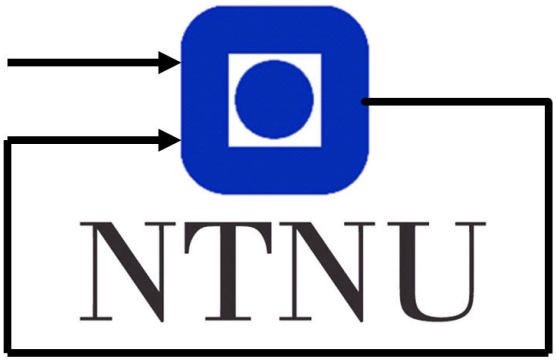
Assignment 1 - in TDT4173
Machine Learning and Casebased Reasoning

Ella-Lovise H. Rørvik – 759559

Fall 2017



Department of Engineering Cybernetics

Contents

1 Theory

1.1 Theory: Exercise 1): Concept Learning

Concept learning consists of inferring the general definition of some concept, given labelling of data from members and non-members of the concept. The definition of Concept Learning from [Mit97] is: "Inferring a boolean-valued function from training examples of is input and output."

Generally any concept learning task can be described by:

- a set of instances over which the target function is defined
- the target function
- the set of candidate hypothesis considered by the learner
- the set of available training examples

An example of concept learning task is the task of learning the target concept: "days Ella enjoys eating an apple". In this example, the task is to train a boolean-valued function to predict if Ella enjoys eating an apple on a particular day. This decision will be based on specific attributes. Attributes are features, which are used to decide whether a data instance is positive, for a particular concept. In this case attributes used to decide if Ella enjoys to eat apple on a given day could be weather, temperature, activity level, and of course if she enjoys eating apple on this day. The attribute that indicates if Ella likes to eat apple on a particular day, is then "enjoys eating apple"

This attribute will be the result of the boolean-valued function. In concept learning you use selected data about certain attributes, to construct the target concept. An example of a training data set is represented in table 1.

Table 1: Example of training set

Activity level	**Weather**	**Temperature**	**Enjoys eating apple**
High	Sun	High	Yes
Low	Sky	Low	No
High	Sky	Medium	Yes

In concept learning, the learner is presented for a number of hypothesis representations. En example of a simple representation of hypothesis is a conjunction of constraints on the instance attributes. For this example a hypothesis on this simple form could be a vector with 3 constraints, specifying the values of the 3 attributes: Activity level, weather and temperature.

Each attribute in the hypothesis will either be:

- "?" : that any value is acceptable for this attribute
- value : a specific value for the attribute
- $\emptyset$: no value acceptable

If for instance a day x satisfies all the constraints of the hypothesis h, then h(x) = 1, meaning that h classifies x as a positive sample. Using the training data we want to find

an expression for the hypothesis such that we classify the examples correctly, and hopefully also other data.

The most general hypothesis,that every day is a positive example, is : $< ? , ? , ? >$. The most specific possible hypothesis, that no day is positive example, is represented by : $<\emptyset, \emptyset, \emptyset>$. A example of a hypothesis is that Ella enjoys eating apple on days with a high activity level, that weather is not important and temperature are not important, meaning $<$ High, ?, ?,$>$.

1.2 Theory: Exercise 2): Function approximation

Function approximation is the process of learning the target function.

The ideal target function V describes the system/process perfectly. Function approximation is used to discover the operational description of the ideal target function V, since the ideal target function is a nonoperational definition. The approximated function $\hat{V}$, is a description which can be used to evaluate states and make decision based on the states. This approximation is found through analysis of training data, [Mit97].

1.3 Theory: Exercise 3): Inductive bias

Inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances, [Mit97].

The more theoretical way of defining the inductive bias from [Mit97], is:

Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = (x, c(x)$ *be an arbitrary set of training examples of c. Let* $L(x_i, D_c)$ *denote the classification assigned to the instances* x_i *by L after training on the data* D_c *The inductive bias of L is any minimal set of assertions B such that for any target concept c and corresponding training examples* D_c

$$(\forall x_i \in X)[B \wedge D_c \wedge x_i) \vdash L(x_i, D_c]) \tag{1}$$

Inductive bias enables target function to generalise beyond the training data. Stronger inductive bias classification leads to more classifications of instances than algorithms with weaker inductive bias. The correctness of such classifications will depend completely on the correctness of this inductive bias. More strongly biased methods make more inductive leaps, classifying a greater proportion of unseen instances.

Viewing inductive bias of a system provides a non procedural means of characterising a systems policy for generalising beyond the observed data. Knowing about the inductive bias, you can also compare different learners according to the strength of the inductive bias they employ.

The Candidate-Elimination algorithm make the assumption that the target concept could be represented by a conjunction of attribute values. Candidate-Elimination are biased by the implicit assumption that the target concept can be represented by a conjunction of attribute values. In cases where these assumptions are not correct, the Candidate-Elimination algorithm will misclassify at lest some instances from X. The inductive bias

of the Candidate-Elimination algorithm is characterised this way in [Mit97]: "The target concept c is contained in the given hypothesis space H."

The inductive bias in decision tree learning are harder to define than candidate elimination because it is heuristic search. In the formulation of the the basic decision tree learning algorithm, ID3, inductive bias is characterised in [Mit97] as:

Shorter trees are preferred over longer trees. Threes that place high information gain attributes close to the root are preferred over those that do not.

1.4 Theory: Exercise 4): Overfitting and underfitting

A hypothesis overfits the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances, including instances other than the original validation and training set. Overfitting therefore means that the model learns the detail and noise of the training data to the extent that it negatively impacts the performance of the model on new data. This problems impacts the models ability to generalise in a negative way, [Mit97].

The official definition of overfitting from [Mit97] is: "Given a hypothesis space H, a hypothesis $h \in H$ is said to overfit the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution instances."

Underfitting is that a hypothesis/model can neither model the training data nor generalise to new data. If the machine learning model if underfitted it will not give a suitable hypothesis, and give poor performance on training data and test data.

Validation sets are sets used to evaluate the accuracy of a hypothesis over instances, and to evaluate the impact of pruning the hypothesis, [Mit97].

Cross-validation is used to assess how a model generalises to an independent data set, the validation set. Cross-validation can mitigate overfitting by letting the algorithm during training phase monitor the development of the model-error with respect to both the validation set and training set, while using the training set to produce the model. Overfitting starts when the error of the validation set starts to increase, while the error of the training set decreases during the training phase. We want a result such that the error from both the validation set and training set are as low as possible, meaning the system generalises well. So if overfitting happens, you can go back to a iteration with satisfying low error of the validation and training set.

1.5 Theory: Exercise 5): Candidate elimination

Before starting the candidate elimination the boundary sets of the version space needs to be initialised. Initially the most general hypothesis, G_0 and the most specific hypothesis, S_0, are :

$\mathbf{S}_0 = < \emptyset, \emptyset, \emptyset, \emptyset >$, $\mathbf{G}_0 = <?, ?, ?, ? >$

Iteration 1: Training data 1: < Female, Back, Medium, Medium>, Treatment Successful = Yes

Training data 1 implies that the specific hypothesis are overly specific, and fails to cover the positive example. Therefore the boundary is revised to the least more general general hypothesis, that covers the new example. Since this was a positive example and G_0 covers this example, the G boundary is not changed.

$\mathbf{S}_1 =$ < Female, Back, Medium, Medium> , $\mathbf{G}_1 =$<?, ?, ?, ? >

Iteration 2: Training data 2: < Female, Neck, Medium, High>, Treatment Successful = Yes

Training data 2 are also positive, and again the hypothesis are overly specific, and fails to cover the positive example. This means that two boundaries are revised to be more general, meaning "Activity Level" = "Sleep Quality" = ?. Since this is a positive example, and G_1 covers the positive example, the G boundary is not changed.

$\mathbf{S}_2 =$ < Female, ? , Medium, ?> , $\mathbf{G}_2 =$<?, ?, ?, ? >

Iteration 3: Training data 3:< Female, Shoulder, Low, Low>, Treatment Successful = No

Training data 3 are negative, and as a result the G boundary of the version space are overly general. To specialise G we see that the only attribute which correctly classifies the training example as negative, and are consistent with the previously positive examples, are the attribute "Activity Level". So for G_2 the attribute "Activity Level" are changed to medium.

$\mathbf{S}_3 =$ < Female, ? , Medium, ?> , $\mathbf{G}_3 =$ <?, ?, Medium,?>

Iteration 4: Training data 4: < Male, Neck, High, Medium>, Treatment Successful = Yes

Training data 4 are also positive, but are not consistent with the specific hypothesis S_3. This leads to generalisation of S_3 to S_4. Since attribute 2 and 3 are not consistent with the training example, we get S_4 only consisting of generalised attribute values.

The general hypothesis also needs to be updated such that it is consistent with the training data, leading to G_4 consisting of only generalised attribute values. By definition of G and S, G_4 can not be more specific than S_4.

$\mathbf{S}_4 =$ < ?, ? , ?, ?> , $\mathbf{G}_4 =$ <?, ?, ?, ?>

Iteration 5: Training data 5: < Male, Back, Medium, Low>, Treatment Successful = Yes

The next training example are also positive, but since the specific hypothesis boundary from the last iteration only consists of general valued attributes, the boundaries are not updated in the 5'th iteration.

$\mathbf{S}_5 =$ < ?, ? , ?, ?> , $\mathbf{G}_5 =$ <?, ?, ?, ?>

Discussion of result of CE

Looking at the training data given in this assignment, there are no attributes in the positive training data which have the same values on the same attributes, leading to the problem of finding a specialised hypothesis which is not only consisting of generalised attributes.

Looking at the different iterations shown above, if we stopped before iteration 3, we would have a version space, since S != G, and a possibility of finding a target concept. Stopping

at iteration 3 means not including all the data points, which leads to the question if that would really give a good target concept. We may look at the inductive bias of Candidate-Elimination algorithm, and whether it could be possible to use boolean-function to try to predict if the treatment is successfully or not. However, given the data presented above and the complex question of predicting if a patients treatment is successful, I would say in this case that is not possible to use boolean-function to try to predict if the treatment is successfully or not.

2 Programming

The included zip-file includes the following folders: regression, classification and code, where the folders regression and classification contains data. In the folder named code, there are three files: `linear_regression.py`, `logistic_regression.py` and `utility.py`.

The file called `utility.py` contains the functions collecting the data of the files, and the functions which makes the variables $\mathbf{X}$, $\mathbf{x}$ and $\mathbf{y}$. The functions in the file `utility.py` are used in both the files `linear_regression.py` and `logistic_regression.py`. The file called `linear_regression.py` contains implementation of linear regression, error calculation and plotting functions for the necessary plots. This file also includes functions running the linear regression on the training and test data sets, plotting and printing the results.

The file called `logistic_regression.py` contains the implementation of logistic regression, calculates error and implementation of the necessary plotting functions. This file also includes functions running the logistic regression on the training and test data sets, plotting and printing the results.

2.1 Linear regression

2.1.1 Linear regression: Exercise 1): Implementation of OLS

The linear regression with ordinary least squares using the closed-form solution in equation (9) in the assignment are:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{2}$$

The implementation of equation (2) is implemented in the file `linear_regression.py` in the function `linear_regression_OLS(X,y)`. This function's implementation is also shown in the listing 1.

Listing 1: Calculation of weights with ordinary least square

```
1 def linear_regression_OLS(X,y):
2     X_t = np.transpose(X)
3     w = np.dot(np.linalg.pinv(np.dot(X_t,X)),np.dot(X_t,y))
4     return w
```

To find the weights $\mathbf{w}$, the vector $\mathbf{y}$ and $\mathbf{X}$ needed to be found. These vectors where found using the functions `create_X(data)` and `create_y(data)`, both functions in the file `utility.py`.

2.1.2 Linear regression: Exercise 2): Linear regression on 2d data set

Finding the weights and mean square error of the 2d training and test data set are performed in the function `linear_regression(train_file_name,test_file_name)` in file `linear_regression.py`, and show in listing 2.

Listing 2: Testing function for linear regression

```
1
2  def test_linear_regression(train_file_name,test_file_name,plot
       ):
3      training_data = get_data_from_file(train_file_name)
4      X = create_X(training_data);
5      y = create_y(training_data);
6      w = linear_regression_OLS(X,y)
7      print("Weights after training")
8      print(w)
9      print('Error training',error_mse(X,w,y))
10
11     test_data = get_data_from_file(test_file_name)
12     X = create_X(test_data);
13     y = create_y(test_data);
14     print('Error testing',error_mse(X,w,y))
15
16     if(plot == 1):
17         plot_result_linear_regression(test_data,training_data,
               w)
```

To find the mean square error the function `error_mse(X,w,y` was implemented using equation (5) in the assignment, in this equation:

$$E_{mse} = \frac{1}{N}||\mathbf{Xw} - \mathbf{y}||^2 \tag{3}$$

The implementation of the mean square error according to equation (3) is implemented in the function `error_mse(X,w,y)` in file `linear_regresssion.py`, and shown in listing 3.

Listing 3: Mean square error

```
1
2  def error_mse(X,w,y):
3      N = X.shape[0]
4      error = np.square(np.linalg.norm(X*w-y))/N
5      return error
```

The results of mean square error with the weights trained of the training set are:

- Training data set: $E_{mse} = 0.0103868507315$
- Test data set: $E_{mse} = 0.00952976445062$

The weights after the linear regression are $\mathbf{w} = [0.24079271, 0.48155686, 0.0586439]^\top$.

The mean square errors after the training are larger than the mean square error of the test data set. This implies that we don't have overfitting, and therefore have a more generalised model, able to predict outcome values for previously unseen data.

This exercise can be run be launching the file `linear_regression.py`, which starts the function `test_linear_regression(train_file_name,test_file_name)` testing this exercise.

2.1.3 Linear regression: Exercise 3): Linear regression on 1d data set

Finding the weights for training data set 1d, works in the same manner as in the previous assignment, using the same code for finding the weights and calculating the mean square error, i.e. running function `linear_regression(test_file_name,train_file_name)`.

The result of the mean square error with the weights trained from the training data set are:

- Training data set: $E_{mse} = 0.0137587911265$
- Test data set: $E_{mse} = 0.012442457462$

The weights resulting from the linear regression are $\mathbf{w} = [0.1955866, 0.61288795]^\top$. These weights where used to draw the fitted line from the linear regression, shown in figure 1, employing the definition of decision boundary $\mathbf{w}^\top \mathbf{X} = 0$, leads to the line $x_2(x_1) = -w_0/w_2 - x_1 w_1/w_2$. The plotting of the data points and drawing of the fitted line, are done using the function `plot_result_linear_regression(test_data,training_data,w)` in the file `linear_regression.py`.

In the figure 1, we see that the training data set are more spread than test data set, which may be an explanation of why the mean square error of the training data set are higher than the mean square error of the test data set.

Looking at the plot 1 it seams like the line fit well to both the training and test data set, meaning the linear regression was able to fit a hyperplane (in this case a line) relatively good to our data. This is also indicated by the relatively low square error of both test and the training data set, leading to the conclusion that the model is generalising pretty good.

This exercise can be executed be running the file `linear_regression.py`, which starts the function `test_linear_regression(train_file_name,test_file_name)`.

2.2 Logistic regression

2.2.1 Logistic regression: Exercise 1): Logistic regression on data set 1d

The update rule for the weights, using gradient descent for logistic regression was represented equation (20) in the assignment set, and are :

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \sum_{i=1}^{N} (\sigma(z) - y_i)\mathbf{x}_i \tag{4}$$

$\sigma(z)$ was defined in equation (10) in the assignment, and is:

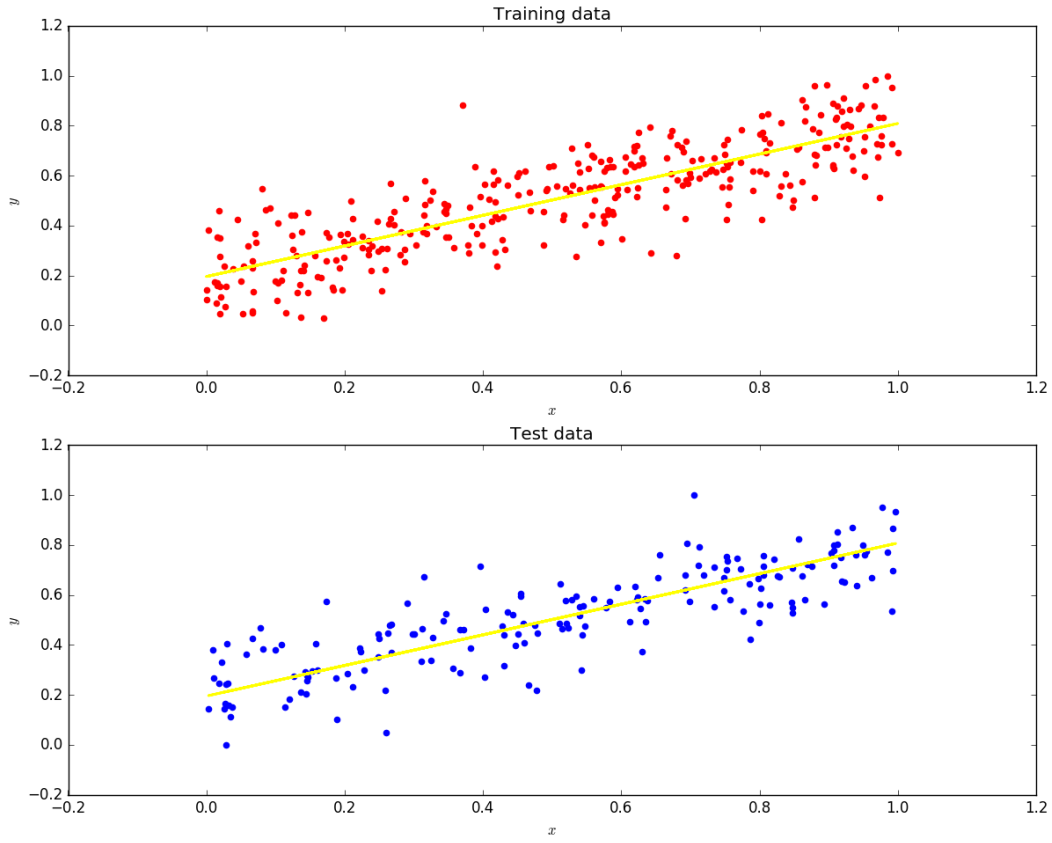


Figure 1: Result after linear regression of the 1d training data set, and tested on the 1d test data set. The red dots are the data from the training set, while the blue dots are the data from the test set. The yellow line is the learnt model, fitted line, of the training data set using OLS.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{5}$$

with $z = \mathbf{w}^\top \mathbf{X}$.

The implementation of update rule for gradient descent for logistic regression, utilising equation (5) and (4), are (respectively) the functions `sigma(w,X)` and `update_w(w,X,y,eta)`, as shown in listing 4. The functions are located in the file `logistic_regression.py`, .

Listing 4: Sigma function and update rule for gradient descent

```
1 def sigma(w,X): # equals y_hat = estimate of y
2     z = np.dot(np.transpose(w),X).item(0)
3     y_hat = 1/(1+np.exp(-z))
4     return y_hat
5
6 def update_w(w,X,y,eta):
7     temp = 0
```

```
    for i in range(0,len(y)):
        X_i = np.transpose(X[i,:])
        y_i = y.item(i)
        temp += (sigma(w,X_i) - y_i)*X_i
    w = w - eta*temp
    return w
```

`test_linear_logistic_regression(file_name_train,file_name_test)` is the function used to test the logistic regression on the first data set containing training and test data. This function is shown in 5. In this function first the variables $\mathbf{X}$ and $\mathbf{y}$ are found for the training data set. Subsequently, the weights are initialised to a 3x1 array with elements of value 0.1 and the learning rate is set to 0.1. After this the training process starts, using 1000 iterations. The value of initial weights and learning rate is found using trial and error, but with these value there is relatively small cross-entropy error for both the first training and test data set used in logistic regression. At the end of the function plots showing the data points (called in the code for property plot) and cross-entropy error are plotted. The plotting of the linear decision boundary is shown in function `property_plot_linear_separable(data,subplot_Num,fig_title,w,fig_num)` and the cross-entropy error are plotted from the function `plot_cross_entropy_error(error,label,fig_num)`.

Listing 5: Test function of logistic regression

```
def test_linear_logistic_regression(file_name_train,
   file_name_test):
    training_data = get_data_from_file(file_name_train)
    X_train = create_X(training_data);
    y_train = create_y(training_data);

    test_data = get_data_from_file(file_name_test)
    X_test = create_X(test_data);
    y_test = create_y(test_data);

    eta = 0.1 # learning rate #0.001
    w  = 0.1*np.matrix([[1],[1],[1]])
    error_train_array = [];
    error_test_array = [];
    error_train  = 0;
    error_test = 0;
    it_array = [];
    for i in range(0,1000):
        w = update_w(w,X_train,y_train,eta)

        error_train = cross_entropy_error(y_train,w,X_train)
        error_train_array.append(error_train);
        error_test = cross_entropy_error(y_test,w,X_test)
        error_test_array.append(error_test);

        it_array.append(i);

    print("The weights after training:")
```

```
    print(w)
    print("Error from train set",error_train)
    print("Error from test set",error_test)

    property_plot_linear_separable(training_data,211,"Plot of
       training set",w)
    property_plot_linear_separable(test_data,212, "Plot of
       test set",w)
    plot_cross_entropy_error(it_array,error_train_array,'train
        data')
    plot_cross_entropy_error(it_array,error_test_array,'test
       data')
    plt.show()
```

During training, the cross-entropy error was saved in an array for both the test and training data set, for later plotting of how the training fared. The cross-entropy error was calculated using equation (14) in the assignment, as follows:

$$E_{ce} = -\frac{1}{N}(\sum_{i=1}^{N} y_i \ln \sigma(z + (1 - y_i) \ln 1 - \sigma(z)) \tag{6}$$

The cross-entropy error implementation is implemented in function `cross_entropy_error(y,w,X)` in the file `logistic_regression.py`, and is shown in listing 6.

Listing 6: Cross-entropy error

```
def cross_entropy_error(y,w,X):
    N = y.shape[0];
    temp = 0
    for i in range(0,len(y)):
        y_i = y.item(i)
        X_i = np.transpose(X[i,:])
        y_hat = sigma(w,X_i)
        temp += y_i*np.log(y_hat) + (1-y_i)*np.log(1 - y_hat)
    error = -1*temp/N
    return error
```

With learning rate of 0.1, and start weights $[0.1, 0.1, 0.1]$ the final weights were: $\mathbf{w} = [9.82760004, -27.74498705, 14.32639528]^\top$ and final cross-entropy error became:

- Train data set : $E_{ce} = 0.0244773840199$
- Test data set : $E_{ce} = 0.0756390079948$

The plot showing the decision boundary with the weights listed above are shown in figure 2. This figure shows that we have one wrong classification of the test data set, with one blue data point on the same side as the red data points of the decision boundary. Looking at these figures makes it clear that the data points, of both the training and test data set, are linearly separable. This is due to the fact that it is possible to draw a straight line separating the data points labeled 0 and the other data points labeled 1.

From figure 2 it would seem that the model is generalised well, since it only wrongly classifies one test data point. If we also look at the figure showing the development of the cross-entropy error, figure 3, we see that as the number of iterations increase, the cross-entropy does not increase for the test data set (validation set), meaning that we do not have overfitting of the model. These observations clearly indicates that the model generalises appropriately.

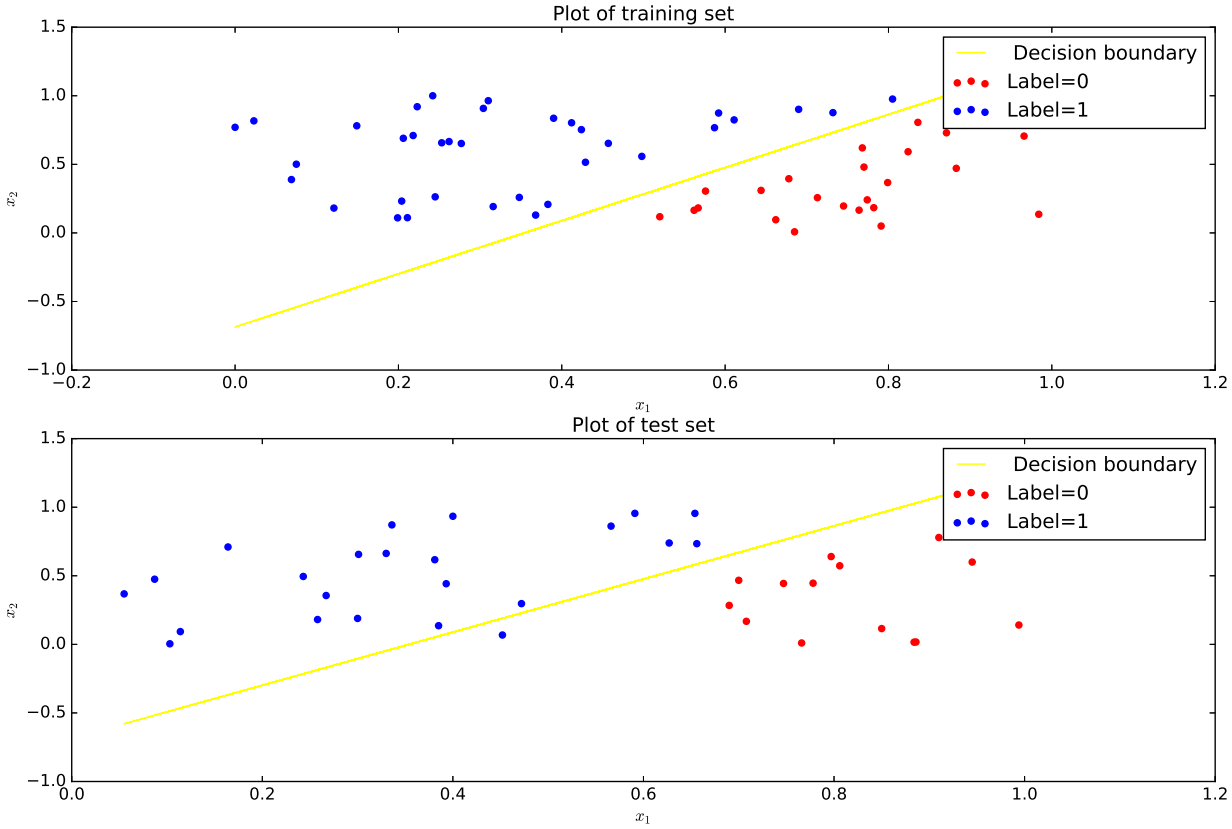


Figure 2: The figure shows the decision boundary placement for the test data set and training data set. The red data points are labeled with 0, and the blue data points labeled 1.

This exercise can be executed be running the file `logistic_regression.py`, which starts the function `test_linear_regression(train_file_name,test_file_name)`.

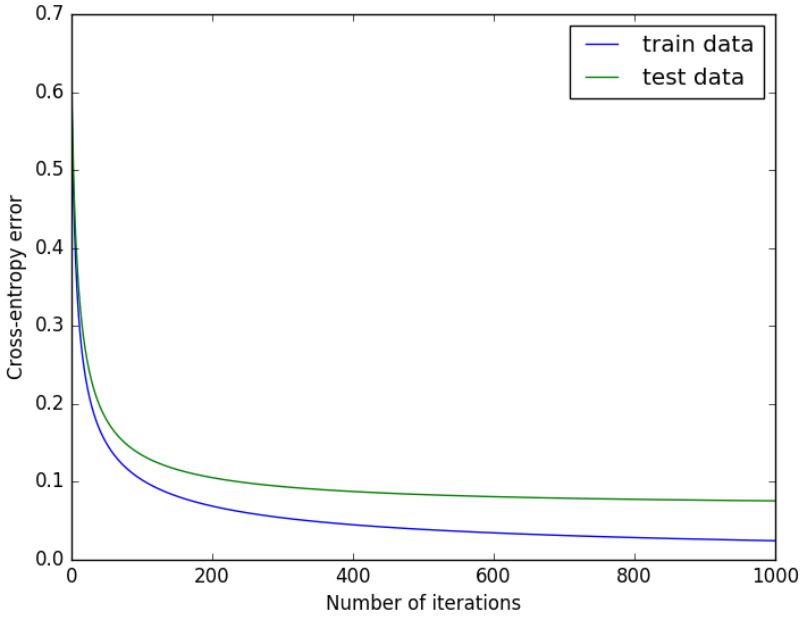


Figure 3: Figure showing the development of the cross-entropy error for both the training dataset and test dataset during the training phase of the model

2.2.2 Logistic regression: Exercise 2): Logistic regression on data set 2

Using `test_linear_logistic_regression(file_name_train,file_name_test)` logistic regression with input $\mathbf{X} = [1, x_1, x_2]^\top$ was preformed, using the same method as in the previous exercise. The results of the logistic regression is shown in figure 4. This figure shows that it is not possible to draw a line separating the different labeled data. Also looking at this figure we can see that they are separable, but with a elliptical boundary.

It is possible to do a transformation of $\mathbf{X}$, so that we will get a elliptical decision boundary. If $\mathbf{X}$ are transformed such that $\mathbf{X} = [1, x_1, x_2, x_1^2, x_2^2]^\top$, we will get the decision boundary $0 = w_0 + x_1 w_1 + x_2 w_2 + x_1^2 w_3 + x_2^2 w_4$, with $\mathbf{w} = [w_1, w_2, w_3, w_4]^\top$. When completing the square, the decision boundary can be written in the form of the elliptical equation, leading to the result:

$$1 = \frac{x_2 + {c_2}^2}{\frac{c}{w_4}} + \frac{x_1 + {c_1}^2}{\frac{c}{w_3}}, c = \frac{w_2^2}{w_4 4} + \frac{w_1^2}{w_2 4} - w_0 \tag{7}$$

This is an ellipse with radiuses $r_2 = \sqrt{c/w_2}$, $r_1 = \sqrt{c/w_3}$ and center of the ellipse equals $(c_1, c_2) = (w_2/w_4/2, w_1/w_3/2)$.

`test_elliptical_logistic_regression(file_name_train,file_name_test)` is the function used to test the logistic regression with input $\mathbf{X} = [1, x_1, x_2, x_1^2, x_2^2]^\top$. The test function is relatively similar to the code shown in 5. One of the exception is that the new test function uses the input $\mathbf{X} = [1, x_1, x_2, x_1^2, x_2^2]^\top$, using the function `create_elliptical_X(data)` to create $\mathbf{X}$. Another difference is that weights $\mathbf{w}$ are a 5x1 array, and $\mathbf{w}$ are initialised to $[0.1, 0.1, 0.1, 0.1, 0.1]^\top$. The learning rate are still set to 0.1. There are also used different

plotting functions, plotting for instance elliptical decision boundary in the plot of with the data points from the test and training data set.

The function `property_plot_elliptical_separable(data,subplot_Num,fig_title,w,fig_num)` plots the elliptical decision boundary.

The function `test_elliptical_logistic_regression(file_name_train,file_name_test)` and the new plotting function are in the file `logistic_regression.py`.

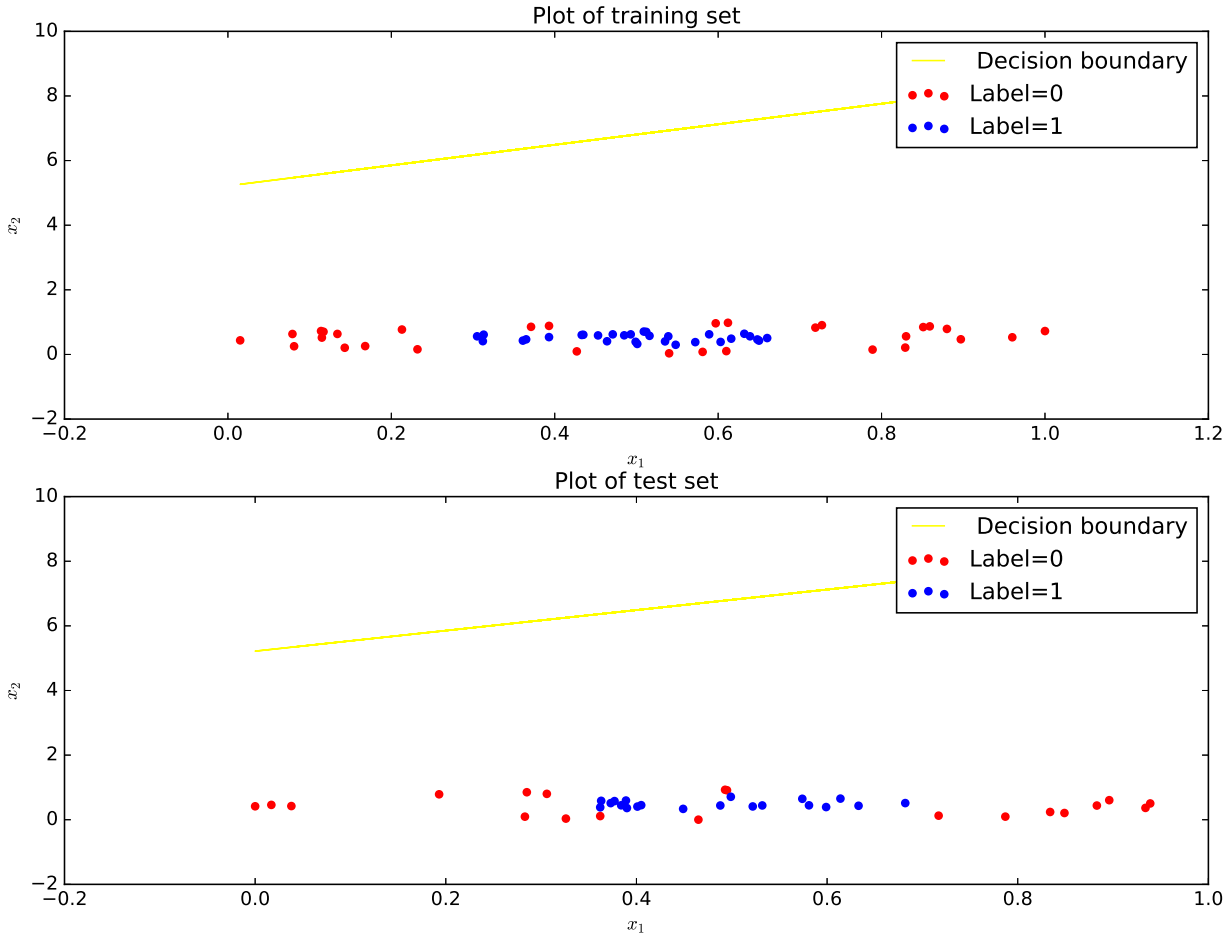


Figure 4: The figure shows the decision boundary placement for the test data set and training data set. The red data points are labeled with 0, and the blue data points being labeled 1.

After transforming $\mathbf{X}$ such that we get elliptical decision boundary, the new decision boundary classified all the test data correctly, as seen in figure 5.

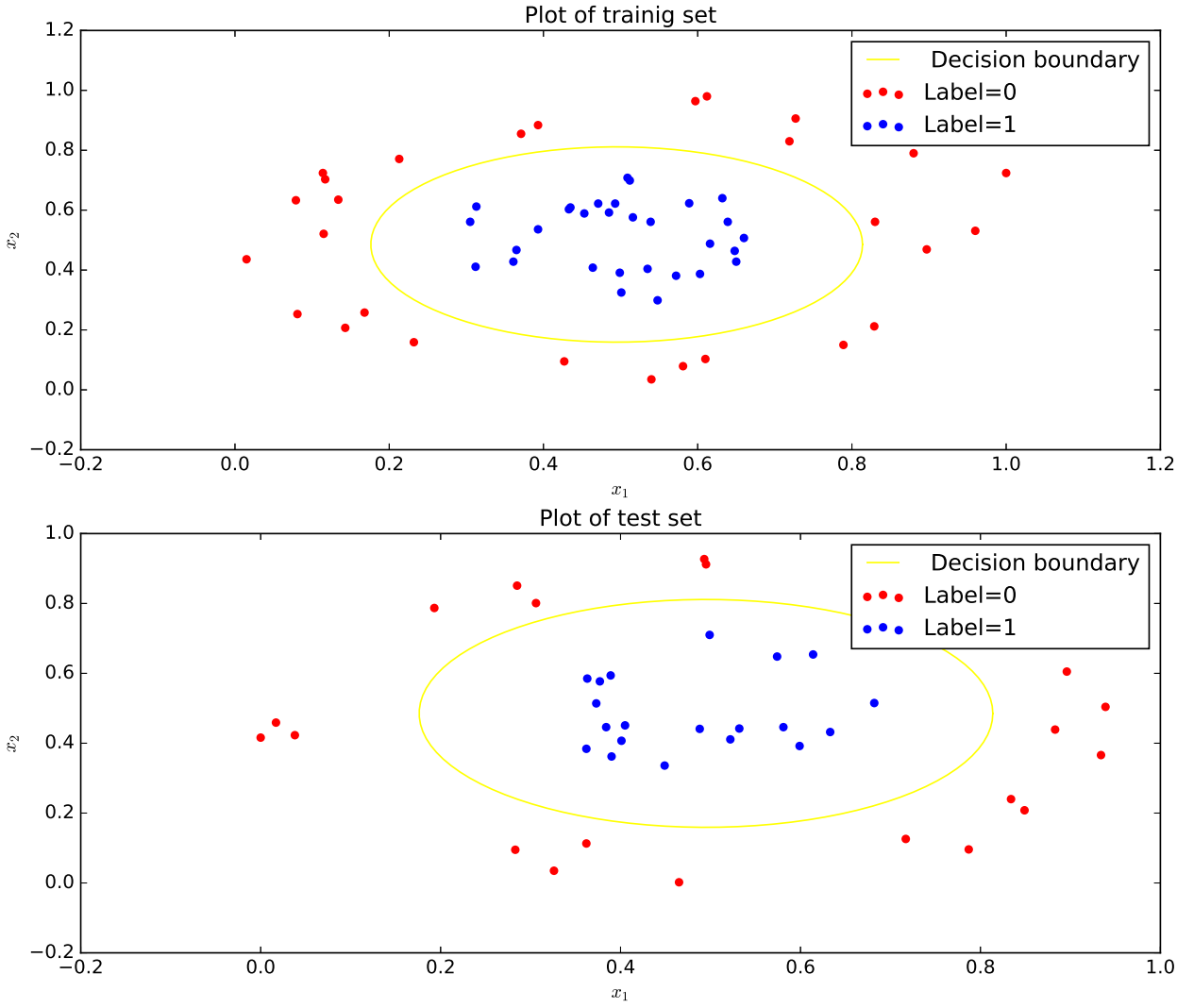


Figure 5: The figure shows the elliptical decision boundary placement of the test data set and traiingn data set. The red data points are labeled with 0, and the blue data points being labeled 1.

With learning rate of 0.1, and start weights $[0.1, 0.1, 0.1]$ the final weights are: $\mathbf{w} = [-9.74103026, 26.16888498, 24.52636526, -26.43587329, -25.26835216]^\top$ with final cross-entropy error:

- Train data set : $E_{ce} = 0.12695465467$
- Test data set : $E_{ce} = 0.116066279494$

This exercise can be executed be running the file `logistic_regression.py`, which starts the function

- `test_linear_regression(train_file_name,test_file_name)`
- `test_elliptical_regression(train_file_name,test_file_name)`

References

[Mit97] T.M Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997. ISBN: 0070428077.