Higgs Boson Machine Learning challenge

Vignoud Julien, Lepeytre Hugo, Benhaim Julien

***Abstract*—This paper discusses our approach to the Higgs Machine Learning challenge as part of the CS-433 course at EPFL. This challenge consists of classifying data collected in the CERN particle accelerator into two categories: background noise and relevant signal [1].**
Machine Learning has been used for a long time by physicists to assist them in their analysis. This challenge connects the two research fields by offering a way to detect a Higgs Boson decaying into other particles even if this signal is buried in background noise.
In the end ee were able to train a model with an accuracy of 82% using a logistic regression model.

I. Introduction

The dataset is composed of 250 000 labeled experiments, containing 30 features representing several physical properties (like the estimated mass of the particles) that are either raw measurements, or values derived by the physicists at ATLAS. However, no knowledge of the underlying physic theory is required to understand the challenge and this paper.

What's more, over 500 000 unlabeled data points were provided as a test set, on which we could make predictions with our model and have them verified by an online platform (AICrowd)

II. Models and Methods

A. Data exploration

The first step in creating a machine learning model is to explore the data and process it. Note that all transformations were applied to both the train and test sets.

First, we transformed the -999 values into NaN, indicating missing or meaningless values [1]. We noticed that the NaN values in the columns depended on the category given by the value of the PRI_jet_num feature, as we can see in Figure 1. After separating the data into the 4 categories, we removed the columns full of NaN values. Then only the first column still contained some NaN values that were filtered out. In the test set, remaining NaN values were set to 0.

Secondly, outliers in the training set were removed, data was standardized, resulting in a mean of 0 and standard deviation of 1, and a bias term was added. Again, note that the data standardization was applied to both sets using the mean and standard deviation of the train set both times.

Noticing the heavy-tail distributions of some features, we tried to apply a logarithm to the relevant distributions, in order to reduce the amount of extreme values. However the accuracy decreased. We theorize that some extreme values may not have been considered as outliers anymore due to the log, and so were not filtered out.

We also tried to remove the highly correlated features, however the accuracy slightly decreased.

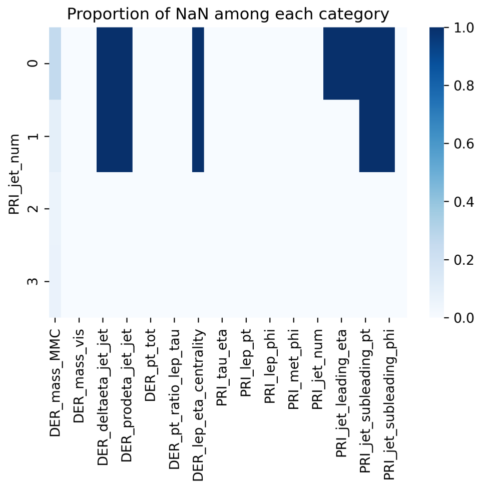

Figure 1. NaN proportion in each categories for some features. We can see that apart from the first feature, the proportion of NaN is either 1 or 0.

B. Model selection

Dividing the feature points in categories implies fitting a different model for each category. For predicting data, we would then use the right model according to the category of the datapoint. From the models studied in the lab, we decided to use only two:

1 - Least-squares regression/Ridge regression:
Since this model analytically minimises the mean square error, it is better that the gradient descent methods, and thus we don't need to work with those. We would then decide based on cross-validation results if the model needed a regularization parameter, in which case ridge regression would be used instead.

2 - Logistic regression :
This model uses negative log-likelihood as its loss function instead of mean square error, which is more appropriate for a classification problem. The downside is that the minimization of the loss function can not be solved analytically. We had to use a gradient descent instead, in this case stochastic, so that training the model would not be too slow, and we could perform cross-validation on it. Penalized logistic regression was not needed here because we noticed that the data would not be linearly separable (since we don't reach 100% accuracy and the weights magnitude doesn't increase to infinity).

C. Hyperparameter tuning

The next step was hyperparameter tuning. To make this step easier, we created k-fold cross-validation pipeline to easily try several ranges of values for hyperparameters and find the best of them for each model.

III. Results

In this section, we give some numerical results and explain our reasoning for the decision on our final model. Note that since we separated the data into 4 categories, we fine-tuned each model independently, so that if the differences were significant we could train a model with different hyperparameters for each category. Finally, we used the accuracy as a metric for fine-tuning: since the loss functions are different for the two models, this allows use to compare their results, which would not be possible using their respective losses.

First of all, to tune each hyperparameter, the number of folds in the k-fold cross-validation process was 4, a good compromise between execution time and number of repetitions.

A. Fine-tuning least-squares model

Since the basic least-squares model has no hyperparameters, we experiment with polynomial expansion of the data by creating new features x_i^k for each existing feature x_i except the bias.

Our results were as follows : Polynomial expansion increases accuracy, up to a point (between degrees 10 and 12 depending on the category) where both testing and training accuracies start to decrease. Since both are going down, this is not an overfitting problem. Our theory is that with high-degree polynomial expansion, the matrix $X^T X$ becomes singular, and its rank goes down with the degree, so we have to use the numpy function 'lstsq' instead of 'solve' which give a less and less precise approximation of the analytical optimum for the loss, and thus our accuracy can only decrease.

To try and fix the problem of $X^T X$ singularity, we switched the model to ridge regression, which creates a matrix $X^T X + \lambda I$ which is always non-singular, with λ big enough. But since we had no overfitting to correct, we can keep lambda very low.

Using ridge regression, we see in Figure III-A that indeed, the big drop in accuracy disappears, and we can find slightly better optimal solutions with higher-degree expansion.

B. Fine-tuning logistic regression model

For this model, we start by finding the best γ value with 300 iterations, and for polynomial expansion of the features of degrees 1 to 3.

We then fine-tune the number of iterations of gradient descent, but it does not change much with respect to the accuracy.

C. Comparison

For each category, we searched for the model with the best accuracy. The different values are listed in Table I.

In the end, even though the difference in our cross-validation was very low, the ridge regression worked way better on the test set with degrees [2, 2, 2, 2] than [5, 15, 5, 15] (the best degrees found by cross-validation). We don't

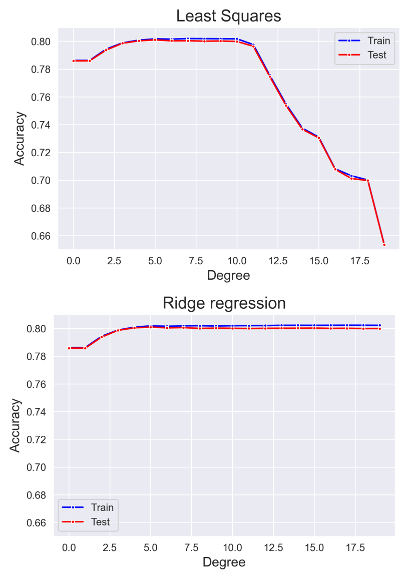

Figure 2. Cross-validation with Least Squares and Ridge Regression with lambda 0.003 on the polynomial expansion degree on category 1. We can see that ridge regression's accuracy is more stable when the degree increases

Table I
Best model hyper-parameters by category

	Ridge Regression		Logistic Regression	
Category	λ, degree	Accuracy	γ, degree, max_iter	Accuracy
1	0.00027, 2	0.800	0.1179, 3, 1000	0.802
2	0.00019, 2	0.782	0.0848, 2, 1000	0.781
3	0.0010, 2	0.811	0.1179, 3, 1000	0.807
4	1.3e-5, 2	0.816	0.0610, 3, 1000	0.814

know why this is the case, but the final results with the values of the table were 0.819 for logistic regression, and 0.810 for ridge regression. So we left both models available in run.py, but select the logistic regression as our preferred model.

IV. Conclusion

Through this project, we had the opportunity to work on some "real-world" machine learning that we didn't get in the labs. We had a large degree of freedom, so we could try different ideas, model to aim for a better precision. We also faced new challenges like treating data not prepared for machine learning with missing values and outliers. We also noticed some unexpected behavior such as overflow in training that we mostly solved with normalized features. This reminds us that applying theory to a model isn't as straightforward as it looks. We couldn't always explain the behavior of our model, such as a big difference between our test accuracy and the AICrowd grading.

To get a better accuracy, we think that it would be more appropriate to use some more complex model like neural network or random forest. We are looking forward to tink with them on the second project to appreciate their power.

References

[1] C. Adam-Bourdarios, "Learning to discover: the higgs boson machine learning challenge."