# Employing Machine Learning Techniques to distinguish signal from noise in Higgs measurements

Computational Quantum Physics

Giorgos Michaildis

ARISTOTLE UNIVERSITY OF THESSALONIKI

School of Physics

**June 2025**

For question (a), we will use two classifiers known as: *Decision Tree*, and *K-Nearest Neighbors*. Their implementation in Python is very similar and not a challenging task since libraries like Scikit-learn allow us to access these classifiers without needing too much programming skills.

First, before using the classifiers, we cleaned the data from NaN values and converted possible non-numeric values to numeric. Both of these, are easily done using the Panda module. Then, we separated the data to the columns we tried to predict from and the columns with the values we tried to predict. The last, was the column with the ones and zeros, defined as $y$ and all the other columns where denoted as $x$. The last steps, before we proceeded to the machine learning models, were the definition of the $x$-data to the low and high level quantities, and the split of the dataset to training and testing set. The definition of training and testing datasets was performed with *train test split* from Scikit-learn, were we used 20% of the data as testing and 80% as training set. It should me mentioned here that these steps were performed two times, one for each classifier, since the scripts were developed in different kernels on Jupyter Lab. This way, made the scripts independent.

The first classifier was the Decision Tree which works like a flowchart. Each internal node represents a decision based on a feature, each branch represents the outcome of that decision, and each leaf node represents a final prediction, in our case 1 or 0. We run the algorithm two times, one for the low-level quantities and one for the high-level, using the *DecisionTreeClassifier().fit()* from scikit-learn. This command took the training data, both $x$ and $y$, to create the decision tree and train the model. Then, we could make prediction using the *dtree.predict()* and the $x$ testing data to predict the $y$ outcomes, 0 or 1. Finally, we access the accuracy of the model using the *accuracy score()* and the testing $y$ values along with the predicted $y$ values. The results were:

- **For the low-level quantities:** $\sim 0.54$ or 54%.
- **For the high-level quantities:** $\sim 0.59$ or 59%.

So, for the low-level quantities, 54% of the predicted values, matched the actual label, while for the high-level quantities that percentage reached 59%. The higher accuracy in the high-level quantities can be attributed to the nature of the data: high-level quantities are typically more informative, as they represent processed data, while low-level quantities are raw and often noisy, making it harder for the model to identify meaningful patterns.

For the K-nearest Neighbors, the process was very similar. After the same steps that resulted in the training and testing sets, we used the *KNeighborsClassifier( n-neighbors = 5 ).fit()* along with the $x$ and $y$ training values. The algorithm is named like this because when given a new data point to classify, K-NN looks at the 'K' closest training examples in the feature space. It checks the labels of those K neighbors, 0 or 1, and then the new point is assigned the majority label -the class most common among its K neighbors. The number of these neighbors is defined in the script by *n-neighbors* , where we used a common value found in the documentations. Then, we used the testing $x$ dataset with *knn-high.predict()* to make the predicted data and access the accuracy with the *accuracy-score().* The results were:

- **For the low-level quantities:** $\sim 0.54$ or 54%.
- **For the high-level quantities:** $\sim 0.65$ or 65%.

So, for the low-level quantities, 54% of the predicted values, matched the actual label, while for the high-level quantities that percentage reached 65%. Again, we have an expected difference in the accuracies between the quantities, while the K-nearest Neighbors algorithm performed much better that decision tree in the high-level quantities.

For question (b), we utilized the *TensorFlow* library to design and train a regression neural network model. The initial process was the same: we cleaned the data, defined the column we want to predict and the columns we used to predict from, and separate the data into high and low level quantities. Then, with the same way, we defined the training and testing datasets and proceed to the creation of the model. We used the TensorFlow command *tf.keras.models.Sequential()* to create a linear stack of three layers with the neurons reducing from 64 to 32 to 1, in order to get as output, one prediction, either 0 or 1. In the definition of the model, we took into consideration the case, that the relation between the data is non-linear and gave this information to the model with the parameter: $activation\ =\ relu$. Then, we trained the model using *model.fit()* and making the model go 20 times through the dataset for its training. This is defined by the epoch parameter, along with the *validation-split* parameter that defines the percentage of the training data to be used as a validation test. Finally, we computed the predicted values with *model.predict(x)* $>$ *0.5.* This predicts the probabilities for each event to be signal or noise. If the probability for the event is $> 0.5$ it is considered as signal so the output is 1, if it is $< 0.5$ then it is considered as noise, so the output is 0. The accuracy, was once again, accessed with the *accuracy-score.* The results were:

- **For the low-level quantities:** $\sim 0.51$ or $51\%$.
- **For the high-level quantities:** $\sim 0.66$ or $66\%$.