Project 2
The Higgs Machine Learning Challenge

Casuga, Carlisle Aurabelle; Conde Villatoro, Daniel Eduardo; González-Miret Zaragoza, Luis; Langelund, Samuel; Muñoz Méndez, Jesús Eduardo

Abstract

Finding elementary particles requires solving difficult signal-versus-background classification problems, hence machine-learning approaches are often used. In order to improve these classification techniques, the Higgs Machine Learning Challenge invited participants to tackle the task of applying advanced machine learning methods to classify simulated events into either a decay of a Higgs boson or a background event. The dataset consists of 30 predictors and had clear divisions according to the number of hadronic jets detected. Furthermore, before fitting a machine learning model, missing values in the dataset needed to be replaced; however, we found that the value we chose to replace it with didn't have such a significant effect. In total, we studied and tuned four different classifiers. A case of overfitting is presented for some of these models, and parameter values were chosen to avoid that. Lastly, we found that these models gave different accuracy scores when fitted against subsets divided according to the number of jets. Overall, in comparing the four models, we concluded that the Random Forest model was the model that performed the best.

I. Problem Statement and Outline

In this project, we apply different data analysis and machine learning techniques to a binary classification problem: determining whether an entry represented a signal or a background event. Recent advances in the field of deep learning make it possible to learn more complex functions and better discriminate between signal and background classes [1]. This classification problem comes from the Higgs Machine Learning Challenge posted by the collaboration between the ATLAS and different research centers in 2014 [2, 3]. This involves a dataset of simulated events using the official ATLAS full detector and the main task is to classify if an entry is a tau - tau decay of a Higgs boson or just a background signal. These events mimic the statistical properties of the real events of the signal type as well as several important background events.

This paper represents the group's attempt (a very late one) at the challenge. In this work, we first present a look into the dataset and how the group preprocessed it before fitting different models. In the later sections, we present the predictions and scores of different classifiers: Logistic Regression, Decision Tree, Random Forest and Neural Network. For each classifier we present critical parameters and tune them to achieve higher accuracy scores. We calculate the confusion matrix to characterize the true and false positives and negatives of the predictions of our models. In the last sections, we summarize by comparing our models using these confusion matrices and other measures.

II. Preprocessing the data

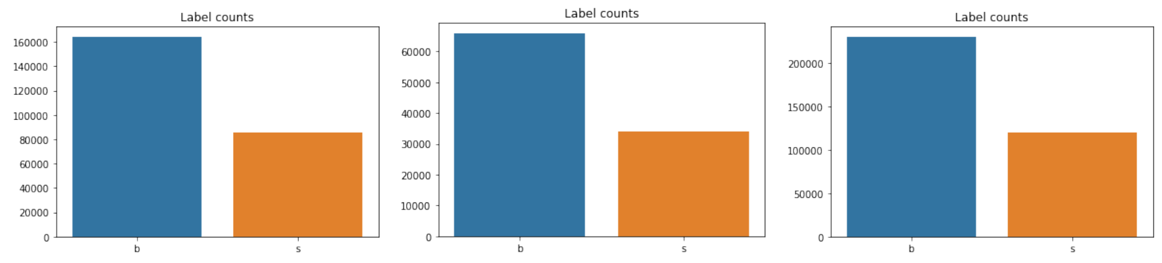
A. Description of the Data: Visualization and Feature selection

A first look at the data reveals that there is a clear structure to the missing values that are set at -999.0. Most of these are related to the properties of hadronic jets in the detector. Jets are these pseudo particles that originate from a high energy quark or gluon and appear in the detector as a collimated energy deposit.

If an event contains no jets at all, all the jet-related features have missing values. If an event contains exactly one jet, the leading jet features are present, but the subleading jet features are absent. For each of these cases, the columns representing these features were dropped. This also gives us information that the model could be evaluated separately on subsets of the data divided according to the number of jets (the `PRI_num_jet` column).

Furthermore, the dataset has already been split into the test and train data, labelled as 't' and 'b' under the `KaggleSet` column, respectively.

The target variable is the `Label` column, indicating which is a Higgs decay, labelled as 's', and which is a background event, labelled as 'b'. The count for each of these label values are presented in Figure 1. It is important to underline that this dataset is enriched with signal events but in real experimental data there will be only approximately two signal events in a thousand events after preselection.



(a) (b) (c)

Figure 1: Background 'b' and Higgs decay signal 's' counts for (a) train, (b) test, and (c) total dataset

Figure 2 shows the histogram of signal and background entries for each feature. The `-phi` features showed uniform densities for both the signal and background events, an indication that these features are not useful in distinguishing the true signals from the background noise and can therefore be dropped from the featrue matrix. The figure also provides us with a complete list of the predictors, a total of 30.

B. *Data Imputation: Dealing with missing values*

Going back to the missing values, the simplest way to deal with these is to drop them; however, by doing this, we risk losing a significant portion of our dataset. Table I shows the amount of missing data entries per jet number. So instead of dropping these values, we could replace them with some estimated value; this process is called data imputation [4]. This value can either be replaced with the mean or the most frequent in the feature column. They can also be replaced with zeros. In this project we use these simpler methods to deal with the -999 values in our dataset. There are, however, more complicated imputation methods that can be used. Examples of which is the K-nearest neighbor imputation that bases the substitution on the nearest neighbors of the missing values.

	-999 entries (%)
Jet 0	34.20
Jet 1	23.66
Jet 2	0.19
Jet 3	0.22
All data	21.07

Table I: Percentage of entries with unphysical values

In this project, for simplicity, we use datasets imputed using Scikit-learn's SimpleImputer() where the missing values can be found and replaced to the mean, or most frequent, or to zeros. After dropping some of the features, since some are not physically relevant, we end up with at least 25 features to train the models.

III. Simple Logistic Regression Analysis

A. *What is Logistic Regression?*

In the interest of simplicity, the following discussion of classifiers will revolve around a binary case. To begin, linear regression, or regression in general, aims to fit a polynomial function in order to predict the response of a variable on unknown data while minimizing the cost function [5]. In logistic regression, the function being fit is the so-called logit function (or the sigmoid function) [6], one that represents the likelihood for a given event, t, to belong to one of two categories. It is given by

$$p(t) = \frac{1}{1+\exp(-t)} = \frac{\exp(t)}{1+\exp(t)}. \tag{1}$$

Furthermore, for linear regression we model the relationship between target and features with a linear equation of a weighted sum between its features; however, with, logistic regression, the logit function in equation 1 transforms these weights into probabilities ranging from 0 to 1.

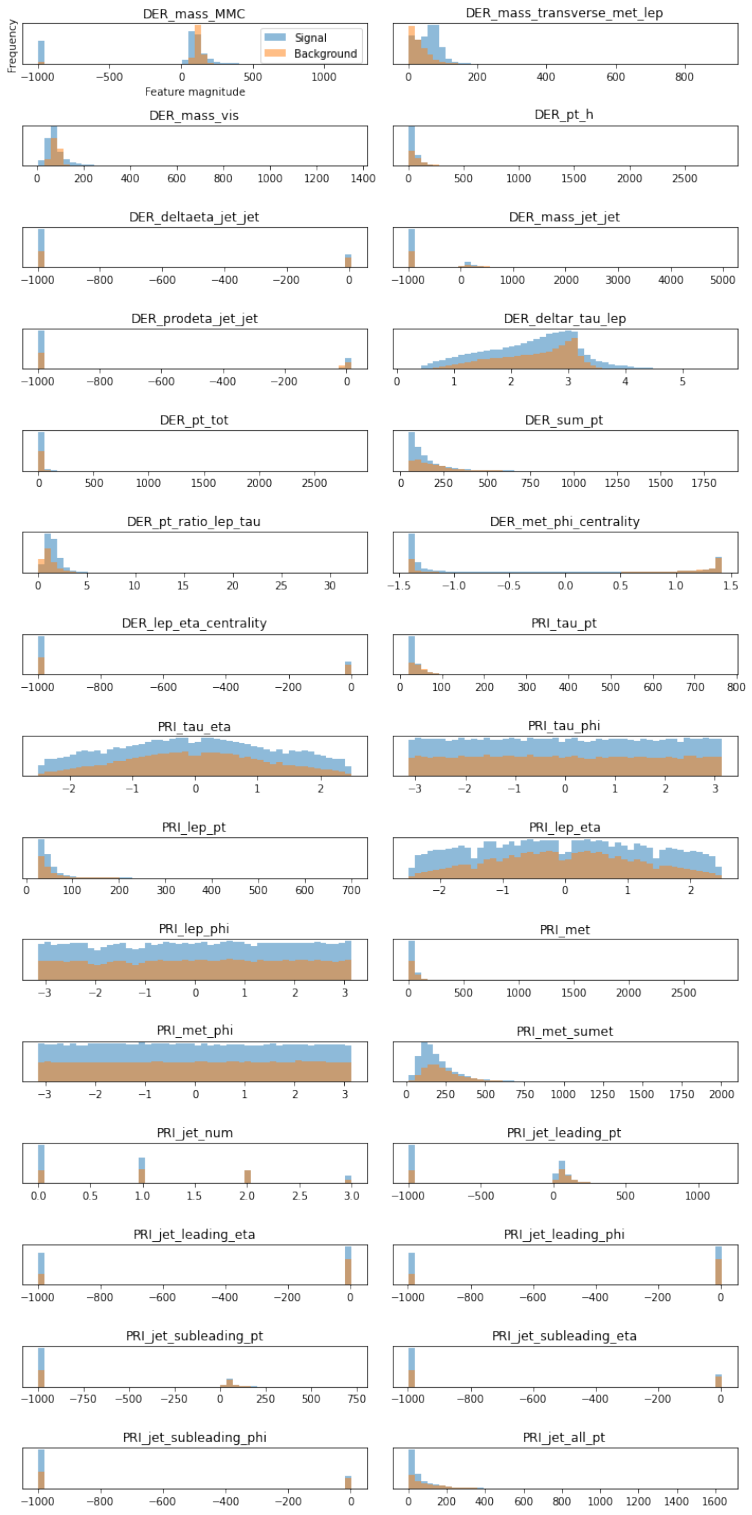


Figure 2: Histograms of the features in the dataset.

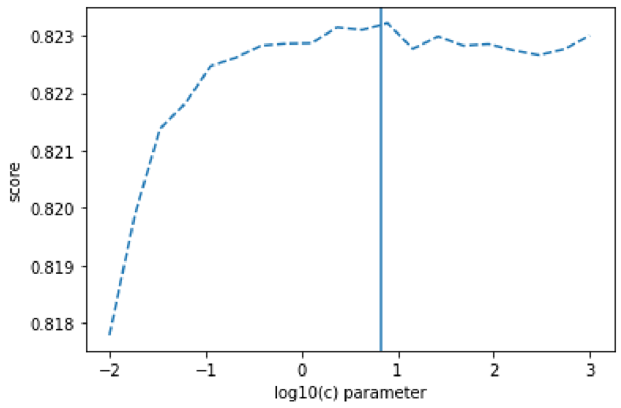


Figure 3: Accuracy score versus C parameter

B. Fitting and Tuning the Logistic Regression

This subsection presents the results from training a simple logistic regression model on the dataset using Scikit - Learn's LogisticRegression() model. Here, we optimize the model, fitting a 2nd order polynomial function to the feature matrix and performing k - fold cross validation. We begin by optimizing the C parameter. This parameter is said to be the regularization strength parameter, and the higher this parameter is, the more the model is allowed to increase its complexity. This parameter is essentially the inverse of the hyperparameter, λ. Figure 3 shows the relationship between. We find a maximum score at a C value around 7.85.

Additionally, the model is trained and fit into different imputed datasets. The results are shown in Table II

Imputation	Score (Train)	Score (Test)
Mean	0.8233	0. 8101
Most Frequent	0.8227	0.8052
Zero	0.8233	0.7851

Table II: Accuracy scores of logistic regression model on datasets whose missing values were replaced by the mean, or the most frequent in the column or to zero

Using the best value for C and the dataset, whose missing values were replaced with the most frequent value in the feature, we now look at how this model behaves with subsets of the data divided according the number of jets. We present the results in Table III. Highest accuracy score we could achieved in the jet = 0 dataset, although there is not much difference compared to the others.

Number of Jets	Score (Train)	Score (Test)
0	0.8261	0.8228
1	0.7122	0.7126
2	0.7401	0.7373
3	0.7305	0.7287

Table III: Accuracy score of logistic regression model on train and test of jet number subset (using most_frequent dataset)

There is an underperformance in the jet = 3 subset, and we later on find out that the accuracy scores given by the logistic regression classifier pale in comparison to other models. Further evaluation of the model can be presented through a confusion matrix that will be presented in Chapter VII.

IV. Decision Tree Classifier Analysis

A. What is a Decision Tree?

Decision tree algorithms can be applied to a classification or regression problem. Advantages of decision trees include its easy construction and interpretability and its ability to exclude unimportant features. It does, however, have a tendency to overfit. The main goal of a decision tree is to find features which contain the most information regarding the target variable. The tree then splits the target features and continues until the sub-datasets are as 'pure'

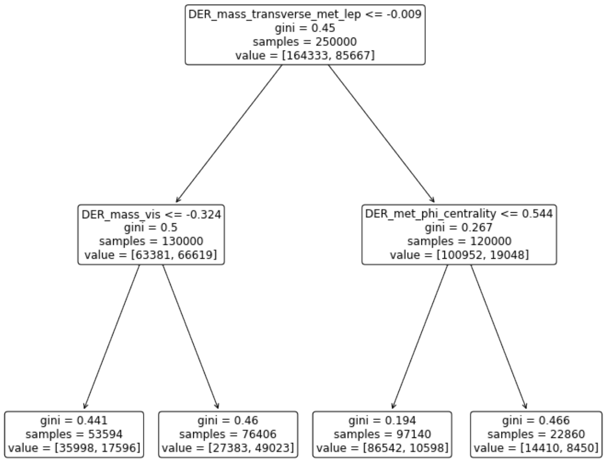


Figure 4: Visualization of a simple decision tree with max_depth = 3 applied to the Higgs dataset

as possible, meaning the values in the target feature in these datasets are of the same class. These datasets at the final splitting are the leaf nodes, or simply leaves, and they contain the model's predicted values [5].

In this paragraph, we underline some important terminology. A leaf, to reiterate, provides the classification of a given instance. A node specifies a test of some attribute of the instance, and a branch corresponds to possible values of this attribute. Overall, an entry is classified beginning with the root node of a tree and there it is tested as specified by the certain node. It moves down to the tree branch corresponding to the value of the attribute. Repeating this process means 'growing' the tree.

The tree model is trained by a recursive splitting of the target feature into branches along the values of its descriptive features that defines the predictor space [5]. How it is split is according to a measure of information gain during the training process. Examples of these measures are the misclassification error, the gini index, and the information entropy. The latter two are what is focused on in this project. The Gini index is

$$\text{Gini} = 1 - \sum_{i=1}^{n} p^2(c_i), \tag{2}$$

and the information entropy is

$$\text{Entropy} = -\sum_{i=1}^{n} p(c_i)\log_2(p(c_i)), \tag{3}$$

where $p(c_i)$ is the fraction of events of class, c_i, in a node. Entropy is a measure of a node's impurity; if all the elements in a dataset belong to the same class, then entropy should be zero. Similarly, the Gini index is maximal when classes are mixed. Gini index's range is [0,0.5] while the entropy's range is [0,1]. By default, a decision tree is grown until the gini index or entropy of a node is zero.

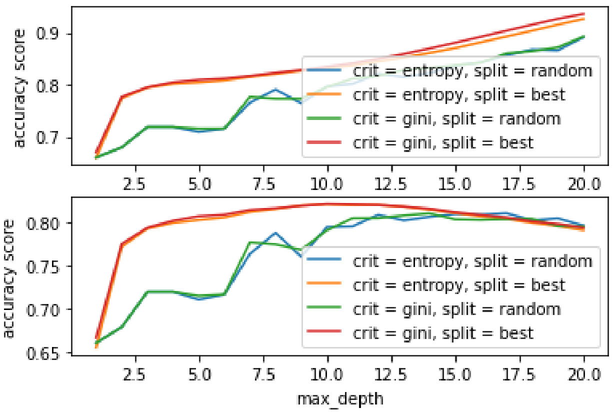
In this section, we look at the DecisionTreeClassifier() from Sci-Kit learn, tune it and show the results in the following section. Finally we apply the BaggedClassifier(), an ensemble method, to our optimized decision tree to see if it improves the results.

B. Fitting and Tuning the Decision Tree

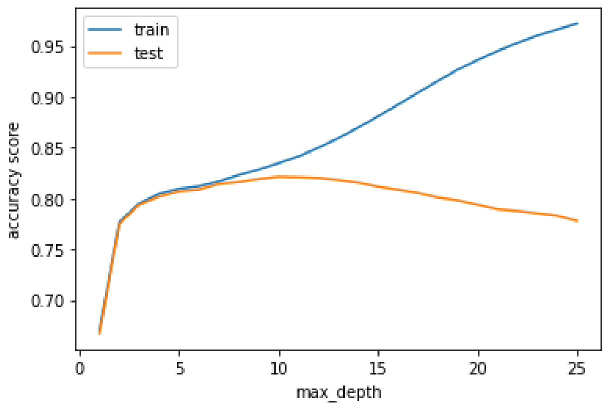
As an introduction, using Scikit - Learn's plot_tree() function, Figure 4 shows a simple decision tree of max_depth = 2, applied to the Higgs dataset. Each box represents a node, that is split into two, and it contains the following information (in the same order as shown inside each box in the figure):

1) the feature where the splitting is based on,
2) the impurity at this node, in this case, the gini index,
3) the number of samples at this node; and,
4) the threshold value that represents the boundary value where the samples are split.

Each box is split into two until a set max_depth; otherwise it will continue to be split until the gini index (or entropy) is zero. So this max_depth parameter value can be set and tuned and is found to be much better than setting it to `max_depth = None` which ultimately leads to overfitting. Additionally, the DecisionTreeClassifier() has the option to change the splitting to 'random', meaning the splitting will not depend on the impurity measure. The accuracy score as a function of max_depth is shown in Figure 5(a) for each combination of criterion (entropy or gini) and splitter type (best or random). As expected, there is no difference between entropy and gini graphs. It is only supposed to matter in calculation time where the information entropy will take a little more because of the log function (see equation 3). It is the 'random' and 'best' splitter graphs that is different from each other, with the 'best' splitter having higher scores, but they both seem to approach each other at higher max_depths. Another information we can get from Figure 5(a) is that although the training data accuracy score increases with increase max_depth, the test data score decreases at a certain point. To further prove this point, the accuracy score vs max_depth of the model on the train and test dataset is shown in Figure 5(b). We observe an overfitting starting at max_depth = 8. From here on out, unless specified, we choose the best options to be criterion = 'gini', splitter = 'best' , and max_depth = 8.



(a)



(b)

Figure 5: (a) Comparison of between criteria and splitter types of accuracy score versus max_depth for test (bottom) and training (top) data with (b) providing a better look at the overfitting of the test data

We also look at fitting a non linear decision boundary between the classes. This means the design matrix is non linear. To set this up we use Scikit - Learn's PolynomialFeatures() function and fit it into the design matrix. It is important to underline that for only this case, we choose only three features for the feature matrix. These are `DER_mass_MMC`, `DER_mass_transverse_met_lep`, `DER_mass_vis` and is chosen because these are, looking at Figure 2, the ones whose signal and background distributions have a significant difference. This way we have the computing power to go up to a complexity of 10 to study whether complexity mattered. From Figure 6, we see that it is not of great importance and in fact gives lower scores than using our original design matrix. From here, we can infer that introducing "high degree non-linearity" to our model is unhelpful and even tends to overfit the data.

Next, we compare how the model works for differently imputed datasets. These datasets with missing data represented as -999 were replaced with either the mean, or the most frequent in the column or simply to zero. The accuracy scores are presented in Table IV. There is no significant difference, however the dataset with zeros is the one giving the higher accuracy scores.

After optimizing parameters in the DecisionTreeClassifier(), we try one last attempt to boost the accuracy score by bagging the decision tree. Bagging is an ensemble method that simply combines the prediction from many decision trees trained from independent samples [5]. It is defined as bootstrap aggregation and is a general purpose procedure for reducing the variance of a classifier. It aims to improve prediction accuracy, with decreased interpretability as a

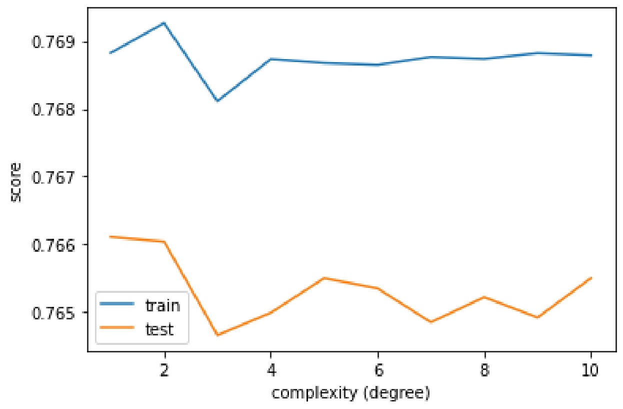


Figure 6: Accuracy score of model the fitted on test and train data as a function of complexity

Imputation	Score (Train)	Score (Test)
Mean	0.8221	0.8162
Most Frequent	0.8248	0.8177
Zero	0.8251	0.8193

Table IV: Accuracy scores of datasets whose missing values were replaced by the mean, or the most frequent in the column or to zero

cost (a collection of bagged trees is more complex than a lone decision tree). Ultimately, it increases our accuracy score, say, for the test data of the dataset imputed with zero, from 0.819 to 0.825. Another ensemble method will be discussedd in the next classifier type.

Number of Jets	Score (Train)	Score (Test)
0	0.8556	0.8454
1	0.8156	0.8012
2	0.8438	0.8284
3	0.8640	0.8228

Table V: Accuracy score of decision tree model on train and test of jet number subset (using most_frequent dataset)

Earlier we alluded to the fact that there is a clear division in the data based on the number of jets. We divide our datasets in accordance to the `PRI_jet_num` with values of either 0, 1, 2, or 3; and, test the decision tree model. The accuracy score values are presented in Table V for each jet number. Each jet number dataset has columns of only missing data and these columns are dropped. Recall that for 2 or 3 jets, there are no additional columns to be dropped because they do not have columns of only the missing data.

The decision tree gave accuracy scores range from 0.75 to 0.83, with only a slight boost with resampling. To help evaluate the model, we present a confusion matrix that summarizes the most important result of this procedure in chapter VII, where we compare this with other models.

V. Random forest analysis

A. What is a Random Forest?

A Random Forest Analysis consists of performing a relatively big number of decision trees analysis and then averaging their results. Usually, a resampling or bootstrap of the data is performed for each of the tries, in order to assure that the trees are the least correlated as possible. The main advantage of Random Forest over decision trees is that they are way less prone to overfitting with effectively no cost on bias, as the bootstraps and averaging assure that the results do not depend on the exact data given.

In this project we use the Random_Forest_model function of Scikit-Learn. As in our previous analysis, the effect of the use of the Gini and Entropy indices will be discussed. However, the most important factor to take into account is the number of trees in each Random forest.

B. Fitting and Tuning the Random Forest

In order to obtain the suitable number of trees and the best performing method, we fit a Random Forest classifier using both the Gini and Entropy indices for a number of trees between 5 and 500 using the data with the -999 substituted with the most frequent value for that entry. Figure 7 shows the accuracy score on the test data for a growing number of trees. As it can be seen in the figure, the quality of both analyses is more or less the same, and remains constant once the number of trees is around 50. For the rest of the analysis we use 200 trees with the Entropy index.

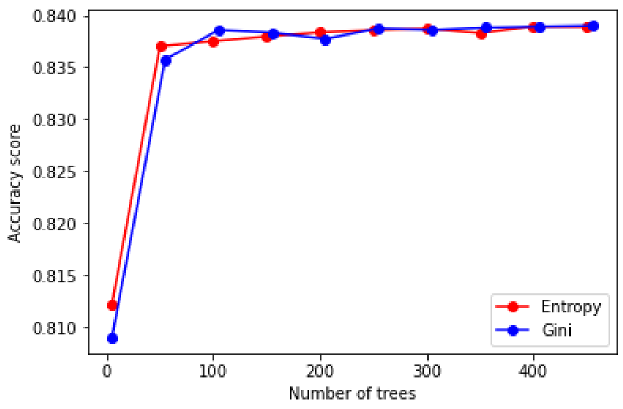


Figure 7: Accuracy of the Random Forest models for different trees number with Gini and Entropy indices

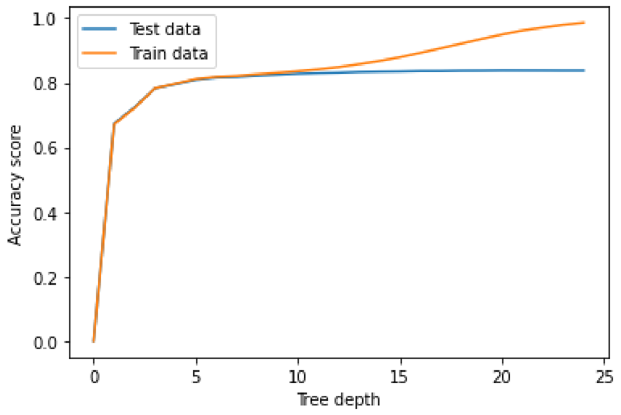


Figure 8: Accuracy of the different trees depth for the zero substitution dataset using Random Forest

Following this, we tested the model for the three different datasets according to which substitution was made for the -999 data, as discussed previously. The results of the accuracy can be seen in Table VI. As we can see, the accuracy is mostly the same for all of the three datasets, which was expected from the results of the previous analysis with decision trees, as a Random Forest is just a combination of many of them. The model was then tested with different depths of each of the trees. The results, shown in Figure 8, show that although the train score increases with the depth, the train score reaches a point of saturation. Setting a maximum depth of 30, the data for each number of jets was analysed independently. These last results can be found in Table VII.

Imputation	Score(Train)	Score (Test)
Mean	0.8359	0.8274
Most Frequent	0.8390	0.8306
Zero	0.83606	0.82750

Table VI: Accuracy scores of the test data of datasets whose missing values were replaced by the mean, or the most frequent in the column or to zero using Random Forest

Number of Jets	Score (Train)	Score (Test)
0	0.8603	0.8462
1	0.8244	0.8053
2	0.8691	0.8447
3	0.8844	0.8415

Table VII: Accuracy score of decision tree model on train and test of jet number subset using Random Forest

The confusion matrix of our final random forest model will be present in Chapter VII to compare with other models.

VI. Neural Networks Analysis

A. *What is a Neural Network?*

Neural networks are built of simple elements called neurons, which take in a value, multiply it by a weight and add a bias, and run it through a non-linear activation function. By constructing multiple layers of neurons, each of which receives part of the input variables, and then passes on its results to the next layers, the network can learn very complex functions [5].

The way we measure the effectiveness of a neural network is by a cost function. Using a technique called gradient descent which deals with small batches of the dataset in order to avoid being stuck in a local minima, one can minimize the cost function by tuning the weights and biases. This is done through backwards propagation where one fine-tunes the weights of a neural net based on the cost function obtained in the previous epoch, i.e. iteration.

Among the advantages of neural networks is their effectiveness for high dimensionality problems and their ability to deal with complex relations between variables. Nonetheless they are theoretically complex, non-intuitive and require expertise to tune. In some cases, they even require a large training set to be effective.

A way to deal with the complex implementation of a neural network is to use a pre-existing package. Keras is a high-level neural networks API, written in Python and makes the creation of a neural network easier. When adding a layer to a model using Keras, a gradient operation will be created in the background and it will take care of computing the backward gradient automatically. Keras also contains initializers which set the initial random weights of the layers.

B. *Fitting and Tuning the Neural Network*

Besides wanting to compare this model with the previous ones (to be discussed in Chapter VII), we wanted to see how dependent a neural network is on the number of entries in the dataset. To achieve this, we decided to use both a dataset where any row that had a unphysical value (-999) was dropped and a dataset where these values were replaced by the most frequent value in each column.

The neural networks used to analyze these datasets consisted of 3 inner layers composed of 1200, 200 and 20 neurons, respectively. For each layer, the ReLU activation function was used. Keras, by default, draws the weights from a uniform distribution and sets the initial values of the biases to zero. We also used a dense layer, which is the basic feed forward fully connected layer option. As the optimizer for the stochastic gradient descent we picked the algorithm "adam". This is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems.

The hyperparameters that need to be tuned in this case are the batch size, which is the number of samples processed before the model is updated, and the epochs that represent the number of times the model goes through the whole dataset. Nonetheless in this case, the hyperparameters were randomly picked.

Rows Dropped			Most Frequent Value		
Number of Jets	Score (Train)	Score (Test)	Number of Jets	Score (Train)	Score (Test)
0	0.8292	0.8117	0	0.8547	0.8489
1	0.8403	0.8063	1	0.8435	0.8158
2	0.9188	0.8299	2	0.9144	0.8323
3	0.9201	0.8082	3	0.9322	0.8194

Table VIII: Accuracy scores for both datasets used in the neural network

From Table VIII we can see that dropping the rows with a -999 value doesn't have a big repercussion in the accuracy of the model, which is also what happened with the decision trees model. We can just remove these entries and don't have a huge repercussion. We can also see that the entry with 3 jets is the one with the highest accuracy since it's one of the subsets with the least number of unphysical values.

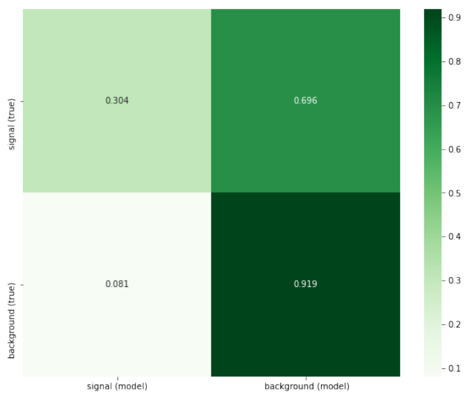
VII. Comparison of Models

Objectively all the other models turned out to have similar accuracy rates, as it can be seen in Table IX, with the exception the under-performing logistic regression. However, there are big differences to note. On one hand, the random forest analysis turned out to be around 1% more accurate for every of the analysis performed. It is interesting to note how the test accuracy decreases for high depth, but it doesn't for Random Forest, as we can see in Figures 5 and 8. This is due to the fact that Random Forest is way less prone to overfit. On the other hand, the accuracy of neural networks is very similar to the one of Random Forest, even when the train accuracy of the later is usually higher than the former.

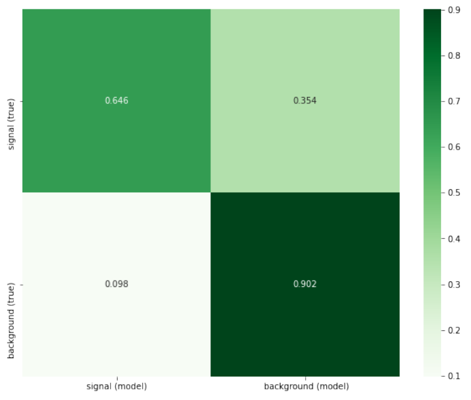
Model	Accuracy (Test)
Logistic Regression	0.7287
Decision Tree	0.8228
Neural Network	0.8194
Random Forest	0.8415

Table IX: Accuracy scores for the subset with jet=3

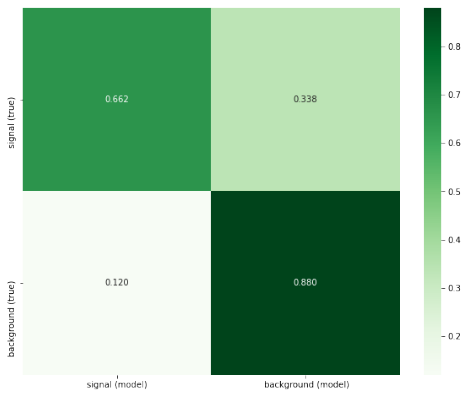
It is important to take into account that the decision trees and random forest tend to give more false negatives, while the neural network gives more false positives than the other two. This can be seen in Figure 9. In real experimental conditions the amount of signal over background would be much smaller. In general, we consider that having a lower amount of false positives is preferable, as we are trying to determine the existence of a particle. This way we are more sure that every positive prediction we get comes from a real Higgs event. For this reason, we believe that Random Forest is the better model for this case, as it is also the one correctly identifying the most Higgs events, with over 68% of them rightly identified.



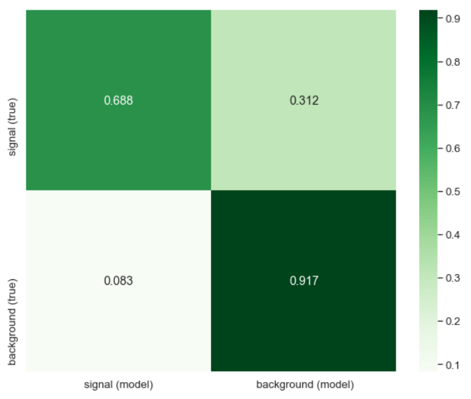
(a) Logistic Regression



(b) Decision Tree



(c) Neural Network



(d) Random Forest

Figure 9: Confusion matrices for the subset with jet = 3

VIII. Conclusions

Overall, a relatively good performance was obtained with most of the models, with Random Forest being the best suited. In order to fully test the capabilities of our models, we should test them with real data where the Higgs decays are less frequent. Additionally a better tuning of the hyperparameters of the neural network should be considered.

The code used for this analysis can be found in the following GitHub repository.

References

[1] P Baldi, P Sadowski, and D Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nature Communications* 5.1 (2014), p. 4308.

[2] *Higgs Boson Machine Learning Challenge*. URL: https://www.kaggle.com/c/higgs-boson.

[3] C Adam-Bourdarios et al. "The Higgs Boson Machine Learning Challenge". In: *Proceedings of the 2014 International Conference on High-Energy Physics and Machine Learning - Volume 42*. JMLR, 2014, pp. 19–55.

[4] W Badr. *6 Different Ways to Compensate for Missing Data (Data Imputation with examples)*. Jan. 2019. URL: https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779.

[5] M Hjorth - Jensen, K Wold, and P Boumbouras Sønderland. *Machine Learning and Data Analysis for Nuclear Physics, European ERASMUS+ Master of Science program*. 2021. URL: https://github.com/CompPhysics/MLErasmus.

[6] C Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2019. URL: https://christophm.github.io/interpretable-ml-book/.