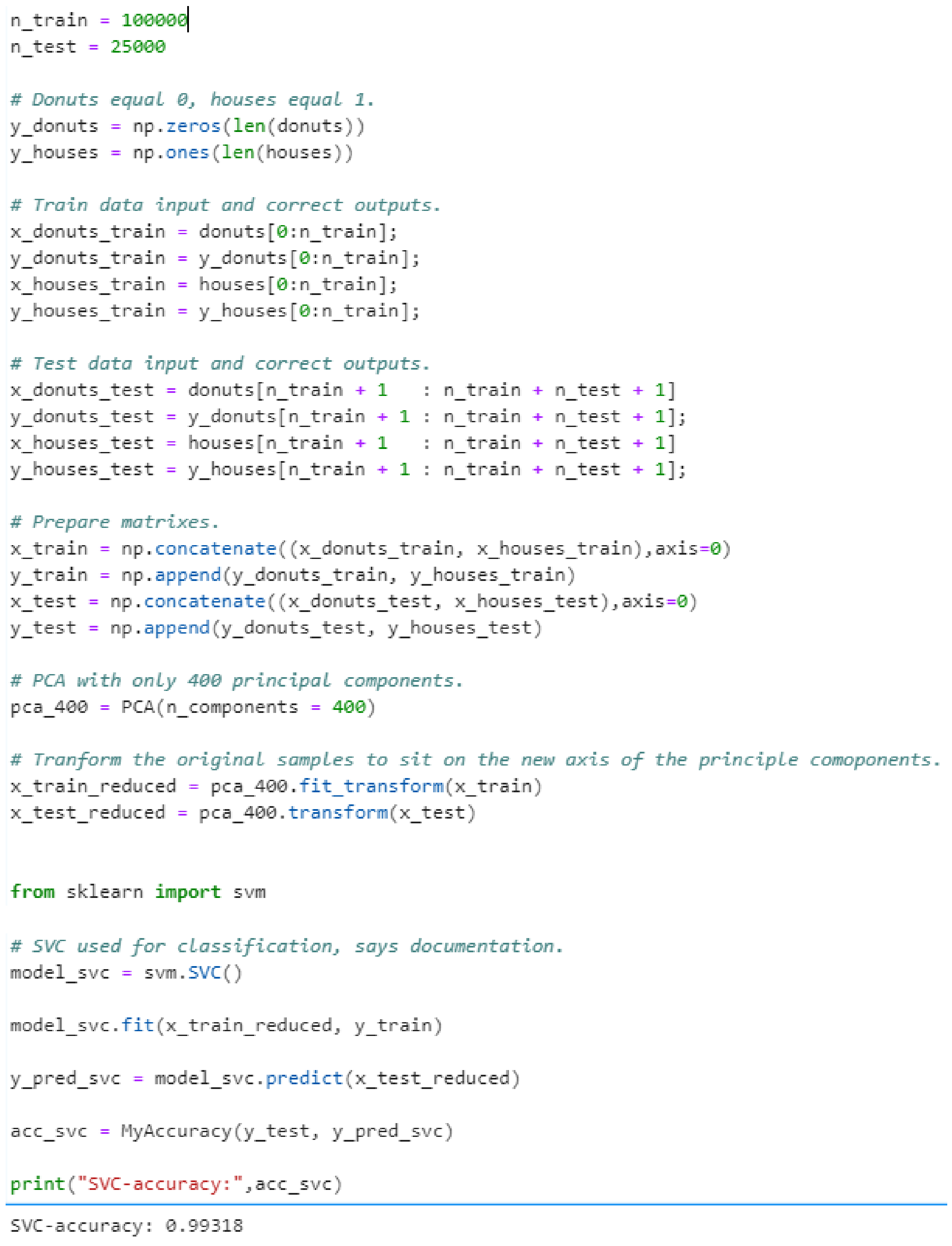
```
MLP-accuracy: 0.9818
SVC-accuracy: 0.9846
SGD-accuracy: 0.948
```

Figur 15: Trying out Stochastic Gradient Descent(SGD), C-Support Vector Classification (SVC) and Multi-Layered Perceptron (MLP) learning methods on a smaller section of the dataset.

From the experimentation three main things are concluded:

- The SVC takes the shortest time to train, while also achieving the best accuracy.
- The MLP-classifier takes a long time to train. Initially a hidden layer size of 400 was used, but resulted in poor accuracy on top of training taking several minutes. After increasing the hidden layer size to 4000, the training now takes about an hour, while still only training on around 5 percent of the entire dataset. This time however, the accuracy almost matched that of the SVC.
- The SGD-classifier is likely not right for the task. It is somewhat slow to train while also getting significantly worse accuracy in the experimentation phase.

Training on the entire dataset:
While the MLP-classifier is showing potential and might even perform the best out of the three algorithms when training on the entire dataset, it would likely have taken too long to train. SVC is instead chosen as the right algorithm for the situation. On figure 16 the training of the SVC on over 200000 samples each with 400 features is shown.

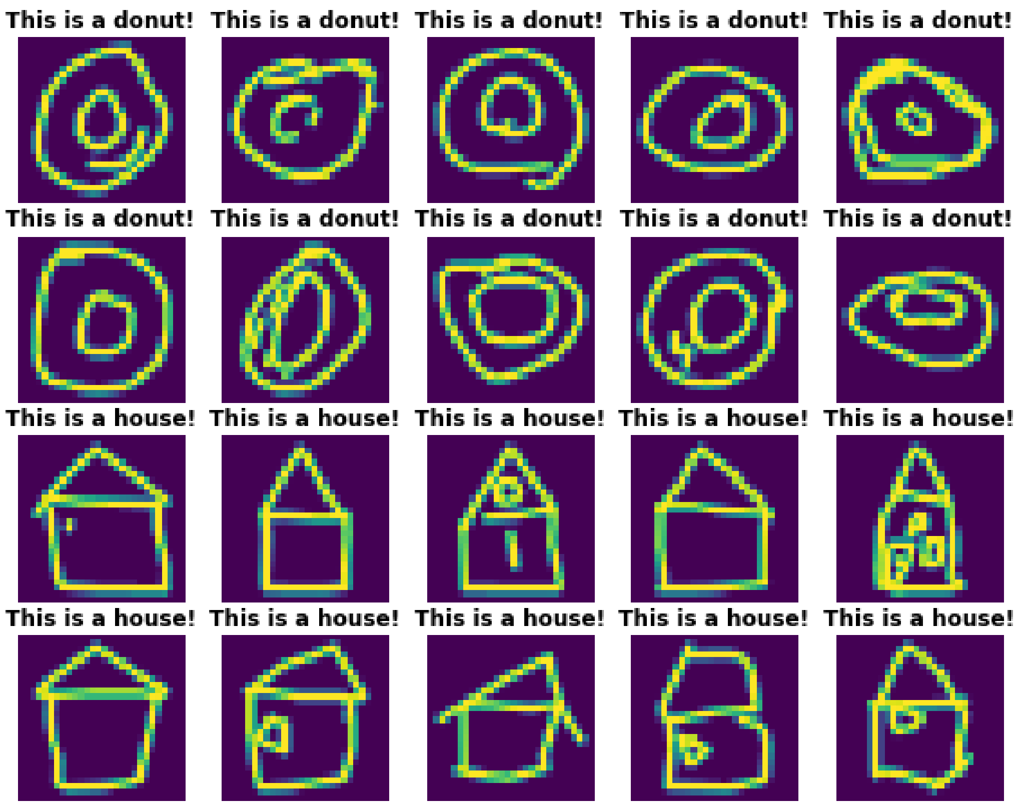


```
n_train = 100000
n_test = 25000

# Donuts equal 0, houses equal 1.
y_donuts = np.zeros(len(donuts))
y_houses = np.ones(len(houses))

# Train data input and correct outputs.
x_donuts_train = donuts[0:n_train];
y_donuts_train = y_donuts[0:n_train];
x_houses_train = houses[0:n_train];
y_houses_train = y_houses[0:n_train];

# Test data input and correct outputs.
x_donuts_test = donuts[n_train + 1   : n_train + n_test + 1]
y_donuts_test = y_donuts[n_train + 1 : n_train + n_test + 1];
x_houses_test = houses[n_train + 1   : n_train + n_test + 1]
y_houses_test = y_houses[n_train + 1 : n_train + n_test + 1];

# Prepare matrixes.
x_train = np.concatenate((x_donuts_train, x_houses_train),axis=0)
y_train = np.append(y_donuts_train, y_houses_train)
x_test = np.concatenate((x_donuts_test, x_houses_test),axis=0)
y_test = np.append(y_donuts_test, y_houses_test)

# PCA with only 400 principal components.
pca_400 = PCA(n_components = 400)

# Tranform the original samples to sit on the new axis of the principle comoponents.
x_train_reduced = pca_400.fit_transform(x_train)
x_test_reduced = pca_400.transform(x_test)


from sklearn import svm

# SVC used for classification, says documentation.
model_svc = svm.SVC()

model_svc.fit(x_train_reduced, y_train)

y_pred_svc = model_svc.predict(x_test_reduced)

acc_svc = MyAccuracy(y_test, y_pred_svc)

print("SVC-accuracy:",acc_svc)
```

```
SVC-accuracy: 0.99318
```

Figur 16: Training the SVC algorithm on the 200000 samples and testing it on a testing set of 50000 samples. Reaching accuracy of 99.3 percent.

A function was created to show a grid of the samples with the models prediction written above it. In 99.3 percent of cases, the model is correct! A cutout of the grid can be seen on figure 17.



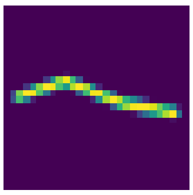
Figur 17: Training the SVC algorithm on the 200000 samples and testing it on a testing set of 50000 samples. Reaching accuracy of 99.3 percent.

10 Optimering og forbedringer

Applying image processing
To properly predict on image data, all the images should be processed to give meaningful context to the image. As an example comparing donuts and houses would be easier if the predicting algorithm knows whether the image is circular or square. In this solution only each pixel position is compared to new data, instead first understanding how the pixels are positioned next to each other, and then predicting could result in a much better solution.

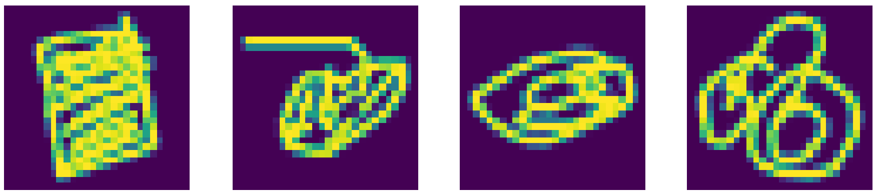
Additional hyperparameter search
We tried applying gridsearch with different levels of regulization to see of the model was overfit, underfit or balanced. However, this process does not complete within hours, even after decreasing dataset size and search space.

Data clearning
Not all data in dataset is equally useful for training. Many doodles are simply a line that has nowhere near enough information to be identifiable. We suspect these samples are included when the Google AI randomly guesses correctly at the start of a drawing.



Figur 18: A line drawing. Possibly a roof was recognized, but this should not be included for training.

Other drawings are from QuickDraw!-players not taking the doodling serious, but the AI guessing it anyway, perhaps somewhat randomly. On figure



Figur 19: Very bad drawings of donuts.

Unsupervised Learning
It would be interesting to see if was possible to detect the and categorize the two different doodled object without labels. Perhaps the houses and donuts would form seperate clusters, if the data is looked at the right way.

11 Conclusion

Using an SVC-training algorithm, a model is now able to classify doodles as either a donut or a house with over 99 percent accuracy. The training of the algorithm took about an hour training on 200.000 donuts and houses and its performance was evaluted based on a seperate test dataset of 50.000.

The use of the entirety of the dataset is made possible by first simplifying the data with the help of Principle Component Analysis (PCA), where the number of dimensions features is reduced from 784 to only 400 while keeping 99 percent variance. A classification using an MLP might result in an even better model, but training times quickly becomes too long as more and more of the dataset is used.