Data Analysis and Machine Learning: Logistic Regression

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Jan 28, 2019

Logistic Regression

In linear regression our main interest was centered on learning the coefficients of a functional fit (say a polynomial) in order to be able to predict the response of a continuous variable on some unseen data. The fit to the continuous variable y_i is based on some independent variables $\hat{x}_i$. Linear regression resulted in analytical expressions (in terms of matrices to invert) for several quantities, ranging from the variance and thereby the confidence intervals of the parameters $\hat{\beta}$ to the mean squared error. If we can invert the product of the design matrices, linear regression gives then a simple recipe for fitting our data.

Classification problems, however, are concerned with outcomes taking the form of discrete variables (i.e. categories). We may for example, on the basis of DNA sequencing for a number of patients, like to find out which mutations are important for a certain disease; or based on scans of various patients' brains, figure out if there is a tumor or not; or given a specific physical system, we'd like to identify its state, say whether it is an ordered or disordered system (typical situation in solid state physics); or classify the status of a patient, whether she/he has a stroke or not and many other similar situations.

The most common situation we encounter when we apply logistic regression is that of two possible outcomes, normally denoted as a binary outcome, true or false, positive or negative, success or failure etc.

Optimization and Deep learning

Logistic regression will also serve as our stepping stone towards neural network algorithms and supervised deep learning. For logistic learning, the minimization of the cost function leads to a non-linear equation in the parameters $\hat{\beta}$. The optmization of the problem calls therefore for minimization algorithms. This forms the bottle neck of all machine learning algorithms, namely how to find

reliable minima of a multi-variable function. This leads us to the family of gradient descent methods. The latter are the working horses of basically all modern machine learning algorithms.

We note also that many of the topics discussed here regression are also commonly used in modern supervised Deep Learning models, as we will see later.

Basics

We consider the case where the dependent variables, also called the responses or the outcomes, y_i are discrete and only take values from $k = 0, \ldots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $\hat{X} \in \mathbb{R}^{n \times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and $y_i = 1$. Our outcomes could represent the status of a credit card user who could default or not on her/his credit card debt. That is

$$y_i = \begin{bmatrix} 0 & \text{no} \\ 1 & \text{yes} \end{bmatrix}.$$

Linear classifier

Before moving to the logistic model, let us try to use our linear regression model to classify these two outcomes. We could for example fit a linear model to the default case if $y_i > 0.5$ and the no default case $y_i \leq 0.5$.

We would then have our weighted linear combination, namely

$$\hat{y} = \hat{X}^T \hat{\beta} + \hat{\epsilon}, \tag{1}$$

where $\hat{y}$ is a vector representing the possible outcomes, $\hat{X}$ is our $n \times p$ design matrix and $\hat{\beta}$ represents our estimators/predictors.

Some selected properties

The main problem with our function is that it takes values on the entire real axis. In the case of logistic regression, however, the labels y_i are discrete variables.

One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to values $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 if otherwise. We will encounter this model in our first demonstration of neural networks. Historically it is called the “perceptron" model in the machine learning literature. This model is extremely simple. However, in many cases it is more favorable to use a “soft" classifier that outputs the probability of a given category. This leads us to the logistic function.

The code for plotting the perceptron can be seen here. This si nothing but the standard Heaviside step function.

The logistic function

The perceptron is an example of a "hard classification" model. We will encounter this model when we discuss neural networks as well. Each datapoint is deterministically assigned to a category (i.e $y_i = 0$ or $y_i = 1$). In many cases, it is favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category k. Logistic regression is the most common example of a so-called soft classifier. In logistic regression, the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so-called logit function (or Sigmoid) which is meant to represent the likelihood for a given event,

$$p(t) = \frac{1}{1 + \exp -t} = \frac{\exp t}{1 + \exp t}.$$

Note that $1 - p(t) = p(-t)$. The following code plots the logistic function.

Two parameters

We assume now that we have two classes with y_i either 0 or 1. Furthermore we assume also that we have only two parameters β in our fitting of the Sigmoid function, that is we define probabilities

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

where $\hat{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

Note that we used

$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}).$$

Maximum likelihood

In order to define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0, 1\}$ and where the data points are drawn independently, we use the so-called Maximum Likelihood Estimation (MLE) principle. We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is

$$P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \hat{\beta})\right]^{y_i} \left[1 - p(y_i = 1|x_i, \hat{\beta}))\right]^{1-y_i}$$

from which we obtain the log-likelihood and our **cost/loss** function

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i \log p(y_i = 1|x_i, \hat{\beta}) + (1 - y_i) \log \left[1 - p(y_i = 1|x_i, \hat{\beta}) \right] \right).$$

The cost function rewritten

Reordering the logarithms, we can rewrite the **cost/loss** function as

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right) \right).$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. Since the cost (error) function is just the negative log-likelihood, for logistic regression we have that

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right) \right).$$

This equation is known in statistics as the **cross entropy**. Finally, we note that just as in linear regression, in practice we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization as we did for Ridge and Lasso regression.

Minimizing the cross entropy

The cross entropy is a convex function of the weights $\hat{\beta}$ and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to the two parameters β_0 and β_1 we obtain

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)} \right),$$

and

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)} \right).$$

A more compact expression

Let us now define a vector $\hat{y}$ with n elements y_i, an $n \times p$ matrix $\hat{X}$ which contains the x_i values and a vector $\hat{p}$ of fitted probabilities $p(y_i|x_i, \hat{\beta})$. We can rewrite in a more compact form the first derivative of cost function as

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T \left(\hat{y} - \hat{p} \right).$$

If we in addition define a diagonal matrix $\hat{W}$ with elements $p(y_i|x_i,\hat{\beta})(1-p(y_i|x_i,\hat{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2\mathcal{C}(\hat{\beta})}{\partial\hat{\beta}\partial\hat{\beta}^T}=\hat{X}^T\hat{W}\hat{X}.$$

Extending to more predictors

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log\frac{p(\hat{\beta}\hat{x})}{1-p(\hat{\beta}\hat{x})}=\beta_0+\beta_1x_1+\beta_2x_2+\cdots+\beta_px_p.$$

Here we defined $\hat{x}=[1,x_1,x_2,\dots,x_p]$ and $\hat{\beta}=[\beta_0,\beta_1,\dots,\beta_p]$ leading to

$$p(\hat{\beta}\hat{x})=\frac{\exp\left(\beta_0+\beta_1x_1+\beta_2x_2+\cdots+\beta_px_p\right)}{1+\exp\left(\beta_0+\beta_1x_1+\beta_2x_2+\cdots+\beta_px_p\right)}.$$

Including more classes

Till now we have mainly focused on two classes, the so-called binary system. Suppose we wish to extend to K classes. Let us for the sake of simplicity assume we have only two predictors. We have then following model

$$\log\frac{p(C=1|x)}{p(K|x)}=\beta_{10}+\beta_{11}x_1,$$

$$\log\frac{p(C=2|x)}{p(K|x)}=\beta_{20}+\beta_{21}x_1,$$

and so on till the class $C=K-1$ class

$$\log\frac{p(C=K-1|x)}{p(K|x)}=\beta_{(K-1)0}+\beta_{(K-1)1}x_1,$$

and the model is specified in term of $K-1$ so-called log-odds or **logit** transformations.

The Softmax function

In our discussion of neural networks we will encounter the above again in terms of the so-called **Softmax** function.

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression (also known as softmax regression), multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks. Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the

predicted probability for the k-th class given a sample vector $\hat{x}$ and a weighting vector $\hat{\beta}$ is (with two predictors):

$$p(C = k|\mathbf{x}) = \frac{\exp(\beta_{k0} + \beta_{k1}x_1)}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_{l1}x_1)}.$$

It is easy to extend to more predictors. The final class is

$$p(C = K|\mathbf{x}) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_{l1}x_1)},$$

and they sum to one. Our earlier discussions were all specialized to the case with two classes only. It is easy to see from the above that what we derived earlier is compatible with these equations.

To find the optimal parameters we would typically use a gradient descent method. Newton's method and gradient descent methods are discussed in the material on optimization methods.

A scikit-learn example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0

from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)

X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
plt.show()
```

A simple classification problem

```
import numpy as np
from sklearn import datasets, linear_model
import matplotlib.pyplot as plt


def generate_data():
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    return X, y


def visualize(X, y, clf):
    # plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
    # plt.show()
```

```
    plot_decision_boundary(lambda x: clf.predict(x), X, y)
    plt.title("Logistic Regression")


def plot_decision_boundary(pred_func, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
    plt.show()


def classify(X, y):
    clf = linear_model.LogisticRegressionCV()
    clf.fit(X, y)
    return clf


def main():
    X, y = generate_data()
    # visualize(X, y)
    clf = classify(X, y)
    visualize(X, y, clf)


if __name__ == "__main__":
    main()
```

The two-dimensional Ising model, Predicting phase transition of the two-dimensional Ising model

The Hamiltonian of the two-dimensional Ising model without an external field for a constant coupling constant J is given by

$$H = -J \sum_{\langle ij \rangle} S_i S_j, \tag{2}$$

where $S_i \in \{-1, 1\}$ and $\langle ij \rangle$ signifies that we only iterate over the nearest neighbors in the lattice. We will be looking at a system of $L = 40$ spins in each dimension, i.e., $L^2 = 1600$ spins in total. Opposed to the one-dimensional Ising model we will get a phase transition from an **ordered** phase to a **disordered** phase at the critical temperature

$$\frac{T_c}{J} = \frac{2}{\log\left(1 + \sqrt{2}\right)} \approx 2.26, \tag{3}$$

as shown by Lars Onsager.

Here we use **logistic regression** to predict when a phase transition occurs. The data we will look at is a set of spin configurations, i.e., individual lattices with spins, labeled **ordered** `1` or **disordered** `0`. Our job is to build a model which will take in a spin configuration and predict whether or not the spin configuration constitutes an ordered or a disordered phase. To achieve this we will represent the lattices as flattened arrays with 1600 elements instead of a matrix of 40×40 elements. As an extra test of the performance of the algorithms we will divide the dataset into three pieces. We will do a conventional train-test-split on a combination of totally ordered and totally disordered phases. The remaining "critical-like" states will be used as test data which we hope the model will be able to make good extrapolated predictions on.

```
import pickle
import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn.model_selection as skms
import sklearn.linear_model as skl
import sklearn.metrics as skm
import tqdm
import copy
import time
from IPython.display import display

%matplotlib inline

sns.set(color_codes=True)
```

Reading in the data

Using the data from Mehta et al. (specifically the two datasets named `Ising2DFM_reSample_L40_T=All.pkl` and `Ising2DFM_reSample_L40_T=All_labels.pkl`) we have to unpack the data into numpy arrays.

```
filenames = glob.glob(os.path.join("..", "dat", "*"))
label_filename = list(filter(lambda x: "label" in x, filenames))[0]
dat_filename = list(filter(lambda x: "label" not in x, filenames))[0]

# Read in the labels
with open(label_filename, "rb") as f:
    labels = pickle.load(f)

# Read in the corresponding configurations
with open(dat_filename, "rb") as f:
    data = np.unpackbits(pickle.load(f)).reshape(-1, 1600).astype("int")

# Set spin-down to -1
data[data == 0] = -1
```

This dataset consists of 10000 samples, i.e., 10000 spin configurations with 40×40 spins each, for 16 temperatures between 0.25 to 4.0. Next we create a

train/test-split and keep the data in the critical phase as a separate dataset for extrapolation-testing.

```
# Set up slices of the dataset
ordered = slice(0, 70000)
critical = slice(70000, 100000)
disordered = slice(100000, 160000)

X_train, X_test, y_train, y_test = skms.train_test_split(
    np.concatenate((data[ordered], data[disordered])),
    np.concatenate((labels[ordered], labels[disordered])),
    test_size=0.95
)
```

Logistic regression

Logistic regression is a linear model for classification. Recalling the cost function for ordinary least squares with both L2 (ridge) and L1 (LASSO) penalties we will see that the logistic cost function is very similar. In OLS we wish to predict a continuous variable $\hat{y}$ using

$$\hat{y} = X\omega, \tag{4}$$

where $X \in \mathbb{R}^{n \times p}$ is the input data and $\omega^{p \times d}$ are the weights of the regression. In a classification setting (binary classification in our situation) we are interested in a positive or negative answer. We can thus define either answer to be above or below some threshold. But, in order to limit the size of the answer and also to get a probability interpretation on how sure we are for either answer we can compute the sigmoid function of OLS. That is,

$$f(X\omega) = \frac{1}{1 + \exp(-X\omega)}. \tag{5}$$

We are thus interested in minizming the following cost function

$$C(X, \omega) = \sum_{i=1}^{n} \left\{ -y_i \log\left(f(x_i^T \omega)\right) - (1 - y_i) \log\left[1 - f(x_i^T \omega)\right] \right\}, \tag{6}$$

where we will restrict ourselves to a value for $f(z)$ as the sigmoid described above. We can also tack on a L2 (Ridge) or L1 (LASSO) penalization to this cost function in the same manner we did for linear regression.

Exploring the logistic regression

The penalization factor λ is inverted in the case of the logistic regression model we use. We will explore several values of λ using both L1 and L2 penalization. We do this using a grid search over different parameters and run a 3-fold cross validation for each configuration. In other words, we fit a model 3 times for each configuration of the hyper parameters.

```
lambdas = np.logspace(-7, -1, 7)

param_grid = {
    "C": list(1.0/lambdas),
    "penalty": ["l1", "l2"]
}
clf = skms.GridSearchCV(
    skl.LogisticRegression(),
    param_grid=param_grid,
    n_jobs=-1,
    return_train_score=True
)
t0 = time.time()
clf.fit(X_train, y_train)
t1 = time.time()

print (
    "Time spent fitting GridSearchCV(LogisticRegression): {0:.3f} sec".format(
        t1 - t0
    )
)
```

We can see that logistic regression is quite slow and using the grid search and cross validation results in quite a heavy computation. Below we show the results of the different configurations.

```
logreg_df = pd.DataFrame(clf.cv_results_)

display(logreg_df)
```

Accuracy of a classification model

To determine how well a classification model is performing we count the number of correctly labeled classes and divide by the number of classes in total. The accuracy is thus given by

$$a(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} I(y_i = \hat{y}_i), \tag{7}$$

where $I(y_i = \hat{y}_i)$ is the indicator function given by

$$I(x = y) = \begin{array}{ll} 1 & x = y, \\ 0 & x \neq y. \end{array} \tag{8}$$

This is the accuracy provided by Scikit-learn when using **sklearn.metrics.accuracyscore**.

Below we compute the accuracy of the best fit model on the training data (which should give a good accuracy), the test data (which has not been shown to the model) and the critical data (completely new data that needs to be extrapolated).

```
train_accuracy = skm.accuracy_score(y_train, clf.predict(X_train))
test_accuracy = skm.accuracy_score(y_test, clf.predict(X_test))
```

```
critical_accuracy = skm.accuracy_score(labels[critical], clf.predict(data[critical]))

print ("Accuracy on train data: {0}".format(train_accuracy))
print ("Accuracy on test data: {0}".format(test_accuracy))
print ("Accuracy on critical data: {0}".format(critical_accuracy))
```

We can see that we get quite good accuracy on the training data, but gradually worsening accuracy on the test and critical data.

Analyzing the results

Below we show a different metric for determining the quality of our model, namely the **reciever operating characteristic** (ROC). The ROC curve tells us how well the model correctly classifies the different labels. We plot the **true positive rate** (the rate of predicted positive classes that are positive) versus the **false positive rate** (the rate of predicted positive classes that are negative). The ROC curve is built by computing the true positive rate and the false positive rate for varying **thresholds**, i.e, which probability we should acredit a certain class.

By computing the **area under the curve** (AUC) of the ROC curve we get an estimate of how well our model is performing. Pure guessing will get an AUC of 0.5. A perfect score will get an AUC of 1.0.

```
fig = plt.figure(figsize=(20, 14))

for (_X, _y), label in zip(
    [
        (X_train, y_train),
        (X_test, y_test),
        (data[critical], labels[critical])
    ],
    ["Train", "Test", "Critical"]
):
    proba = clf.predict_proba(_X)
    fpr, tpr, _ = skm.roc_curve(_y, proba[:, 1])
    roc_auc = skm.auc(fpr, tpr)

    print ("LogisticRegression AUC ({0}): {1}".format(label, roc_auc))

    plt.plot(fpr, tpr, label="{0} (AUC = {1})".format(label, roc_auc), linewidth=4.0)

plt.plot([0, 1], [0, 1], "--", label="Guessing (AUC = 0.5)", linewidth=4.0)

plt.title(r"The ROC curve for LogisticRegression", fontsize=18)
plt.xlabel(r"False positive rate", fontsize=18)
plt.ylabel(r"True positive rate", fontsize=18)
plt.axis([-0.01, 1.01, -0.01, 1.01])
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.legend(loc="best", fontsize=18)
plt.show()
```

We can see that this plot of the ROC looks very strange. This tells us that logistic regression is quite inept at predicting the Ising model transition and is therefore highly non-linear. The ROC curve for the training data looks quite

good, but as the testing data is so far off we see that we are dealing with an overfit model.

Credit Card data set

```
import pandas as pd
import os
import numpy as np

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score
```

The following runs the data preparation that is used for all models.

We scale all features by SciKit-learn's standard scaler. The standard scalars subtracts the mean, so that the means of the standardized variables equal zero. Furthermore the standard scaler divides the features by their respective variances, meaning that the variances of the standardized features equals one.

```
# Trying to set the seed
np.random.seed(0)
import random
random.seed(0)

# Reading file into data frame
cwd = os.getcwd()
filename = cwd + '/default of credit card clients.xls'
nanDict = {}
df = pd.read_excel(filename, header=1, skiprows=0, index_col=0, na_values=nanDict)

df.rename(index=str, columns={"default payment next month": "defaultPaymentNextMonth"}, inplace=Tr

# Features and targets
X = df.loc[:, df.columns != 'defaultPaymentNextMonth'].values
y = df.loc[:, df.columns == 'defaultPaymentNextMonth'].values

# Categorical variables to one-hot's
onehotencoder = OneHotEncoder(categorical_features = [3])
X = onehotencoder.fit_transform(X).toarray()
X = X[:, 1:]

# Train-test split
trainingShare = 0.5
seed  = 1
XTrain, XTest, yTrain, yTest=train_test_split(X, y, train_size=trainingShare, \
                                              test_size = 1-trainingShare,
                                             random_state=seed)

# Input Scaling
sc = StandardScaler()
XTrain = sc.fit_transform(XTrain)
XTest = sc.transform(XTest)

# One-hot's of the target vector
```

```
Y_train_onehot, Y_test_onehot = to_categorical(yTrain), to_categorical(yTest)

# Remove instances with zeros only for past bill statements or paid amounts

df = df.drop(df[(df.BILL_AMT1 == 0) &
                (df.BILL_AMT2 == 0) &
                (df.BILL_AMT3 == 0) &
                (df.BILL_AMT4 == 0) &
                (df.BILL_AMT5 == 0) &
                (df.BILL_AMT6 == 0) &
                (df.PAY_AMT1 == 0) &
                (df.PAY_AMT2 == 0) &
                (df.PAY_AMT3 == 0) &
                (df.PAY_AMT4 == 0) &
                (df.PAY_AMT5 == 0) &
                (df.PAY_AMT6 == 0)].index)

df = df.drop(df[(df.BILL_AMT1 == 0) &
                (df.BILL_AMT2 == 0) &
                (df.BILL_AMT3 == 0) &
                (df.BILL_AMT4 == 0) &
                (df.BILL_AMT5 == 0) &
                (df.BILL_AMT6 == 0)].index)

df = df.drop(df[(df.PAY_AMT1 == 0) &
                (df.PAY_AMT2 == 0) &
                (df.PAY_AMT3 == 0) &
                (df.PAY_AMT4 == 0) &
                (df.PAY_AMT5 == 0) &
                (df.PAY_AMT6 == 0)].index)

# Descriptive information
print('Number of empty elements in data: ', df.isnull().values.any())
print('Observations: ', df.shape[0])
print('Percentage defaults: ', df['defaultPaymentNextMonth'].astype(bool).sum(axis=0)/df.shape[0]
```

This is not the same number of observations as in Yeh and Lien (2009). Yeh and Lien (2009) have 25 000 observations. However, we have the same number of observations as in Pyzhov and Pyzhov (2017), which is said to use the same dataset as Yeh and Lien (2009).

The percentage of individuals with default is the same as in the Yeh and Lien (2009).

Will will essentially deal with two classes. For two classen and two paramaters we have

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

The solution is found by maximizing the likelihhod function

$$P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{n} \Big[p(y_i = 1|x_i, \hat{\beta})\Big]^{y_i} \Big[1 - p(y_i = 1|x_i, \hat{\beta}))\Big]^{1-y_i}$$

We will use the cross-entropy as objective function. The cross-entropy is the negative log of the likelihood function

C()=-$\sum_{i=1}^{n}\left(y_i(\beta_0+\beta_1 x_i)-\log\left(1+\exp\left(\beta_0+\beta_1 x_i\right)\right)\right)$.*Theminussigninthecross*−*entropyisinordertomakethisaconvexfunctionsothatwegetaminimizationproblem.*

To find the minimum of the above function, we differentiate it and set the result to zero

$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_0}=-\sum_{i=1}^{n}\left(y_i-\frac{\exp\left(\beta_0+\beta_1 x_i\right)}{1+\exp\left(\beta_0+\beta_1 x_i\right)}\right),$

$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_1}=-\sum_{i=1}^{n}\left(y_i x_i-x_i\frac{\exp\left(\beta_0+\beta_1 x_i\right)}{1+\exp\left(\beta_0+\beta_1 x_i\right)}\right).$

The above can be rewritten as $\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}}=-\hat{X}^T(\hat{y}-\hat{p}).$

We see that there is no explicit expression for the parameters, like we have in e.g. OLS. Hence iterative methods must be applied, and we will apply the gradient descent method which basically says for some variabe θ, we update it with

= - $\eta\nabla\hat{\theta}$.

The second term in the above equation is the gradient. In our case the gradient for the regression parameters are given by the linear algebra equation above

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

lmbdas=np.logspace(-5,7,13)
parameters = [{'C': 1./lmbdas}]
scoring = ['accuracy', 'roc_auc']
logReg = LogisticRegression()
gridSearch = GridSearchCV(logReg, parameters, cv=5, scoring=scoring, refit='roc_auc')
# "refit" gives the metric used deciding best model.
# See more http://scikit-learn.org/stable/auto_examples/model_selection/plot_multi_metric_evaluat
gridSearch.fit(XTrain, yTrain.ravel())

def gridSearchSummary(method, scoring):
    """Prints best parameters from Grid search
    and AUC with standard deviation for all
    parameter combos """

    method = eval(method)
    if scoring == 'accuracy':
        mean = 'mean_test_score'
        sd = 'std_test_score'
    elif scoring == 'auc':
        mean = 'mean_test_roc_auc'
        sd = 'std_test_roc_auc'
    print("Best: %f using %s" % (method.best_score_, method.best_params_))
    means = method.cv_results_[mean]
    stds = method.cv_results_[sd]
    params = method.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print("%f (%f) with: %r" % (mean, stdev, param))

gridSearchSummary('gridSearch', 'auc')
```

We create a function for printing the accuracy of the results and the so-called confusion matrices.

```
def createConfusionMatrix(method, printOut=True):
    """
    Computes and prints confusion matrices, accuracy scores,
    and AUC for test and training sets
    """
    confusionArray = np.zeros(6, dtype=object)
    method = eval(method)

    # Train
    yPredTrain = method.predict(XTrain)
    yPredTrain = (yPredTrain > 0.5)
    cm = confusion_matrix(
        yTrain, yPredTrain)
    cm = np.around(cm/cm.sum(axis=1)[:,None], 2)
    confusionArray[0] = cm

    accScore = accuracy_score(yTrain, yPredTrain)
    confusionArray[1] = accScore

    AUC = roc_auc_score(yTrain, yPredTrain)
    confusionArray[2] = AUC

    if printOut:
        print('\n####################  Training  ###############')
        print('\nTraining Confusion matrix: \n', cm)
        print('\nTraining Accuracy score: \n', accScore)
        print('\nTrain AUC: \n', AUC)

    # Test
    yPred = method.predict(XTest)
    yPred = (yPred > 0.5)
    cm = confusion_matrix(
        yTest, yPred)
    cm = np.around(cm/cm.sum(axis=1)[:,None], 2)
    confusionArray[3] = cm

    accScore = accuracy_score(yTest, yPred)
    confusionArray[4] = accScore

    AUC = roc_auc_score(yTest, yPred)
    confusionArray[5] = AUC

    if printOut:
        print('\n####################  Testing  ###############')
        print('\nTest Confusion matrix: \n', cm)
        print('\nTest Accuracy score: \n', accScore)
        print('\nTestAUC: \n', AUC)

    return confusionArray


confusionArrayLogreg = createConfusionMatrix('gridSearch', printOut=False)

import matplotlib.pyplot as plt
import seaborn
import scikitplot as skplt

seaborn.set(style="white", context="notebook", font_scale=1.5,
            rc={"axes.grid": True, "legend.frameon": False,
"lines.markeredgewidth": 1.4, "lines.markersize": 10})
seaborn.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 4.5})

yPred = gridSearch.predict_proba(XTest)
print(yTest.ravel().shape, yPred.shape)
```

```
skplt.metrics.plot_cumulative_gain(yTest.ravel(), yPred)

defaults = sum(yTest == 1)
total = len(yTest)
defaultRate = defaults/total
def bestCurve(defaults, total, defaultRate):
    x = np.linspace(0, 1, total)

    y1 = np.linspace(0, 1, defaults)
    y2 = np.ones(total-defaults)
    y3 = np.concatenate([y1,y2])
    return x, y3

x, best = bestCurve(defaults=defaults, total=total, defaultRate=defaultRate)
plt.plot(x, best)

plt.show()
```

The Pulsar classification case

```
import numpy as np
from sklearn import metrics
from sklearn import linear_model
import sklearn
import random
import time as tm
import xlrd


from sys import platform as sys_pf
if sys_pf == 'darwin':
    import matplotlib
    matplotlib.use("TkAgg")
from matplotlib import pyplot as plt


def predict(x,beta):
    """
    Function that predicts labels based on the beta parameters that has been
    calculatted.
    Inputs: datapoints and beta parameters
    Output: predicted label
    """
    ypred = x @ beta
    ypred = 1 / (1 + np.exp(-ypred))
    if np.isnan(np.sum(ypred)):
        ypred_0 = np.zeros((ypred.shape))
        return ypred_0
    ypred[ypred < 0.5] = 0
    ypred[ypred >= 0.5] = 1
    return ypred


def beta_update(lr,x,y,beta):
    """
    Function to calculate the new beta parameters without regularization.
    Inputs: data, labels, beta parameters
    Output: updated beta
    """
    output = 1 / (1 + np.exp(-(x @ beta)))
    delta = x.T @ (output - y.reshape((len(y),1)))
    new_beta = beta - lr * delta/len(y)
```

```
    return new_beta

def beta_update_l2(lr,x,y,beta,lamb):
    """
    Function to calculate the new beta parameters with l1 regularization.
    Inputs: data, labels, beta parameters, lambda
    Output: updated beta
    """
    output = 1 / (1 + np.exp(-(x @ beta)))
    delta = x.T @ (output - y.reshape((len(y),1)))
    new_beta = beta - lr * (delta/len(y) + lamb * beta)
    return new_beta

def confusion(targets, pred_targets):
    """
    Calculates the onfusion matrix for the SVM.
    Inputs: Predicted and correct labels that are going to be assessed.
    Outputs: confusion matrix.
    """
    #create confuison matrix
    conf = np.zeros((2,2))


    #Remove redundant dimensions
    sq_targets = np.squeeze(targets).astype(int)
    sq_pred_targets = np.squeeze(pred_targets).astype(int)


    #for loop runs through the test input
    for i in range(0,targets.shape[0]):


        #Increments the values in the confusion matrix
        #based on the results
        conf[sq_targets[i]][sq_pred_targets[i]] = conf[sq_targets[i]][sq_pred_targets[i]] + 1


    return conf

#adjustable parameters
trainp = 0.5 #percentage of the data to be used for training
minibatch = 10 #minibatch size
lr = 0.2 #learning rate
lamb = [0.00001, 0.0001,0.001, 0.01, 1, 10,100] #value of lambda
#lamb = [0.00001, 0.0001,0.001, 0.01, 1,10,100,1000,10000] #a larger lambda
boot_runs = 100 #number of bootstrap runs

#Get data
data = np.genfromtxt('pulsar_stars.csv', delimiter=',',skip_header=1)
target = data[:,8]


#Subtract the mean for each inout
data[:,:8] = data[:,:8] - np.mean(data[:,:8], axis=0, keepdims=True)

#Divide the data by the max to reduce the size of the inputs
#max_abs_data = np.std(data[:,:23],axis=0,keepdims=True)
max_abs_data = np.max(data[:,:8], axis=0, keepdims=True)
data = data[:,:8]/max_abs_data


# Randomly order the data
order = list(range(data.shape[0]))
np.random.shuffle(order)
```

```
data = data[order,:]
target = target[order]

#find total samples and calculate the number of training data
sampn = len(target)
trainn = int(sampn*trainp)


#split the data into training and test sets
xt = np.zeros((trainn,data.shape[1]+1))
xt[:,0] = -1
xt[:,1:] = data[:trainn,:]
yt = target[:trainn]

xte = np.zeros((sampn-trainn,data.shape[1]+1))
xte[:,0] = -1
xte[:,1:] = data[trainn:,:]
yte = target[trainn:]



#array to store the ac scores
train_ac = np.zeros(len(lamb))
test_ac = np.zeros(len(lamb))
train_ac_l2 = np.zeros(len(lamb))
test_ac_l2 = np.zeros(len(lamb))
train_ac_sci = np.zeros(len(lamb))
test_ac_sci = np.zeros(len(lamb))



count = 0
for l in lamb:
    #array to store ac score in bootruns
    b_train_ac = np.zeros(boot_runs)
    b_test_ac = np.zeros(boot_runs)
    b_train_ac_l2 = np.zeros(boot_runs)
    b_test_ac_l2 = np.zeros(boot_runs)
    b_train_ac_sci = np.zeros(boot_runs)
    b_test_ac_sci = np.zeros(boot_runs)
    for j in range(boot_runs):

        #use scikit to take randoms samples with replacement
        x_boot, y_boot = sklearn.utils.resample(xt,yt)

        #initialize the beta parameters
        beta = (2/np.sqrt(data.shape[1]+1)) * np.random.random_sample((data.shape[1]+1,1)) -1/np.sqrt(
        beta_l2 = (2/np.sqrt(data.shape[1]+1)) * np.random.random_sample((data.shape[1]+1,1)) -1/np.sq

        #variable to store the best score
        best_score = 0
        best_score_l2 = 0
        for k in range(0,50):

            for i in range(0,trainn,minibatch):

                #update beta parameters
                beta = beta_update(lr,x_boot[i:i+minibatch,:],y_boot[i:i+minibatch],beta)
                beta_l2 = beta_update_l2(lr,x_boot[i:i+minibatch,:],y_boot[i:i+minibatch],beta_l2,l)

            #checks the score of the model and stores the best parameters
            temp_pred = predict(x_boot,beta)
            temp_score = np.sum(y_boot.reshape((trainn,1)) == temp_pred) / len(y_boot)
```

```
            if temp_score > best_score:
                best_beta = beta
                best_score = temp_score

            temp_pred = predict(x_boot,beta_l2)
            temp_score = np.sum(y_boot.reshape((trainn,1)) == temp_pred) / len(y_boot)
            if temp_score > best_score_l2:
                best_beta_l2 = beta_l2
                best_score_l2 = temp_score


            #reshuffle the data so the model does not train the same way
            order = list(range(np.shape(x_boot)[0]))
            np.random.shuffle(order)
            x_boot = x_boot[order,:]
            y_boot = y_boot[order]

        #predicts the labels using beta
        ypred = predict(xte,best_beta)
        ypred_train = predict(x_boot,best_beta)


        #predicts the labels using beta_l2
        ypred_l2 = predict(xte,best_beta_l2)
        ypred_train_l2 = predict(x_boot,best_beta_l2)


        #fiting a scikit model
        scilearn = linear_model.LogisticRegression(penalty='l2',C=1/l).fit(x_boot, y_boot)


        #calualte the score of the predicted labels
        b_test_ac[j]= (np.sum(yte.reshape((sampn - trainn,1)) == ypred) / len(yte))*100
        b_train_ac[j]= (np.sum(y_boot.reshape((trainn,1)) == ypred_train) / len(y_boot))*100


        b_test_ac_l2[j]= (np.sum(yte.reshape((sampn - trainn,1)) == ypred_l2) / len(yte))*100
        b_train_ac_l2[j] = (np.sum(y_boot.reshape((trainn,1)) == ypred_train_l2) / len(y_boot))*100


        #calualte the score of the scikit models
        b_test_ac_sci[j] = scilearn.score(xte,yte) * 100
        b_train_ac_sci[j] = scilearn.score(x_boot,y_boot) * 100


    #Stor the mean of the accuracy of each run in the bootstrap
    train_ac[count] = np.mean(b_train_ac)
    test_ac[count] = np.mean(b_test_ac)
    train_ac_l2[count] = np.mean(b_train_ac_l2)
    test_ac_l2[count] = np.mean(b_test_ac_l2)
    train_ac_sci[count]= np.mean(b_train_ac_sci)
    test_ac_sci[count] = np.mean(b_test_ac_sci)

    #print results
    print("Created minibatch method:")
    print("Train score: %.4f" %train_ac[count])
    print("Test score: %.4f" %test_ac[count])
    print("Test: Max score: %f Min score: %f\n" %(np.max(b_test_ac),np.min(b_test_ac)))

    print("Created minibatch with L2 regularization lambda = %.5f:" %l)
    print("Train score: %.4f" %train_ac_l2[count])
    print("Test score: %.4f" %test_ac_l2[count])
    print("Test: Max score: %f Min score: %f\n" %(np.max(b_test_ac_l2),np.min(b_test_ac_l2)))
```

```
    print("Scikit learn method lambda = %.5f:" %l)
    print("Train score: %.4f" %train_ac_sci[count])
    print("Test score: %.4f" %test_ac_sci[count])
    print("Test: Max score: %f Min score: %f\n" %(np.max(b_test_ac_sci),np.min(b_test_ac_sci)))


    print("--------------\n")



    count += 1
plt.figure(1)
# Plot our performance on both the training and test data
plt.semilogx(lamb, train_ac, 'b',label='Created method train')
plt.semilogx(lamb, test_ac,'--b',label='Created method test')
plt.semilogx(lamb, train_ac_l2,'r',label='Created L2 train',linewidth=1)
plt.semilogx(lamb, test_ac_l2,'--r',label='Created L2 test',linewidth=1)
plt.semilogx(lamb, train_ac_sci, 'g',label='Scikit train')
plt.semilogx(lamb, test_ac_sci, '--g',label='Scikit test')



plt.title("Accuracy scores for test and training data with different values for lambda", fontsize = 16
plt.legend(loc='lower left',fontsize=16)
plt.xlim([min(lamb), max(lamb)])
plt.xlabel('Lambda',fontsize=15)
plt.ylabel('Accuracy score [%]',fontsize=15)
plt.tick_params(labelsize=15)

plt.show()
```