Higgs Boson Classification Report

Github repository: https://github.com/DataExplorers-AUB/Higgs-Boson-Classification

Team:

Lilit Musheghyan (https://github.com/1ilit)
Sam Hamamji (https://github.com/SamHamamji)

A) Data preprocessing

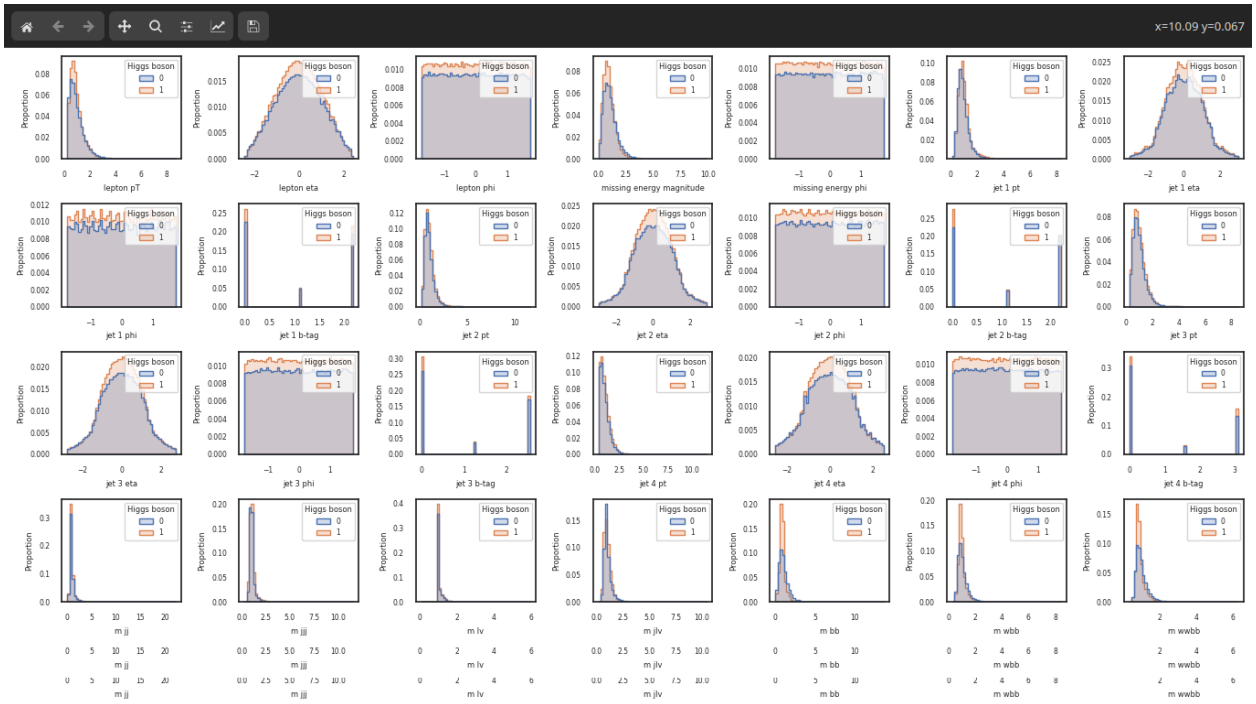
Data cleaning:

- There were three null values, we dropped their rows because it's a ridiculously small fraction of the dataset.
- There was one value that was a string representation of a number, we converted it to a float.

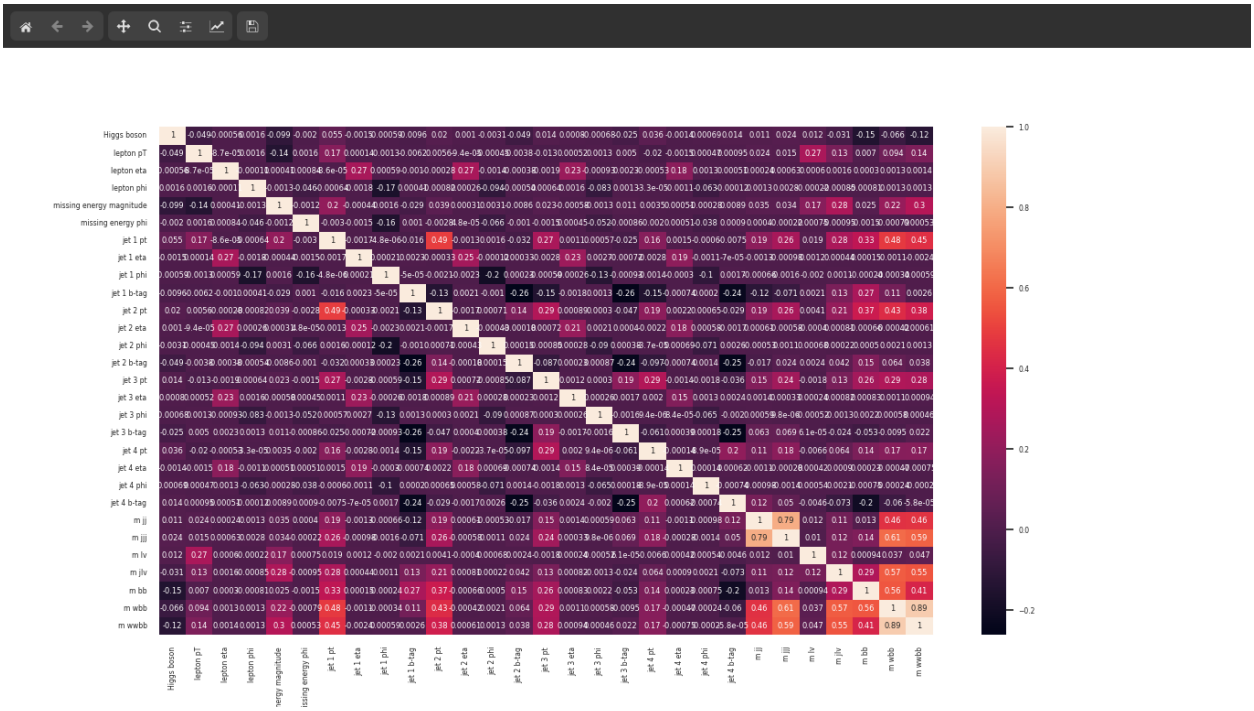
Data visualization:

This was the longest part of the data preprocessing. We came up with the following plots of the data:

1) The data distribution



2) The data correlation



One-hot encoding:

- While visualizing the data, we noticed that the `jet 1-b-tag`, `jet 2-b-tag`, `jet 3-b-tag`, `jet 4-b-tag` features could be one hot encoded into 3 features for each.
- However adding a model layer for categorical encoding turned out to not be practical, and the features would still be encoded as floats which wouldn't help with speeding our training time. So we did not try it.

Normalization:

- We used the Z-score normalization based on the mean and standard deviation of the training datasets. It made the gradient descent converge much faster.
- For the final model submission, we used BatchNormalization() instead of normalizing the features ourselves, it works a bit differently than what we used to do manually because when predicting new values, the mean and standard deviation are updated based on the distribution of the prediction data (which was the testing data in our case).

B) Experiments

Model Benchmarks

Model	Size (in parameters)	Testing Accuracy
Logistic Regression	28	0.6416
Random forests	36	0.7228
XGBoost	100	0.6971
Dense Network	370313	0.7463
Shallow Neural Network	387013	0.7506

Pruning

Since our models were overfitting (with training accuracy 0.81 and testing 0.76) we started looking for ways to simplify the models other than dropouts and regularization, as those were sabotaging how well the model was learning. We tried to prune our model to get rid of insignificant activations and make it sparser. This did help decrease the training accuracy but it also decreased the testing accuracy. We tried applying pruning with constant sparsity of the following values for 4 epochs, and batch size 2048:

Sparsity	**Change in accuracy**
0.4	-0.005
0.6	-0.0047
0.8	-0.0026

Testing different thresholds

Threshold	**Accuracy**
0.4	0.7533
0.45	0.7545
0.5	0.7629
0.55	0.7593
0.6	0.7531

Regularization

- To prevent overfitting we tried using both L1 and L2. We observed that L2 was too harsh and would inhibit the training. On the other hand, L1 seemed to work more effectively and sabotage training less while also preventing overfitting.
- When it came to deciding which layers it was best to regularize initially we had the intuition that penalizing the weights early on during training in the first layers would work more effectively because they would be more prone to overfit since they are close to the data.
- However, with experimentation we saw that regularizing early on prevented the model from learning well so we decided to regularize the last layers more. In addition, dropouts were shown to be very effective against overfitting.

Activation functions

- We tried both ReLu and tanh activation functions for the input layers, and only the sigmoid function for the output layer.
- We tried to use the tanh activation function instead of the ReLu with the architectures of models that were getting a 75.2% accuracy on the testing dataset, and after fine-tuning the hyperparameters, the tanh model wasn't getting past 74.5% so we gave up on using the tanh function.

Batch size:

We kept the training batch size high (1024, 2048 or 4096) because it would make our models much faster. We couldn't afford to try large models with small batch sizes.

MLP ensembles:

- We thought of trying ensembles of neural networks, so we wrote a function called MLP_ensemble() that generates a number of fast_MLPs generated as an argument.
- Surprisingly, an ensemble of fast_MLPs was getting a 73% testing accuracy whereas single fast_MLPs were getting a 75% testing accuracy.
- After a lot of questioning, we came up with the hypothesis that the internal models are learning similar decision boundaries, so they would confirm some wrong deductions that the other models would learn. This idea is expanded on in the next section.

Model diagnosis:

- We wrote functions under the “Evaluate a model” title that are for evaluating two models, which we used multiple times to evaluate the model’s performances.
- The following picture is a recap of how 2 instances of the MLPs returned by the pseudo_final() function perform (‘model1’ has a 0.7542 testing accuracy and ‘test4’ has a 0.7552 testing accuracy)

```
compare(models['model1'], models['test4'], X_test, X_test, y_test)
```

```
model1's confusion matrix
                     Not Higgs    Higgs
Predicted not Higgs    41079.0  14374.0
Predicted Higgs        15122.0  49425.0
------------------------------------------------------------
test4's confusion matrix
                     Not Higgs    Higgs
Predicted not Higgs    41202.0  14379.0
Predicted Higgs        14999.0  49420.0
------------------------------------------------------------
                  Not Higgs test4  Higgs test4
Not Higgs model1            49761         5692
Higgs model1                 5820        58727
------------------------------------------------------------
                   Mistakes model 2  Accuracies model 2
Mistakes model 1              23681                5815
Accuracies model 1             5697               84807
------------------------------------------------------------
```

- As you can see in the previous table, both models do wrong predictions mostly on the same data (23 681 errors are done on the same data points whereas each model messes up on only 5815 and 5697 data points individually)
- We compared MLPs and MLP ensembles multiple times with different splits and architectures, and we always had similar results: the mistakes were common
- This confirms our intuition that the models that we have tested perform badly on the same data points, and we came to the conclusion that having ensembles that consist of more diverse models than simply MLPs could perform better. However, we were limited by our knowledge of neural networks and our computational power to try this solution, so we tried the following approach

Boosted MLP:

- We came up with a solution inspired by XGboost for the fact that all MLP models are performing poorly on the same part of the dataset. And we called it “Boosted MLP”
- The way it works is that the training data is split into 2 parts X_first (80%) and X_second (20%). A first instance of the fast_MLP model is trained on X_first, and a second instance of fast_MLP is trained on the concatenation of X_second and the datasets where X_first was making wrong predictions. We then put both models in an ensemble model where the output is the average of the ensemble output and we train the whole ensemble model for a small number of epochs (20).
- We made the decision to still make a split of 80% and 20% for the dataset of the second model to not be very imbalanced with only mistakes of the first model as its training data.

- When training this Boosted MLP, we were surprised to see that the second model was still having a difficulty to learn, and got a 64% accuracy on the testing data. The accuracy of the whole model (the model computing average of the first and the second one) was 73%, whereas fast_MLP was getting to 75%.
- Our conclusion was that these data points that are frequently getting predicted wrong can hardly be learned by MLPs, and other machine learning models might be more adapted to them.
- Another solution that we thought would have less chances of working well would be to expand on this idea, and having N MLPs instead of only 2, each one would be trained separately and get its own dataset concatenated with the mistakes of the previous MLP. All of these MLPs would then be put in an ensemble where the output would be averaged.
- We also tried variants of this BoostedMLP model:
    - We tried to use other types of MLPs such as tanh_MLP() or deep_MLP()
    - We tried to use different types of models for the first and second model, hoping that the second model would learn differently than the first one
    - We tried changing with the split proportions to decrease or increase the percentage of errors in the second model's training data: we tried 70/30, 80/20 and 100/0
    - All of these variations led us to similar conclusions as the previous one: the second model wasn't learning well, and thus the overall model wasn't performing better than its first model

C) Validation

For validation, we decided to have a simple test-train split and not do cross validation for two reasons:

- This is not academic research, there is no incentive to have a fully transparent and objective measurement of how well our model is performing.
- We randomized the split and all other random variables for every model training (in other words, we had no constant seed), also because our goal here isn't to have transparent information about the performances of our models.

Even though these 2 choices impacted how objectively we could compare the performances of our models, we simply ran the models a few times instead (when we felt the need to). This is not a choice we regret, it could have been more cumbersome to do in terms of organization, and our solution of running the models a few times when we really needed to be sure of things was good enough.

D) Result

The model that performed best with an accuracy score of 76.29 was an MLP with 5 hidden dense layers each with 6000, 4096, 1024, 512, 64 neurons and a single output layer. All the hidden layers used the ReLu activation function, and the output used sigmoid. The input layer was normalized with `mean = 0` and `std = 1`, which helped the network converge significantly

faster. The first hidden layer was regularized using L1 with `lambda = 0.00002`. The last two layers used L1 with `lambda = 0.0008`. Furthermore, dropouts with rate 0.2 were used for the first hidden layer and 0.18 for the rest. We trained our model for 50 epochs using the Adam optimizer with learning rate 0.001, batch size of 3600, binary cross entropy for the loss function, and an 80-20 split to test our model.

E) Conclusion

Technical details

- We first used plain python to plot the data and look at it in different ways, but then quickly moved to using Google Colab because we felt that it was more practical for training, so we ended up training our models on jupyter notebooks.
- We used the `gc` library in Colab which allowed us to link our Google Drives to the jupyter notebooks to read the data directly from our drives or write our models to a shared folder in our drive, this was very practical.
- We first tried GPU and TPU runtimes, and they were less efficient than CPU ones when training models. But they became way more efficient when we increased the batch size. On an MLP with 43777 parameters, increasing the batch size from 32 to 1024, made an epoch take approximately 2 seconds instead of 60.
- We used github properly in the pre-training phase, we had made an organization account, pushed regularly and created a github project (similar to a Jira dashboard) to organize issues. However, we kind of left it out when we started using jupyter notebooks, it wouldn't have been practical to download them and push to github every time we tried a model. We felt like git and github were not as adapted to machine learning as they are to software development.

Difficulties

- We had multiple ideas but looking in the documentation and implementing them took time, especially in the beginning.
- We felt like we didn't have enough knowledge about different neural network architectures and types of layers. We tried to come up with creative model ideas, but it took a lot of energy and the outcomes were not satisfactory.

What we would have done differently

- We think that we spent too much time on the pre-training phase, and representing the data in multiple ways turned out to be of very little help.
- We would have focused more on the code quality. Even though we had multiple functions, we felt that the code structure needed to be refactored towards the end of the project. However, we didn't have enough experience with machine learning and its libraries to know what were efficient ways to organize our code. We would have maybe tried to use OOP instead.