Project 2 on Machine Learning: Classification and Regression

Bjørn Grønntun

November 8, 2019

Abstract

1 Classification: Predicting credit card default

1.1 The data

1.1.1 Context and domain description

Being able to predict whether or not a client is likely to default on payments of his or her debt is an important task for banks and other financial institutions. In this particular case, we have information on a sample of customers and their history of payment.

1.1.2 Description of dataset

According to the description on https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients, the dataset has the following attributes:

- X1: Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.
- X2: Gender (1 = male; 2 = female).
- X3: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).
- X4: Marital status (1 = married; 2 = single; 3 = others).
- X5: Age (year).
- X6 - X11: History of past payment. We tracked the past monthly payment records (from April to September, 2005) as follows: X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005; . . .;X11 = the repayment status in April, 2005. The measurement scale

for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . .; 8 = payment delay for eight months; 9 = payment delay for nine months and above.

- X12-X17: Amount of bill statement (NT dollar). X12 = amount of bill statement in September, 2005; X13 = amount of bill statement in August, 2005; . . .; X17 = amount of bill statement in April, 2005.
- X18-X23: Amount of previous payment (NT dollar). X18 = amount paid in September, 2005; X19 = amount paid in August, 2005; . . .;X23 = amount paid in April, 2005.

Furthermore, the dataset contains one response (target) variable having the value 1 for default (failure to pay) and 0 for non-default.

1.1.3 Variable transforms

The functions `get_clean_data_frame`, `get_design_matrix` and `get_target_values` in `preprocessing/preprocessing.py` reads the data from the Excel file and performs some simple transformations on the data. We consider the columns `'SEX', 'EDUCATION', 'MARRIAGE', 'PAY_0', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6'` as categorical and transform them using one-hot encoding. This encoding scheme entails replacing one categorical column containing k different values with k columns, each having the value 1 if an instance belongs to the corresponding category and 0 otherwise. The remaining columns are treated as numerical, and are scaled using `StandardScaler` from `sklearn.preprocessing`. We do not perform further feature engineering here, even though that might improve our results.

Our goal in the following is to predict whether a given customer will default or not on his or her debt. Most of the work is done in the notebook `classification_models`. There, we begin by reading the data and splitting it into a training set and a test set. In order to measure the quality of our models, we will use accuracy, which is defined as the percentage of observations correctly classified by our models. As most customers do not default on their debt, a very simple model would be simply to predict that no one defaults. This is done in the notebook, and it actually leads to an accuracy of 0.7747, which on the face of it does not seem that bad! We will use this accuracy as a performance benchmark in the sense that any models that obtains a lower test accuracy is deemed worthless.

1.2 The models

1.2.1 Logistic regression

Logistic regression is a method for binary classification that tries to model the posterior probability of a given instance belonging to one of two classes. Given an input vector $x^{(i)}$, representing an observation, we compute $p^{(i)} = \frac{1}{1+e^{-w^T x^{(i)}}}$

and assign the observation to the negative class if $p^(i) < 0.5$ and to the positive class otherwise. Here, w is a weight vector of real numbers. This simple way of predicting class labels is implemented in the `predict` method of the `LogReg` class in `logistic.py`.

The weights are learned by maximum likelihood estimation, that is, by maximizing the quantity

$$\prod_{i:y_i=1} p^{(i)} \prod_{i:y_i=0} (1 - p^{(i)}) \tag{1}$$

This is equivalent to minimizing the cross entropy

$$-\sum_{i=1}^{m} (y^{(i)} w^T x^{(i)} - \log(w^T x^{(i)})) \tag{2}$$

This minimization problem does not have a closed-form solution, but can be solved using iterative methods. In our own code, we use a form of gradient descent in which we pick a batch of training data X and update the weights according to

$$w := w - \eta \frac{1}{m} X^T (\sigma(X) - y) \tag{3}$$

where η is the learning rate and $\sigma(X)$ is the output of the logistic (sigmoid) function on each row of X. In order that the algorithm stop iterating at the right moment, we set aside a validation set and compute the accuracy of the algorithm's predictions on the validation set after each iteration.

In the notebook `classification_models`, we first perform logistic regression using `LogisticRegression` from `sklearn.linear_model`, immediately obtaining a test accuracy of 0.8158. We then turn to our own implementation, which can be found in `models/logistic.py`. We perform training on mini-batches of size 128, and stop training when the accuracy on the validation set exceeds 0.82. This actually happens after 223000 iterations, leading to a test accuracy of 0.8126, rather close to the results obtained by `sklearn`. Further optimization and hyperparameter tuning (learning rate, batch size) is possible, but the results seem to be quite good for such a rather simple model.

1.2.2 Neural networks

A neural network can be thought of as a large system of interconnected nodes, each of which recieves input from and sends output to other nodes. The simplest version, and the one with which we will be working, is known as the multi-layer perceptron. Here, prediction is done in the following way. Given an input vector x, we multiply by a weight matrix $W^{(1)}$, add a bias vector $b^{(1)}$ and apply an activation function $f^{(1)}$, obtaining a vector $f^{(1)}(x^T W^{(1)} + b^{(1)})$ which can then be fed to the next layer, where a new weight matrix $W^{(2)}$, a new bias vector $b^{(2)}$ and a new activation function $f^{(2)}$ awaits it. This process continues until we reach the output layer, where we usually apply the softmax activation function so that the output can be interpreted as the probabilities of the input vector belonging to each of the possible classes. We can then simply predict the class with the highest probability.

In the notebook `classification_models` we use `Keras` to build a simple neural network with three layers, the two first having 32 neurons and a ReLU activation function, whereas the last layer has only 1 neuron and use the sigmoid activation. We obtain an accuracy of 0.8119 on the test set.

We now try do do something similar with our own implementation (which is to be found in `src/models/neural.py`). In order to avoid overfitting and hopefully get a better model, we use a kind of regularization where we subtract a small multiple of the weights for each node (with the exception of the bias), thus bringing the weights closer to zero. In addition to that, we use ordinary back-propagation and subtract the computed gradient multiplied by the learning rate. The best results we were able to obtain were with randomly chosen mini-batches of size 128 for training, a learning rate of 0.1 and regularization factor of 0.0091. We use early stopping and stop iterating when the accuracy on the validation set reaches 0.83. The accuracy on the test set then turns out to be 0.814.

2 Regression: Predicting...