1 Introduction

In the following I will describe my work on the first project or the class in Machine Learning. The project deals with image recognition as well as detection and is separated into the following aspects: Creating training and testing data from the FDDB dataset and extracting HOG-features from this data, creating and training various models, face detection based on these models and finally feature visualization.

Data creation and feature extraction

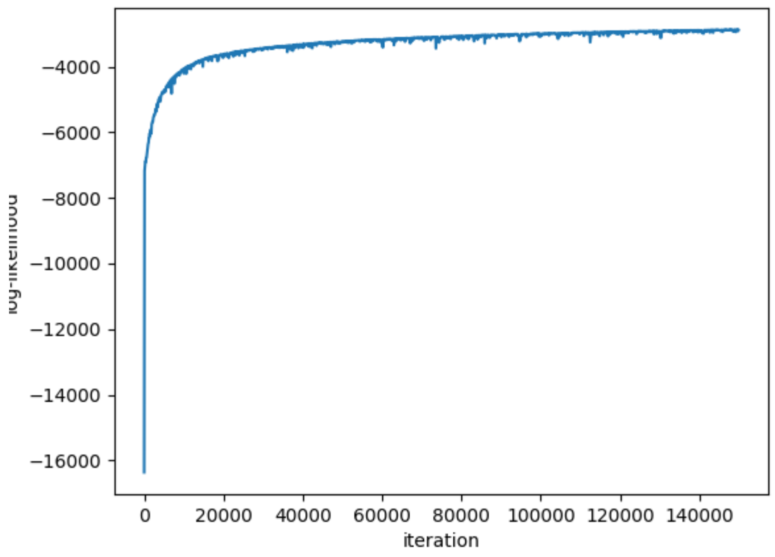
The data creation process took the following main steps: Reading information about the location of faces in the FDDB dataset, cropping a wide box around the faces, using a sliding window to crop positive (with the face centered) and negative (with the face not centered) datapoints. Then, HOG-features were extracted from these images and stored locally to allow training of the algorithms.
The provided dataset was the FDDB dataset, which stands for Face Detection Data Set and Benchmark. It contains images with one or more faces per image. The annotations contain information for every picture about the location of the pictures, as well as the width, height and angle. Originally, this information was meant to be used to create ellipses that exactly fit the face, but for the purposes of the project, rectangular boxes were used. Every sample was cropped and resized to a box of 96 x 96 pixels. Since training a machine learning model requires positive as well as negative samples, every face was used for 9 samples: 1 positive one and 8 negative ones. The former with the face centered, the latter using a sliding window technique, sliding 16 pixels laterally and/or vertically. The result was then again resized to a size of 96 x 96 pixels. The specifics, though time consuming, are not very interesting and not really related to the topics of the course so I will omit any code. If interested it can be found in the "read_data.py" file.
The resulting images are of course not well interpretable for linear models, making a feature extraction step necessary. For this, the HOG-Algorithm was used. A simple to use implementation is provided by the Python library scikit-images. The resulting feature vectors were, together with the corresponding labels 1,-1 saved in a pandas DataFrame for easy access and written to a csv file.

2 Logistic Regression

One of the most popular classification models is logistic regression. It is similar to linear regression in that it assigns a weight to each feature (plus usually an intercept). Unlike linear regression, which predicts nominal features, logistic regression predicts a probability P(y=k—x). This is achieved through the sigmoid function, which puts out values between 0 and 1. Also unlike linear regression, logistic regression can not easily be calibrated through solving a linear equation. Rather, one commonly uses the maximum likelihood algorithm. The log likelihood function is convex and can be maximized

through gradient ascent methods. It can also be proven that the log likelihood function for logistic regression has one maximum, so one need not worry about local minima. In this case, two different methods were employed for fitting the model: Stochastic Gradient Descent and Langevin Dynamics Stochastic Gradient Descent.
Stochastic Gradient Descent is the most simple method of gradient descent. One sample is randomly selected, the log likelihood is derived for this sample and the current weights. Then, a small step is taken into the direction of the gradient. This method converges to a local extremum (in this case also the global extremum), though not smoothly. The code for my implementation Stochastic Gradient Descent and the graph of the log-likelihood is shown below.

The accuracy achieved was 95.8%



The implementation of the stochastic gradient descent algorithm is displayed below:

```
w_ = np.ndarray((x.shape[1],))
       w_.fill(1 / x.shape[1])

       for it in range(iterations):
           i = random.randint(0, n - 1)
           grad = self.gradient(w_, x[i], y[i])
           w_ += r * grad
           if (it % graph_steps == 0):
               #if one wants to see the graph of the log likelihod
               # likelihoods.append(self.likelihood(w_, x, y))
       self.w =  w_
```

Langevin Dynamics Gradient Descent works similarly. The weight updating works slightly differently though. At every step, some Gaussian noise is added, the derivative of the log likelihood of a whole batch plus a prior is taken and the learning rate (as well as the amount of Gaussian noise) at each step decreases.

```
for t in range(iterations):
```

```
learning_rate = 0.05
sample = x_df.sample(sample_n)
sample_x = sample.drop(columns=["y"])
sample_y = sample["y"]
x = sample_x.values
y = sample_y.values
noise_shape = (901,)
e_t = learning_rate ** 2
noise = np.random.normal(np.zeros(shape=noise_shape), np.full(shape=noise_sha
w_t = (e_t / 2) * ((data_n / sample_n) * self.gradient(w_t, x, y) + noise )
learning_rate -= 0.001
```

Unfortunately, the likelihoods did not converge in my implementation so I decided standard Stochastic Gradient Descent was the better option.

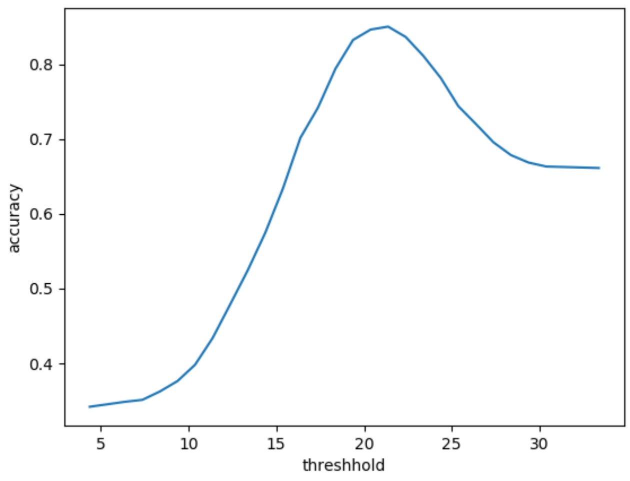
3 Fisher Model

The next model implemented is the Fisher Model. This model works by projecting the data into one dimension so that linear discrimination is maximized, meaning that inter-class variance is maximized, while intra-class variance is minimized.

$$w = (\Sigma_0 + \Sigma_1)^{-}1 \cdot (\mu_0 + \mu_1) \tag{1}$$

This vector w can easily be computed using methods offered by the NumPy library to find the covariance matrix and the mean vectors of both matrices.
Now one can simply multiply each feature vector with the projection vector, with the result being a scalar. The class is then chosen based on a threshold. There is no set rule for said threshold. To find the optimal value, I simply averaged the mean of both classes multiplied with the projection matrix. Then, values around this average were tested and the accuracy was plotted. It turns out the optimal result is achieved for a threshold of around 21. The resulting accuracy on the testing data was 95/

Below is the accuracy for various thresholds:



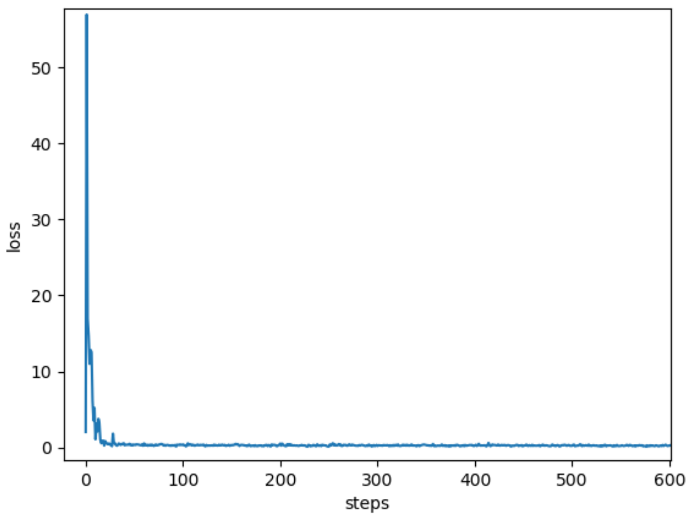
4 Support Vector Machine

A support vector machine operates on the basic idea of finding the optimally separating hyperplane between the two classes, meaning the hyperplane with the maximum margin, meaning the highest distance from the support vectors (difficult to classify data points). For the implementation of this model, using off he shelve code was permitted, so the implementation was relatively quick and straightforward. The popular scikit-learn library offers a support vector classifier as part of its svm module.
As per the instructions, two kernels were tried: A linear kernel and a radial basis function kernel.
The accuracy on the test data using the radial basis function kernel was around 92%, while using the linear kernel achieved 95%. Thus, for the final image detection part, the linear kernel would be used.

5 Convolutional Neural Net

Finally, the last and most complex model is a convolutional neural net. This is a special kind of neural net which is differentiated by its architecture. Instead of featuring only densely connected layers which lead to extremely many connections and exploding complexity in training, a convolutional neural net uses these fully connected layers only as the very final layers. The earlier hidden layers have two main components: convolutions and pools. A convolution uses a smaller filter to extract features from pixels which are next to each other, using a sliding window principle. The intuition behind this is that a pixel in a picture forms a feature only with other pixels which are next to it (say a hundred pixels of a certain configuration in ellipsoid shape and brown, white and black color form an eye). There is of course statistical relevance between pixels even far away, but this is better modeled at a feature level the relationship of the pixels of one eye to the pixels of another eye can be modeled through the relationship of the two eyes. The output of a convolutional layer are the features of each each filter.
The second important component is the pooling layer. This is used to reduce the tensors in size, though not changing the depth. Here, again a sliding window is used which maps the multiple values in the filter to one value usually the max value but the mean or median is also possible.
Due to the limits of the hardware available to me (a laptop with a Pentium CPU, 4 GB or RAM and no GPU), I was not able to extensively test different configurations. A first experiment with two layers achieved 89% accuracy on the testing data. The second and also final iteration was achieved through adding a hidden layer and also normalizing the inputs before the first layer. The resulting accuracy was 95%. Testing more different architectures and hyperparameters would of course have been desirable, but with the training running multiple hours and 95% being a very respectable result, I decided against it.

The final net features a batch normalization layer, a two dimensional convolution layer with 32 output channels, a max-pooling layer of kernel size 2x2, a convolution layer of 64 output channels, another max-pooling layer of kernel size 2x2, followed by a last convolution layer and again a max-pool layer-



6 Image detection

The last step of the project is using the image classification capabilities on which the models are trained to implement image detection. The method for this is fairly simple: Calculate all possible positions at which a face the object being detected could possibly be, and then use the image classification models to decide whether there actually is a face or not.

Unfortunately, even a combination of the models still had a very high false positive rate, meaning that in a try on a test run with only one face, 47 faces were detected. So of that is to be expected since the real face fits many possible bounding boxes at different positions and scales (though steps in scale and position were skipped), but overall the problem remains and means that, while the image classification was achieved, the detection part was left lacking.

7 Summary

The most effective methods were the Fisher Linear Discriminant Analysis, Logistic Regression, a Support Vector Machine using a linear kernel and the Convolutional Neural Net. The latter could most likely be optimized to a much higher level.

A big final issue is that, even with the models being fairly accurate, too many bounding boxes are found, making it unfeasible to use this as an image detection algorithm and display one bounding box around the face in the picture.