Machine Learning Project 1 - Fall 2020

Julien Hu, Matthieu Masouye, Sebastien Ollquist
Department of Computer Science, EPFL, Switzerland

***Abstract*—Implement basic Machine Learning methods on a given data set and analyze the predictions from them in order to compare the different methods and determine which is best.**

I. Introduction

The goal of this first mini project is to implement all basic Machine Learning methods on a given data set, optimize them and analyze the results we obtained. Essentially, the demanded algorithms are:

1) Linear regression using Gradient Descent and Stochastic Gradient descent
2) Least squares regression and ridge regression using normal equations
3) Logistic regression using Gradient Descent
4) Regularized logistic regression using Gradient Descent

II. Data normalization

Before performing any algorithm, we essentially have to clean the data in order for us to use it correctly and generate accurate predictions. Our data standardization consists mainly of three parts:

1) First of all, we have to clean the data. Indeed, the samples of the dataset contained some missing values, represented by -999. We thus mask them using the function *numpy.ma.masked_array* which will allow us to modify all these values later.
2) We then normalize the training data. For each data sample X, we want to compute the value $Y = (X - \mu)/\sigma$ where μ is the computed mean and σ the standard deviation. As for the missing values, we simply replace them by 0, the mean of the normalized dataset.
3) At last, we want to add a 1s in front of the X^T matrix. This represents the bias which allows the linear function to get away from the origin.

There is however an additional step for the logistic regression. We have to modify the labels so that the classes are 0's and 1's instead of -1's and 1's. This is needed to prevent the algorithm from not working properly. Note that we also have to modify their predictions back to -1 and 1 for the submission.

III. Algorithms implementations

A. Linear regression

Linear regression consists of taking a dataset that contains two different data point types and split them using a line described by a linear function in order to divide the points the best way possible. We have performed two different implementations of it: one using Gradient Descent (GD) and the other one using Stochastic Gradient Descent (SGD).

The implementation of GD and SGD involve computing the gradient and the loss for each data sample we go through. The only difference with SGD is that we perform the GD algorithm on one data sample instead of the whole dataset.

B. Least squares regression

The least squares regression involves solving a linear system of equations. The standard way to perform the algorithm is to solve the equation $X^T(y - Xw) = 0$. We can rewrite this equation as $X^T Xw = X^T y$ in order to easily use the numpy function *numpy.linalg.solve(a,b)*. Setting $A = X^T X$ and $b = X^T y$, we get the system $Ax = b$ which is then easy to solve using the provided numpy function.

C. Ridge regression

Ridge regression essentially works the same way as does least squares except from the fact that we add a tradeoff parameter λ. This regularization term discourages large values in the weights vector, as we assume they are less likely to be right.

1) If λ is small, we will obtain a small bias but a large variance. This implies overfitting.
2) If λ is large, we will obtain a large bias but a small variance which implies underfitting.

Setting λ to 1 should give us roughly the same loss as in the least squares regression.

Note that to implement the standard ridge regression algorithm, we also have to solve the normal equations. The derivation is the same as the one done before.

D. Standard and regularized logistic regression

Logistic regression is used to predict a value y given a data label x by computing the probability for it to be in the correct class, by using the sigmoid function. The weights are updated each iteration using gradient descent on the logistic loss.

Regularized logistic regression adds a regularization term to the mix, which penalizes large weights vector. This is particularly useful when the data is linearly separable, because then the weights $w \to \infty$.

E. Cross-validation

Here are the two methods of cross-validation that we used to optimize meta-parameters:

1) Simple cross-validation: This method consists in splitting the training dataset in two, according to a given ratio: One set will be used to train the algorithm as usual, while the other will serve as a validation set: we use the algorithm to make predictions about this second dataset, and check how accurate they are.

2) K-fold cross-validation: Here we split the training dataset into k groups: Then we select one of these groups. The other $k-1$ groups will be used for training the algorithm, while the chosen group will be used to test the predictions of the algorithm. We do that for each group, and average the training loss, testing loss, and the weights vector computed for these k groups.

These methods allow us to better choose the meta-parameters of our models, because they take into account how the algorithm performs when making prediction, instead of just how well it fits the data; this notably helps prevent overfitting. K-fold cross-validation is considered the best method out of the two, but it is more expensive, and wasn't used where it would slow down our computers too much.

IV. Results obtained

A. Gradient descent

We have decided to set our learning rate equal to $\gamma = 10^{-2}$ after some testing by hand. The algorithm then converges quite rapidly to a loss value of approximately 0.38.

B. Stochastic gradient descent

We execute the SGD algorithm which is mostly the same as GD but this time acts on only one sample (the batch size is 1 in that case). The algorithm for a same γ factor converges to a loss value of 0.11.

The model has achieved an accuracy of 0.403 and an F1-score of 0.384 on AIcrowd, a worse score than the GD algorithm which had an F1-score just under 0.5 but an accuracy of 0.693. This is probably due to the fact that SGD learns from way less data samples than GD so it is more sensible to outliers. Also, the learning rate doesn't affect the loss that much, again because less data is treated.

C. Least squares regression

The algorithm is executed rapidly and gives us an average loss value of approximately 0.34 which is comparable to the previous implementation of GD.

D. Ridge regression

We implemented the ridge regression algorithm, and optimized it using 10-fold cross-validation; We tested λ's going from 10^{-6} to 1, with 0.0001623 giving the best testing loss of 0.34.

The testing loss is going slightly up with a smaller λ, and diverges with a bigger λ; So while the algorithm seems to somewhat benefit from the regularization, there doesn't seem to be a lot of overfitting to fight with this algorithm.

The model achieved an accuracy of 0.745 and an F1 score of 0.569 on AIcrowd.

E. Logistic Regression

We implemented this algorithm and optimized it in the following ways: Instead of just returning the last weights and loss, we calculate an average weight vector and loss every 100 iterations, and return the best ones, to reduces outliers' effect. We also use simple cross-validation, with γ's ranging from 10^{-5} to $4 * 10^{-4}$; $6 * 10^{-5}$ gave us the best testing loss of 0.50144 (Keep in mind this isn't the same loss function as the other algorithms).

Smaller γ's made the loss go back up, while bigger ones made the loss quickly diverge and overflow. It is interesting to note how small the learning rate it, and how close the testing and learning rate are; this means there's probably not a lot of overfitting, and thus we don't expect the regularized version to do much better. Also, the algorithm converged for most of the γ's under 2000 iterations.

The model achieved an accuracy of 0.751 and an F1 score of 0.596 on AIcrowd.

F. Regularized Logistic Regression

We implemented this algorithm and optimized it with simple cross-validation. The gamma was set at $6 * 10^{-5}$, as it was the best learning rate for the un-regularized version. Tested λ's ranged from 50 to 0.01, with 10 achieving the best test loss of 0.50144.

Throughout the λ's, the testing loss didn't change by much, and in fact stayed very close to the unregularized version. However, the training loss augmented with the λ's; In particular, the best test result came from a relatively high λ, which didn't yield the best training loss.

The model achieved an accuracy of 0.751 and an F1 score of 0.595 on AIcrowd.

V. Conclusion

Overall, whilst we don't have the best scores on AIcrowd, we think we still achieved our goal: all our algorithm have learned during training, and they behaved as we expected when comparing their results: "Better" and more advanced algorithms gave us better result, with logistic regression achieving our best score.

As we have seen, we don't think we are overfitting our data, instead we think we are underfitting it. Had we more time and submissions available, we could have fine tuned more our models; we are thinking most notably of using K-fold cross-validation on all our models, and playing with the decision boundary of the sigmoid in the logistic regression (E.g. prediction under 0.1 are set to -1 instead of prediction under 0).

All results can be reproduced using run.py. Everything is written in the README file. Additionally, it will produce more in-depth information and graphs that we couldn't include in this report due to size constraint.