NTNU

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Assignment 1

TDT4173: Machine Learning and Case-Based Reasoning

Submitted by
Vittorio Triassi

September 2019

1 Theory

1.1 What is *concept learning*, explain with an example?

When we refer to *concept learning*, we are talking about the ability to learn categories of objects so it should be possible to recognize new instances of those categories in the future. If we want to be more formal, we might say that we aim at inferring a boolean-valued function from the training examples. In order to provide an example of concept learning, we should define a task that learns a target concept. In our case, the *target concept* could be "matches on which a football team is likely to win". Then we will have a list of examples with several attributes. Each example is going to represent a previous match for which we already know the outcome. What we are going to do is to search through the space of the hypotheses, looking for the one that best fits the training examples. The hypotheses can be represented from the most general to the most specific one. As far as our problem, the goal is to learn the sets of the matches for which the attribute e.g. MatchWon = yes. We achieve that thanks to a conjunction of constraints on the attributes.

1.2 What is function approximation and why do we need them?

In machine learning problems, we need *function approximation* when we are performing tasks like classification, prediction, data mining etc. The reason lies in the mechanism by which we "learn". In fact, we are looking for an approximation of the target function rather than its ideal version. More precisely we would like to obtain a function that minimizes the error between the original function (e.g. V) that is the one for which we know the labels to be true and its approximation/prediction $\hat{V}$. Depending on the problem we are trying to solve, some function approximators could be: linear combinations of features, Neural Networks, Decision trees etc.

1.3 What is *inductive bias* in the context of machine learning, and why is it so important? Decision tree learning and the candidate elimination algorithm are two different learning algorithms. What can you say about the inductive bias for each of them?

In order for every model to learn a target function and to generalize beyond the training examples, it is necessary to introduce an *inductive bias.* There are different types of inductive bias but the main idea is to make some assumptions and then use them when predicting on unseen inputs. In other words, we have a set of assumptions made by the learner that allows the target function to generalize on new data. We have covered two learning algorithms so far: *candidate elimination* and *decision tree.* With candidate elimination, the target concept c is included in the hypothesis space H. It is important that the algorithm includes the inductive bias otherwise if it was unbiased, it would not be possible to classify unseen examples. On the other side, there are decision trees, in which a few rules are generally preferred. Since the way we make a decision is *top-down*, it is better to have a tree with good splits close to the root than a tree with bad splits on the top. Even if they still represent the same function. When talking about "good splits" we refer to those splits that have higher *information gain.* Moreover, a shorter tree is always preferred to longer trees. In this way we get to the correct answer faster.

1.4 What is overfitting, and how does it differ from underfitting? Briefly explain what a validation set is. How can cross-validation be used to mitigate overfitting?

Overfitting and underfitting are two of several drawbacks we might encounter while training our models. We run into *overfitting* when our model fits the data too well. This means that it is not generalizing anymore and it memorized our training examples. For instance, we might have 99% of accuracy on our dataset and around 50% on unseen data. Similarly, with *underfitting*, we are not fitting the data well enough. We say that with overfitting we have *low bias* and *high variance*, while in underfitting just the other way around: *high bias* and *low variance.* When building a model, we usually have three sets: the training set, the validation set and the testing set. Depending on how much data we have, a good compromise is to split the dataset in the following way: 60% for the training step, 20% for the validation and the remaining 20% for the testing. The validation set is used to prevent underfitting and its goal is to tune the hyperparameters in order to find the best combination for the model. A few examples of hyperparameters might be the learning rate or the depth of a tree. *Cross-validation* performs the same action of the validation set but it repeates that step several times using subsets of the training set as validation sets. Overall, cross-validation does not really avoid overfitting but it really helps us to understand whether we are overfitting the data or not.

1.5 Candidate Elimination algorithm

In the *Candidate Elimination* algorithm, we aim at returning a subset of hypotheses taken from the hypothesis space H that is consistent with the training data. This is also the formal definition of the *version space*. To achieve that, we use the approach *more_general_than*. In the version space we start with the most general and the most specific boundary possible. In our task we have to build a version space for the "*Treatment successful*" concept learning problem. Every time we run into a positive example, we will increase the specific set. Every time we run into a negative one, we will increase the general set.

Our starting point is going to be the following:

$$S0 \leftarrow \{< \emptyset, \emptyset, \emptyset, \emptyset >\}$$
$$G0 \leftarrow \{<?, ?, ?, ? >\}$$

Let us take into account the first training example, which is known to be a positive one.

$$1. < Female, Back, Medium, Medium >, TreatmentSuccessful = Yes$$

Then we have to increase the boundary for the specific set and at the same time we check the generic set to be consistent, updating its version:

$$S0 : \{< \emptyset, \emptyset, \emptyset, \emptyset >\}$$
$$\downarrow$$
$$S1 : \{< Female, Back, Medium, Medium >\}$$
$$G0, 1 : \{<?, ?, ?, ? >\}$$

For the second training example:

$$2. < Female, Neck, Medium, High >, TreatmentSuccessful = Yes$$

which is still positive, we continue to increase the specific set, making sure that it is more general than the previous one.

$$S0 : \{< \emptyset, \emptyset, \emptyset, \emptyset >\}$$
$$\downarrow$$
$$S1 : \{< Female, Back, Medium, Medium >\}$$
$$\downarrow$$
$$S2 : \{< Female, ?, Medium, ? >\}$$
$$G0, 1, 2 : \{<?, ?, ?, ? >\}$$

Also in this case, we update the version of the general set, making sure it is still consistent (and it clearly is since it is the most general we can have).

Let us consider now the third training example, which is the following:

$$3. < Female, Shoulder, Low, Low >, TreatmentSuccessful = No$$

In this case, we are forced to specialize $G2$ in $G3$ because we are dealing with a negative example. After we create $G3$, we also make sure that the previous $S2$ is still consistent. Since it is, we update its version in $S3$ that remains the same. We end up with the following:

$$\begin{tikzcd}
S0 : \{< \emptyset, \emptyset, \emptyset, \emptyset >\} \arrow[d] \\
S1 : \{< Female, Back, Medium, Medium >\} \arrow[d] \\
S2,3 : \{< Female, ?, Medium, ? >\} \\
G3 : \{<?, ?, Medium, ? >\} \\
G0,1,2 : \{<?, ?, ?, ? >\} \arrow[u]
\end{tikzcd}$$

As far as the fourth example, we have again, a positive example. So this means we are going to further generalize the specific boundary.

$$4. < Male, Neck, High, Medium >, TreatmentSuccessful = Yes$$

In this case, it turns out that if we carry on generalizing, we will not be able to learn any concept, because our attributes are too general. In fact, the final set will be empty as shown.

$$\begin{tikzcd}
S0 : \{< \emptyset, \emptyset, \emptyset, \emptyset >\} \arrow[d] \\
S1 : \{< Female, Back, Medium, Medium >\} \arrow[d] \\
S2,3 : \{< Female, ?, Medium, ? >\} \arrow[d] \\
S4 : \{\} = \emptyset \\
G4 : \{\} = \emptyset \\
G3 : \{<?, ?, Medium, ? >\} \arrow[u] \\
G0,1,2 : \{<?, ?, ?, ? >\} \arrow[u]
\end{tikzcd}$$

It is going to be the same also with our fifth and last training example:

$$5. < Male, Back, Medium, Low >, TreatmentSuccessful = Yes$$

Since we have empty sets, we will not be able to further abstract our version space in any way. So our final version space will be the following:

$$S0 : \{< \emptyset, \emptyset, \emptyset, \emptyset >\}$$

$$\downarrow$$

$$S1 : \{< Female, Back, Medium, Medium >\}$$

$$\downarrow$$

$$S2,3 : \{< Female, ?, Medium, ? >\}$$

$$\downarrow$$

$$S4,5 : \{\} = \emptyset$$

$$G4,5 : \{\} = \emptyset$$

$$\uparrow$$

$$G3 : \{<?, ?, Medium, ? >\}$$

$$\uparrow$$

$$G0,1,2 : \{<?, ?, ?, ? >\}$$

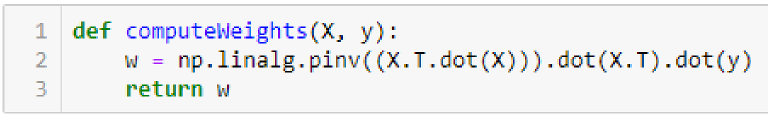
2 Programming

Task 1 - Linear Regression

1. Implement linear regression with ordinary least squares (OLS) using the closed-form solution seen in Equation 9. Tip: This can be done in a single line of code!

To solve the following task, we will write a function called **computeWeights** in which, provided the matrix **X** and its labels vector **y**, we are able to compute the set of the weights we need for the Linear Regression.

$$w = (X^T X)^{-1} X^T y$$



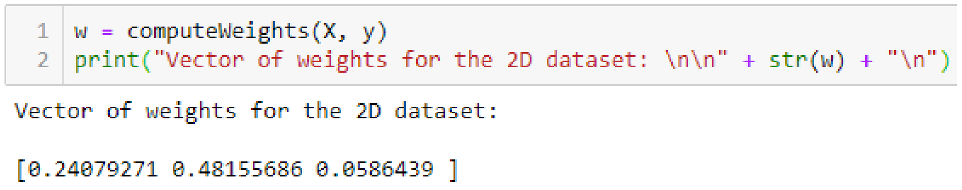
```python
def computeWeights(X, y):
    w = np.linalg.pinv((X.T.dot(X))).dot(X.T).dot(y)
    return w
```

Figure 2.1: computeWeights function

2. Load the data in train_2d_reg_data.csv (training data) and test_2d_reg_data.csv (test data) and use your OLS implementation to find a set of good weights for the training data. Show the weights as well as the model error $E_{mse}(w)$ for the training and test set after training. Is your model generalizing well?

In order to carry out the following task, we need to store our data into two structures: the input matrix **X**, and the **y** vector which will contain the real-valued output. The X matrix has to be modified by adding a column of ones. This will let us to include the bias in the computation. Moreover, since our csv files have an extra row (the first one) that defines the columns, we will get rid of it since we need to fill our matrices just with proper examples.

After that, we can compute the weights for our model:



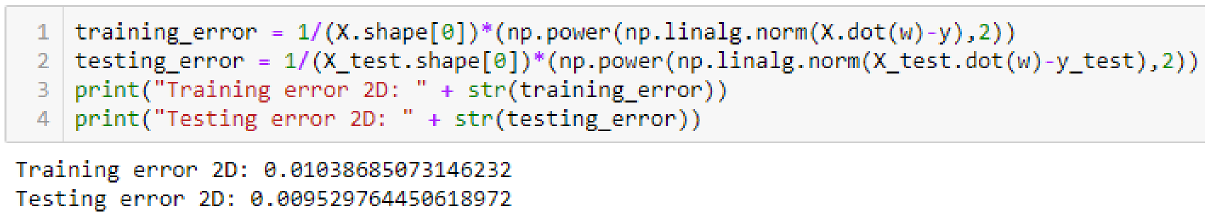
```
w = computeWeights(X, y)
print("Vector of weights for the 2D dataset: \n\n" + str(w) + "\n")
```

```
Vector of weights for the 2D dataset:

[0.24079271 0.48155686 0.0586439 ]
```

Figure 2.2: Weights computed

To find the *training* and *test* errors, we need to compute the mean square error by using the following equation:

$$E_{mse}(w) = \frac{1}{N}||Xw - y||^2$$



```
training_error = 1/(X.shape[0])*(np.power(np.linalg.norm(X.dot(w)-y),2))
testing_error = 1/(X_test.shape[0])*(np.power(np.linalg.norm(X_test.dot(w)-y_test),2))
print("Training error 2D:  " + str(training_error))
print("Testing error 2D:  " + str(testing_error))
```

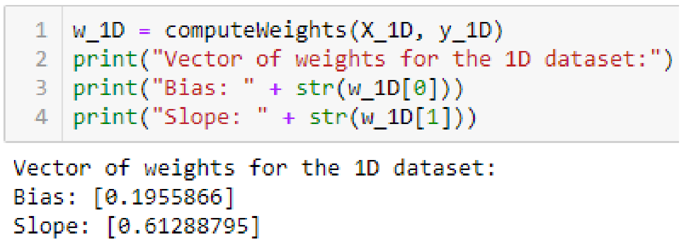
```
Training error 2D: 0.01038685073146232
Testing error 2D: 0.009529764450618972
```

Figure 2.3: Training and testing errors

As we can see from the results shown above, we are doing quite well with our OLS implementation as the mean square error is converging toward zero. It is worth noting though that we end up with a testing error lower than the training one. This is a bit uncommon but most likely it depends on the shape of our data.

3. Load the data in *train_1d_reg_data.csv* and *test_1d_reg_data.csv* and use your OLS implementation to find a set of good weights for the training data. Using these weights, make a plot of the line fitting the data and show this in the report. Does the line you found fit the data well? If not, discuss in broad terms how this can be remedied. Tip: Remember, for this dataset there are only two weights: the first is the bias, while the second is the slope

Also in this case, it is necessary to manipulate the given dataset as well as we did earlier. After doing so, we calculate the set of the weights for the 1D dataset. It is worth noting that we have just two of them this time.

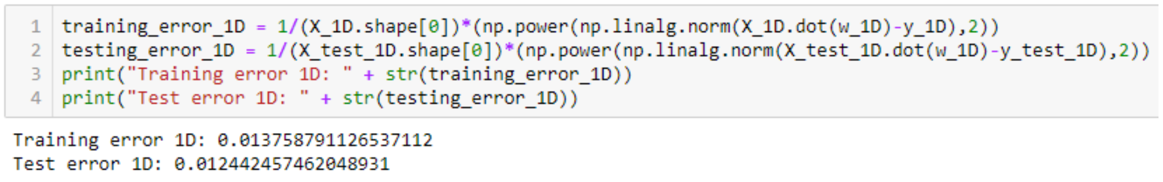


```
w_1D = computeWeights(X_1D, y_1D)
print("Vector of weights for the 1D dataset:")
print("Bias: " + str(w_1D[0]))
print("Slope: " + str(w_1D[1]))
```

```
Vector of weights for the 1D dataset:
Bias: [0.1955866]
Slope: [0.61288795]
```

Figure 2.4: Bias and slope

It is always good to calculate the training and the test error even if they will not be used for the next step.



```
training_error_1D = 1/(X_1D.shape[0])*(np.power(np.linalg.norm(X_1D.dot(w_1D)-y_1D),2))
testing_error_1D = 1/(X_test_1D.shape[0])*(np.power(np.linalg.norm(X_test_1D.dot(w_1D)-y_test_1D),2))
print("Training error 1D: " + str(training_error_1D))
print("Test error 1D: " + str(testing_error_1D))
```

```
Training error 1D: 0.013758791126537112
Test error 1D: 0.012442457462048931
```

Figure 2.5: Training and testing errors

Now we plot the line we obtain with the weights found before. The following line should fit our data. For the purpose of this task, we have stored the coordinates of two points in two vectors and then plotted the result on both, the training and test data.

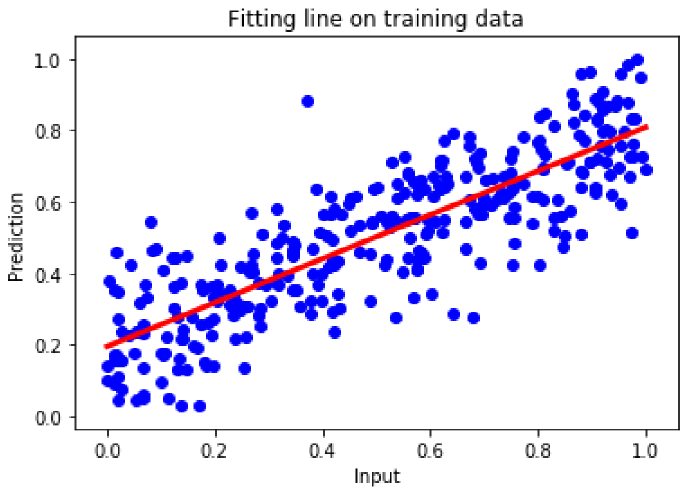


Figure 2.6: Fitting line on training data

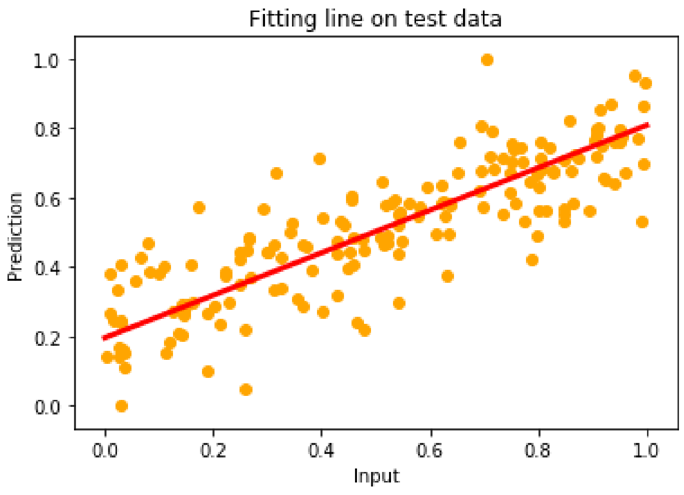


Figure 2.7: Fitting line on test data

As we can see from the results, the line perfectly fits our data (both, training and testing examples), so it means that for the purpose of this task, we computed good weights.

Task 2 - Logistic Regression

1. Load the data in *cl_train_1.csv* and *cl_test_1.csv* and use your logistic regression implementation to train on the data in the training set. Is the data linearly separable? Explain your reasoning. Additionally, show a plot of the cross-entropy error for both the training and test set over 1000 iterations of training. What learning rate η and initial parameters w did you select? Is your model generalising well? Keep in mind that you may have to test several different learning rates to get a good classification. It may be helpful to plot the decision boundary to get a visual representation of the fit. This will allow you to see how well you are doing.

After a bit of manipulation on the csv files, it is a good idea to plot our training examples in order to get a deeper insight about the shape of the data and its distribution in the space. It is quite evident that the two classes (positive and negative) are *linearly separable* because we might fit a line exactly in the middle between them without using a polynomial of higher degree.

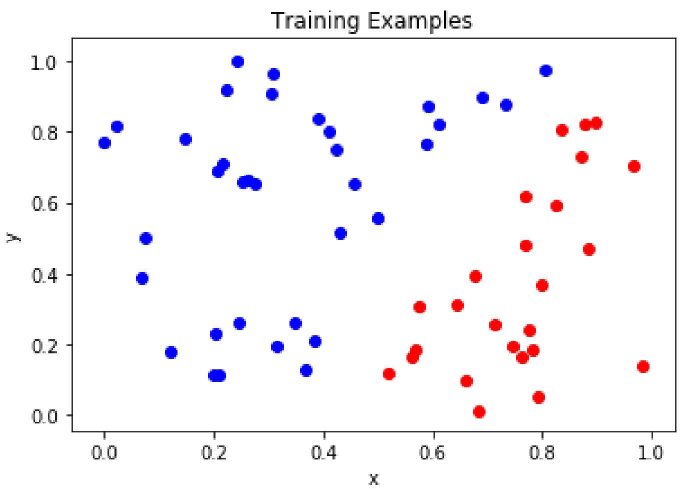
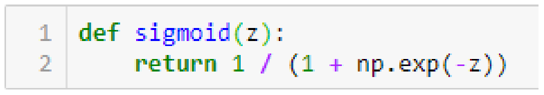


Figure 2.8: Training examples

To carry out the Logistic Regression task, we need to define the *sigmoid* function and the *crossEntropy* function. The sigmoid function or also called logistic function, is defined as follows:

$$\sigma(z) = \frac{1}{1+e^{-z}} \text{ where } z = h(x) = w^T x$$

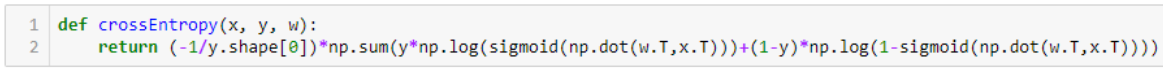


```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Figure 2.9: Sigmoid

While the cross entropy error is computed as:

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



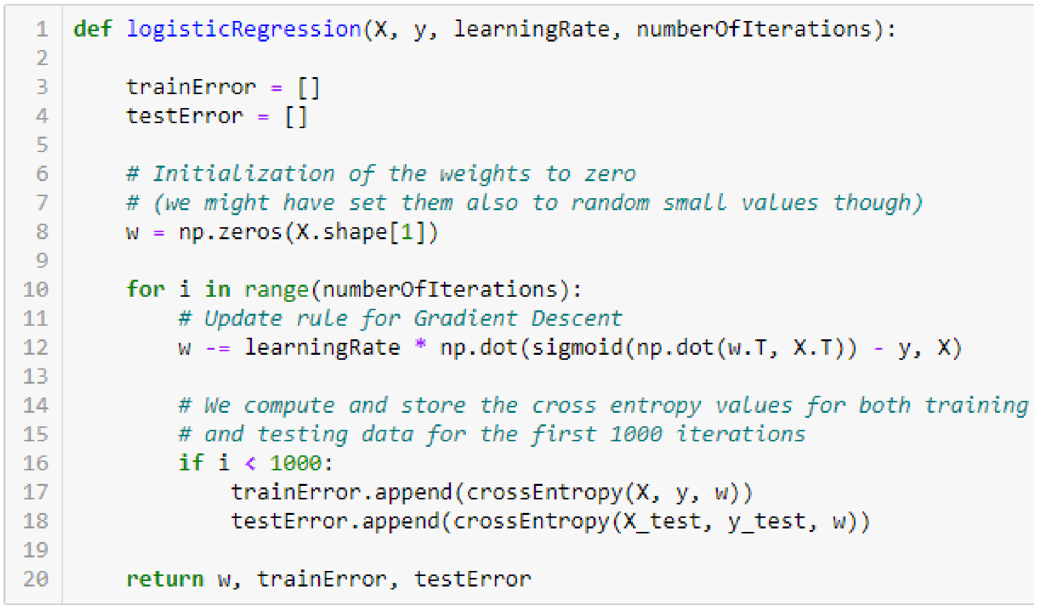
```python
def crossEntropy(x, y, w):
    return (-1/y.shape[0])*np.sum(y*np.log(sigmoid(np.dot(w.T,x.T)))+(1-y)*np.log(1-sigmoid(np.dot(w.T,x.T))))
```

Figure 2.10: Cross Entropy

After that, we can take care of the core function of our task, that is going to perform the update rule for the gradient descent and will let us to come up with a set of weights. The update rule is the following:

$$w(k+1) \leftarrow w(k) - \eta\sum_{i=1}^{N}(\sigma(z) - y_i)x_i$$

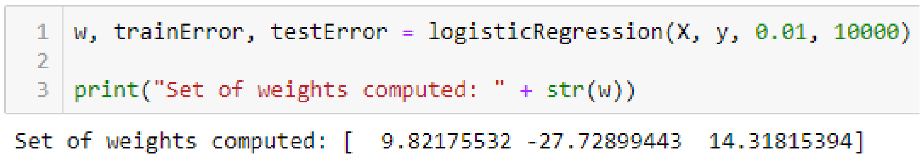
$$\leftarrow w(k) - \eta\sum_{i=1}^{N}(\sigma(w(k)^T x_i) - y_i)x_i$$



```python
def logisticRegression(X, y, learningRate, numberOfIterations):

    trainError = []
    testError = []

    # Initialization of the weights to zero
    # (we might have set them also to random small values though)
    w = np.zeros(X.shape[1])

    for i in range(numberOfIterations):
        # Update rule for Gradient Descent
        w -= learningRate * np.dot(sigmoid(np.dot(w.T, X.T)) - y, X)

        # We compute and store the cross entropy values for both training
        # and testing data for the first 1000 iterations
        if i < 1000:
            trainError.append(crossEntropy(X, y, w))
            testError.append(crossEntropy(X_test, y_test, w))

    return w, trainError, testError
```

Figure 2.11: Logistic Regression

Now, providing in input the learning rate and the number of desired iterations, we can perform the training step. For the purpose of this exercise, we have tried several combinations for these two parameters. Depending on how big is the learning rate, we are going to compute the gradient more or less aggressively. Here we show the results when using a learning rate $\eta = 0.01$ for 10000 iterations. Moreover, in the implementation of our Logistic Regression algorithm, we decided to set our initial weights to all zeros. Of course we might have set them to small random values as well. By running the Logistic Regression we obtain the weights shown below and the two cross entropy errors that will help us to understand how good we are doing with our model.



```
w, trainError, testError = logisticRegression(X, y, 0.01, 10000)

print("Set of weights computed: " + str(w))
```

```
Set of weights computed: [  9.82175532 -27.72899443  14.31815394]
```

Figure 2.12: Weights computed

On the following graph we plot the two cross entropy errors for the first 1000 iterations in order to understand how our model behaves when dealing with unseen data. As we can see, the model is doing quite well on new inputs and it is not overfitting. The trend is almost the same when using a bigger learning rate $\eta = 0.1$ which might be an option if we want to speed up the training computation.

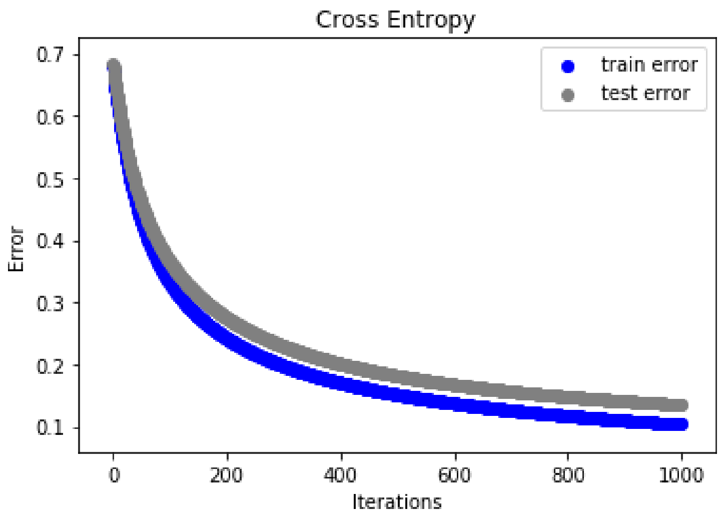


Figure 2.13: Cross Entropy errors

To better show that our classes are linearly separable, we decide to compute the *decision boundary*. The result is the line shown below. To calculate the boundary we do the following.

Suppose that $x = (x_1, x_2)$ and the parameters are defined by $w = (w_0, w_1, w_2)$, we have:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

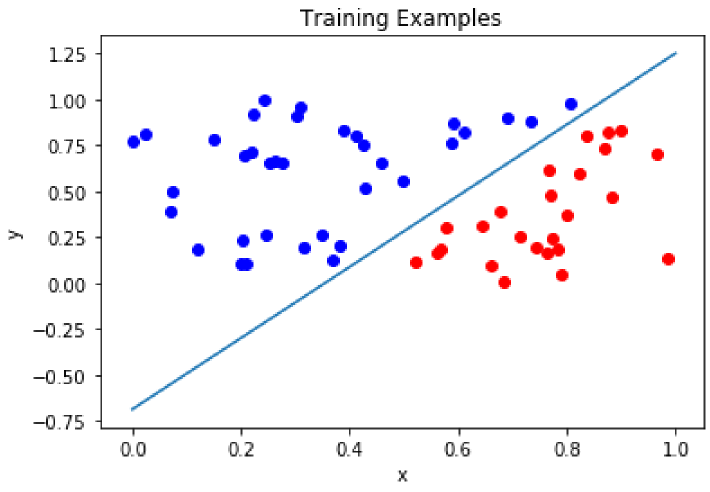
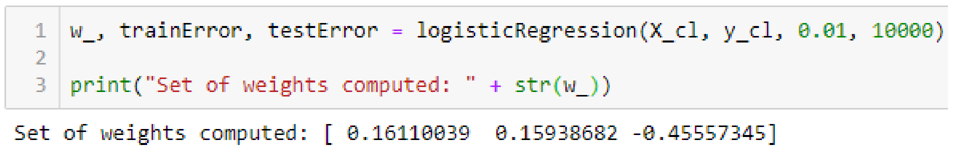


Figure 2.14: Decision boundary

2. Load the data in *cl_train_2.csv* and *cl_test_2.csv* and use your logistic regression implementation to train on the data in the training set. Is the data linearly separable? Explain your reasoning. Plot the *decision boundary* as explained in the previous task as well as the data points in the training and test set. Discuss what can be done for logistic regression to correctly classify the dataset. Make the necessary changes to your implementation and show the new decision boundary in your report.

As usual we load the data and manipulate a bit the csv files. Then, we run the Logistic Regression function previously defined, choosing the learning rate and the number of iterations desired. The weights computed by the algorithm are shown below.



```
w_, trainError, testError = logisticRegression(X_cl, y_cl, 0.01, 10000)

print("Set of weights computed: " + str(w_))
```

```
Set of weights computed: [ 0.16110039  0.15938682 -0.45557345]
```

Figure 2.15: Weights computed

By plotting the training and testing examples and computing the decision boundary, it turns out that the two classes are not linearly separable. For this reason we should look for a solution that allows us to separate the two classes with more than just a line. One option might be to project our data in a higher dimensional space. Basically, we move from our two-dimensional space to at least a three-dimensional space. Depending on the complexity of our data we could increase the dimensionality way more.

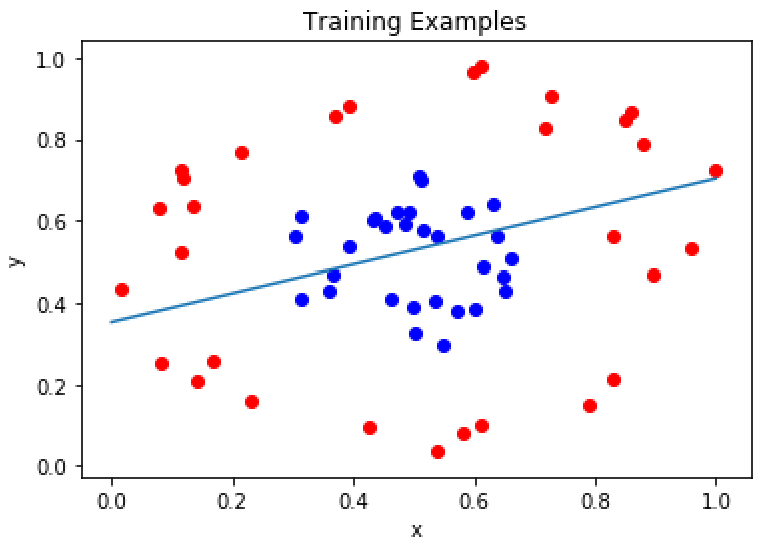


Figure 2.16: Train data

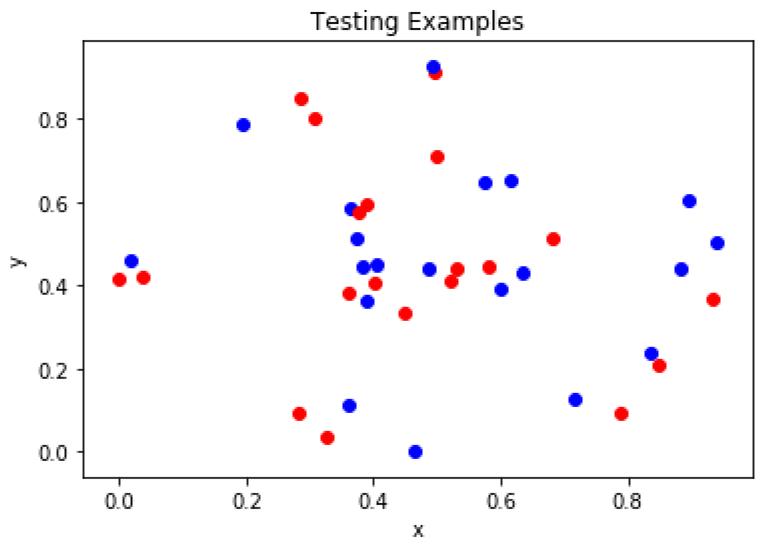
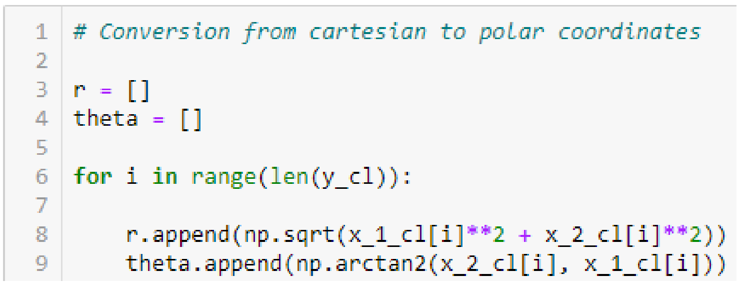


Figure 2.17: Test data

Our idea is to convert the coordinates of our data points from cartesian to polar ones. In order to do that we perform the following conversion:

$$r = \sqrt{x^2 + y^2} \text{ and } \Theta = \arctan|\tfrac{y}{x}|$$



```python
# Conversion from cartesian to polar coordinates

r = []
theta = []

for i in range(len(y_cl)):

    r.append(np.sqrt(x_1_cl[i]**2 + x_2_cl[i]**2))
    theta.append(np.arctan2(x_2_cl[i], x_1_cl[i]))
```

Figure 2.18: Conversion from cartesian to polar coordinates

This different representation lets us to linearly separate the data points. Another option might have been to introduce a kernel function, but of course it would have been more difficult to find an expression to properly plot the boundary.