TDT4173 - Assignment 1

Håkon Ødegård Løvdal

January 2016

1 Theory

Basic Concepts

1.

A facial analysis problem

- Task T: analysis of faces
- Performance measure P: percentage of correct facial features recognized
- Training experience E: a training set of faces with given classifications

An automatic email-subject creator

- Task T: create email-subjects based on email-content
- Task P: average percentage of reasonable subjects generated
- Task E: database of emails, with human-given labels

2.

An inductive bias of a learning algorithm, L, is the minimal set of assumptions, B, such that L will be able to predict a result for any unknown input x_i, given the training examples, D_c.

Inductive bias is important in machine learning, because if a learning algorithm does not implement a inductive bias, it will not be able to classify new instances. An example of this is the `ROTE-LEARNER`. Due to its lack of a inductive bias, the classification process for new instances is purely an deduction of observed training examples. The `CANDIDATE-ELIMINATION` and `FIND-S` algorithms use inductive biases for classifying new instances.

The `CANDIDATE-ELIMINATION` has a inductive bias that represent a categorical restriction of the hypothesis, a *restriction bias*. It searches this hypothesis space completely to find every hypothesis that is consistent with the training data. The `ID3`-algorithm on the other hand, has no restriction bias, but a *preference bias* instead. It prefers hypothesises with the largest information gain. Information gain is given from the expected reduction in entropy caused by the partitioning of the examples.

3.

A hypothesis is said to *overfit* the training data if there exists another hypothesis has higher error on the training data, but actually performs better on the entire distribution of data.

The learners bias helps it predict a correct classification for input not represented in the training data. When the learning is performed too long, the learner may learn specific features of the training data and increase the overall performance on it, while performance on unknown data will decrease. The features learned has nothing to do with the actual concept. The use of a bias is introduced to reduce this effect (overfitting).

$$error_t(h) < error_t(h^{'})$$
$$\text{and}$$
$$error_D(h) > error_D(h^{'})$$

1.1 Concept Learning

1.

Initial state:
$S_0 : \{< \emptyset, \emptyset, \emptyset, \emptyset, \emptyset >\}$
$G_0 : \{<?, ?, ?, ?, ? >\}$

After training 1:
$S_1 : \{< Hair, Live, False, False, Flat >\}$
$G_1 : \{<?, ?, ?, ?, ? >\}$

After training 2:
$S_2 = S_1$
$G_2 : \{< Hair, ?, ?, ?, ? >, <?, Live, ?, ?, ? >, <?, ?, False, ?, ? >, <?, ?, ?, False, ? >$
$, <?, ?, ?, ?, Flat >\}$

After training 3:
$S_3 : \{< Hair, Live, False, False, ? >\}$
$G_3 : \{< Hair, ?, ?, ?, ? >, <?, Live, ?, ?, ? >, <?, ?, False, ?, ? >, <?, ?, ?, False, ? >$
$\}$

2.

The first example is classified as `True`, since is more specific than the most specific boundary in the version space. Example two and three are somewhere in between the boundary of the most specific and most general, and as a result it is not possible to classify them as either positive or negative.

1. $\{< Hair, Live, False, False, None >\} \rightarrow True$

2. $\{< Feathers, Egg, False, True, Pointed >\} \rightarrow ?$
3. $\{< Scales, Egg, True, False, Flat >\} \rightarrow ?$

3.

The system should choose a strategy that makes it ask for a training example that reduces exactly half of the hypotheses in the current version space. This means that if the learner classifies it as positive, S is generalised, or as negative, G is specialised. No matter what, the version space is reduced by half. An example of such a training example is: $\{< Hair, Live, True, False, None >\}$

2 Programming

1.

The code-implementation of linear regression using gradient descent is included in the folder `code.zip`. The code is written in Python3, using the following requirements (all of which could be installed using `pip3`):

- numpy
- matplotlib
- pandas

In order to run the code, run `regression.py`. This will execute the code using my final configurations. Should you want to change these, edit the parameters in the `BASIC_CONFIG` located at the top of `regression.py`. After the code has completed, a plot of the loss function during the epochs and a model of the hyperplane will be available in the folder `figures`.

2.

a)

Initial parameters:

W: `np.random.uniform(size=feature_count)`, where feature_count = p = 2

b: 1

α: 0.5

The initial weights are randomly selected by *numpy*. Choosing optimal weights is a hard problem, and is often a result of trial and error. In order to make the gradient-descent algorithm learn properly, I chose to randomly set the weights to different values between [.00001 and 1.0]. Hopefully, this would in most cases produce weights that require less epochs to converge than manually set weights.

The bias is used to push the decision boundary and therefore make the hyperplane be able to separate unseen inputs. I tried multiple biases in the range [0, 1], but it did not seem to affect the overall result that much. In the end, I chose to use 1 as bias, since it always produced acceptable results, and is often used as an example representing the x_0/θ.

The learning rate, α, is also a result of trial and error. To find the optimal learning rate is a difficult task. I tried different learning rates in the interval [0.0001, 0.5] and 0.5 seemed to yield the best result during these tests. Choosing a too small learning rate would make the learning algorithm use a long time to find the minimal error. If it is too large, the weight updating will oscillate, and not learn features in an optimal way. So in order to avoid this oscillation/overfitting, 0.5 seemed to be a preferable learning rate.

b)

```
# Learning rate: 0.5,
  Maximum epochs: 1500
  Convergence Threshold: 1e-06
# Initial weights and bias: W[ 0.76625467  0.39110352] B: 1

# Intermediate result of loss function after 5 epochs:
    [ 0.01507546] / [ 0.02023] (training/test)
# Intermediate result of loss function after 10 epochs:
    [ 0.01333152] / [ 0.01804244] (training/test)
```

c)

```
# Convergence reached after 130 epochs
# Weights and bias after training:
    W[ 0.38024844  0.53079818], B[ 0.06966714]
# Error:
    [ 0.00800273] / [ 0.00995515] (training/test)
```

Initially, the algorithm seems to start adjusting the weights drastically during the first couple of epochs. After approximately 80 epochs, it continues to adjust the weights very sensitively. When reaching convergence (with a threshold of 10^{-6}) the total error on the training set ends up at 0.008. When testing the test-set, it produces an error of approx. 0.01. After all my testing, I find these results acceptable. The algorithm seems not to have overfitted on the training-set. Furthermore, when visualising the hyperplane among the original data-points of the training-set in 3D, the plane seems to have fitted the data very well.

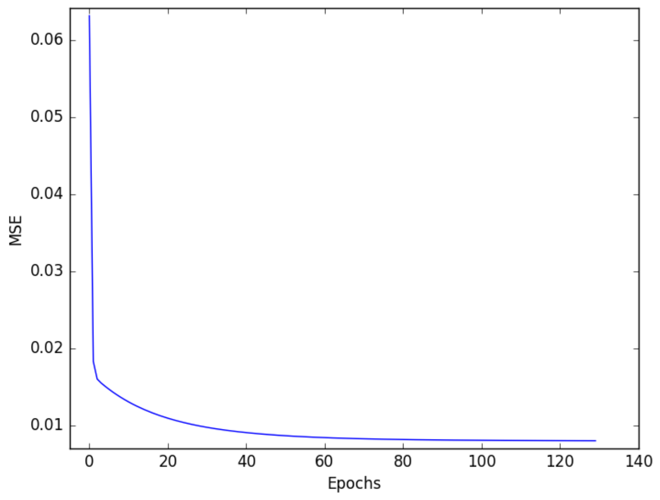

Figure 1: Loss function during the course of optimisation

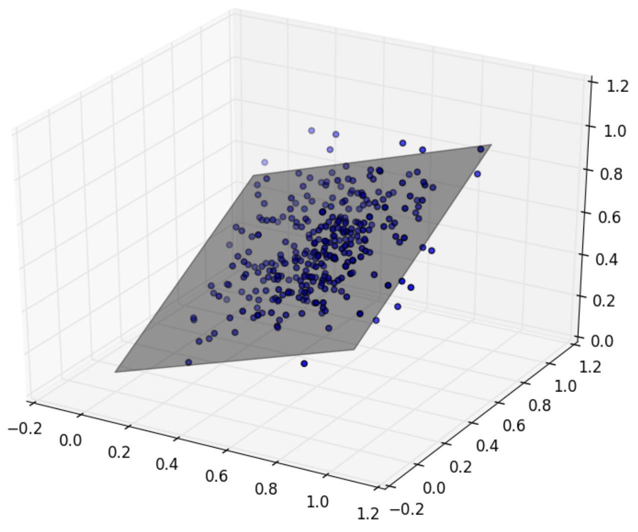

Figure 2: Fitted hyperplane