Machine Learning Report
Project 1

Boujdaria Omar, Ndoye Mohamed, Rinaldi Vincent
EPFL - Fall 2018

INTRODUCTION

The aim of this project is first to learn using the concepts we have seen in the lectures so far, and practiced in the labs, on a real-world problem which is the Higgs-Boson Dataset.

In the next sections, we summarize and outline each step that made up our work all along the achievement of this project, from implementing the basic regression functions, to generating a working model describing our predictions, but also performing feature processing and taking actions according to the fitting of our method to the data set.

I. DATA INSPECTION

The datasets were originally released by CERN, and were used in the search of the Higgs Boson.

The Train set contains 250k rows, while the Test set contains 568k rows, and there are 30 feature columns in total, all being real-valued.

11 of the 30 features contain missing data, which is represented by the value -999. One should also note that the feature 'PRI_jet_num' is categorical, having values in $\{0, 1, 2, 3\}$.

II. IMPLEMENTATION OF REGRESSION FUNCTIONS

We first had to implement six different regression functions used to establish our predictions for the actual data set.

Those functions can take as arguments :

- y : vector of data set labels
- tx : matrix of data set features
- lambda_ : regularization parameter
- initial_w : vector of initial weight parameters
- max_iters : number of iterations for computation
- gamma : step size for weight update rule

Every functions are written to `implementations.py`. Some of them use different external functions such as *compute_mse*, *compute_gradient*, *batch_iter*, *sigmoid*, *calculate_loss_log*, *calculate_gradient_log*, *calculate_hessian_log*, or *penalized_logistic_regression*, that are located in the same file.

A. Least_Squares_GD(y, tx, initial_w, max_iters, gamma)

This function, at each iteration, will update the weight vector, starting from *initial_w*, by calculating the gradient vector (via *compute_gradient* function), and then by using the weight update *gamma*.

B. Least_Squares_SGD(y, tx, initial_w, max_iters, gamma)

This function does the same calculations as *Least_Squares_GD*, except that random batches are generated for iterating over the data.

C. Least_Squares(y, tx)

This function solves the linear system $X^T X w = X^T Y$ (for *X = tx* and *Y = y*) by using QR decomposition to compute the optimal weight vector w.

D. Ridge_Regression(y, tx, lambda_)

This function has the same behavior as *Least_Squares*, but in this case, the linear system to solve is now, such that λ' = 2 * *lambda_* * data-size : $(X^T X + \lambda' I)w = X^T Y$.

E. Logistic_Regression(y, tx, initial_w, max_iters, gamma)

This method computes the cost by relating it to the MAP (Maximum A Posteriori) rule. The predictions are mapped from real values into probabilities by using the logistic function $\sigma(x) = \frac{e^x}{1+e^x}$. Then the weight is chosen to minimize the cost function.

F. Reg_Logistic_Regression(y, tx, lambda_, initial_w, max_iters, gamma)

This method is the same as *Logistic_Regression*, but here we add a penalty term, that is $\frac{\lambda}{2}\|w\|^2$ for the cost.

All of these functions return a tuple *(w, loss)*, where *w* is the last calculated weight vector of the method, and *loss* the corresponding loss value calculated with the cost function *compute_mse* (*Mean Square Error*) or, especially for *Logistic_Regression*, *calculate_loss_log*, and for *Reg_Logistic_Regression*, *penalized_logistic_regression*.

III. Preprocessing and Data Cleaning

The preprocessing involved different steps. The main one was to handle the erroneous values -999.0 inside the provided sets. We first turned them to numpy NaN values, then we trained a **ridge regression** on the different columns that were fully available (i.e. not containing any "NaN") to predict each one of those NaN values in both train and test sets, by using the function *Train_Pred_Missing_Col*, located in the `preprocessing.py` file, that we wrote for this purpose.

Then we expanded our feature-matrix columns by using the functions *Build_Poly_Mtrx*, and *Build_Trig_Mtrx*, that are located in the `cross_validation.py` file, which respectively apply powers, and trigonometric functions (as cosinus, sinus and tangent) on values of already existing columns to form and append new columns to our whole set, and to catch any non-linear relation between the dependent and the independent variables.

Finally, we standardized our data before any calculations.

IV. Generating Predictions

In this part, we explain how we chose the best parameters and model to use in order to generate the best possible predictions. With the intention of choosing the right parameters and, at the same time, avoiding over fitting our model, we used the **K-Fold Cross-Validation** technique, which separates the train set into k subgroups, and run k regressions, each time using a different subgroup as the test set, and the remaining ones being the training set. K-Fold Cross-Validation returns an unbiased estimate of the generalization error.

Furthermore, the Cross-Validation technique then helped us choosing the best *lambda_* parameter (penalty coefficient) for which we picked values in the interval $[10^{-9}, 1]$ for our local Cross-Validation test, but also the best *degree* parameter (for the polynomial transformation of our columns) for which we tested values from 2 to 10. We were looking for the most optimal values for these two parameters, because we decided to compute our optimal weight vector *w* using **ridge regression**.

According to the results of our local Cross-Validation test, we have set *lambda_* = 10^{-8} and *degree* = 10, which then seem to be the best parameters generating the lowest generalization error. Therefore we used those specific values as arguments of our regression technique.

For our case, it seems that the other regression techniques produce bigger losses, and that's why we made the choice of using ridge regression.

Results

With the chosen model and parameters, the precision score, which is calculated on 50% of the test set, is 82.111%, according to the Kaggle report, which has been the best score online we could reach so far.