Machine Learning Engineer Nanodegree

Udacity

Capstone Project Report

Johan Kielbaey

July 21th, 2020

Project Definition

Project Overview

Physics is the study of matter and energy within the boundaries of space and time. Through mathematical models physicists describe the world around us, ranging from subatomic particles to the movement of galaxies. An important aspect of physics is experimental verification of those mathematical models. Every theory must be confirmed by observations (e.g. The bending of light near massive objects as predicted by Albert Einstein's Theory of General Relativity was observed in 1919 during a solar eclipse[1].

In 1964 Peter Higgs, Robert Brout and François Englert theorized the Higgs mechanism and predicted the existence of the Higgs fields and Higgs Boson. In essence, the Higgs Boson is the subatomic particle that provides mass to matter through its interaction with the Higgs field. Unlike the observations that confirm the bending of light, a subatomic particle is so small that it cannot be observed directly. In addition the Higgs Boson only exists for a fraction of time ($1.6 * 10^{-22}$s) before decaying into lighter particles, which makes detecting the Higgs Boson even more difficult[2]. Instead physicists rely on indirect detection by measuring the properties of the particles into which the Higgs Boson decays, which is called the decay signature. These Higgs Bosons are formed by making protons collide near the speed of light inside a particle accelerator (such as the Large Hadron Collider (LHC) at CERN). In 2012 CERN announced a particle matching the properties of the Higgs Boson was found, but further validation was required[3].

Experiments run at LHC produce about 10^{11} collisions every hour. On average in about 300 collisions a Higgs Boson is formed[4]. The sheer amount of data generated during such experiments require the use of reliable and accurate algorithms and models to retain only the data of collisions that is relevant for further analysis.

Problem Statement

Collision events are classified as either Signal or as Background. A Signal means that the collision produced particles of interest for further research (thus potentially a Higgs Boson was formed). Background means that the collision produced other particles not

[1] https://www.britannica.com/science/relativity/Experimental-evidence-for-general-relativity
[2] https://en.wikipedia.org/wiki/Higgs_boson
[3] https://home.cern/science/physics/higgs-boson
[4] https://rdcu.be/b442C

relevant for this research. In order to spend budget most efficiently it is important to be able to classify events with a high level of confidence.

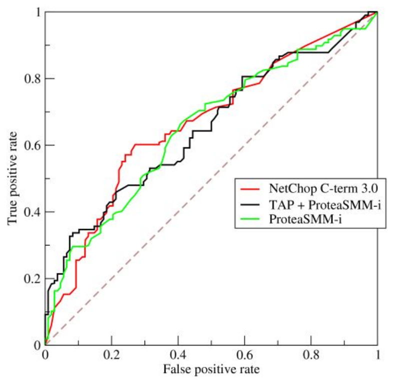
The main problem to solve in this capstone project is to create a well-trained model to classify collision events either as Signal or as Background with a high level of accuracy. The input for the model will be the measurements of these collision events.

Metrics

As the amount of interesting events is very small compared to the total number of events, it is important not to misclassify actual Signals (false negatives) as this would mean missed opportunities to learn more about the Higgs Boson. At the same time the model should not result in too many Backgrounds classified as Signals (false positive) as this would result in research effort wasted on irrelevant events.

Recall is the metric showing us how many of the Signal events have actually been classified correctly. A low recall means that many Signal events were classified as Background, which is definitely not good. Precision is a different metric and it shows how many of the events classified as Signal are in fact Signal. If this metric is low, then many Background events have been classified as Signals. This leads to more events that will be investigated further. Ideally, the model that has a high recall and high precision. However often optimizing for 1 metric means sacrificing on the other metric.

The metric used in this project to evaluate the performance of a model is the *Area under the ROC Curve* (AUC) metric. This metric provides a balance between both recall and precision. The Receiver Operating Characteristic (ROC) curve (see image[5] on the right) shows the relation between recall and precision. A higher value means that the model is better at classifying.



[5] Image used from https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Analysis

Data set description

The dataset used for the capstone project is the HIGGS data set, available at the UCI Machine Learning Repository[6]. It contains 11,000,000 items with 28 features. Each item contains the measurements of collision events generated using Monte Carlo methods. These events can either be Signal or Background.

The 28 features consist of 21 low-level and 7 high-level kinematic features of these collision events. All of these are numerical features. The low-level (or raw) features are measured by the particle detectors in LHC. The high-level (or derived) features are functions of these raw features that are manually derived by physicists in order to increase accuracy of the models used at the time of creation of the data set. Crafting these high-level features requires a great deal of domain knowledge, expertise and is both labor and time intensive.

Below is a list of all features.

- Low level features (21): lepton_pt, lepton_eta, lepton_phi, missing_energy_phi, missing_energy_magnitude, jet_1_pt, jet_1_eta, jet_1_phi, jet_1_b-tag, jet_2_pt, jet_2_eta, jet_2_phi, jet_2_b-tag, jet_3_pt, jet_3_eta, jet_3_phi, jet_3_b-tag, jet_4_pt, jet_4_eta, jet_4_phi, jet_4_b-tag
- *High level features (7)*: m_jj, m_jjj, m_lv, m_jlv, m_bb, m_wbb, m_wwbb

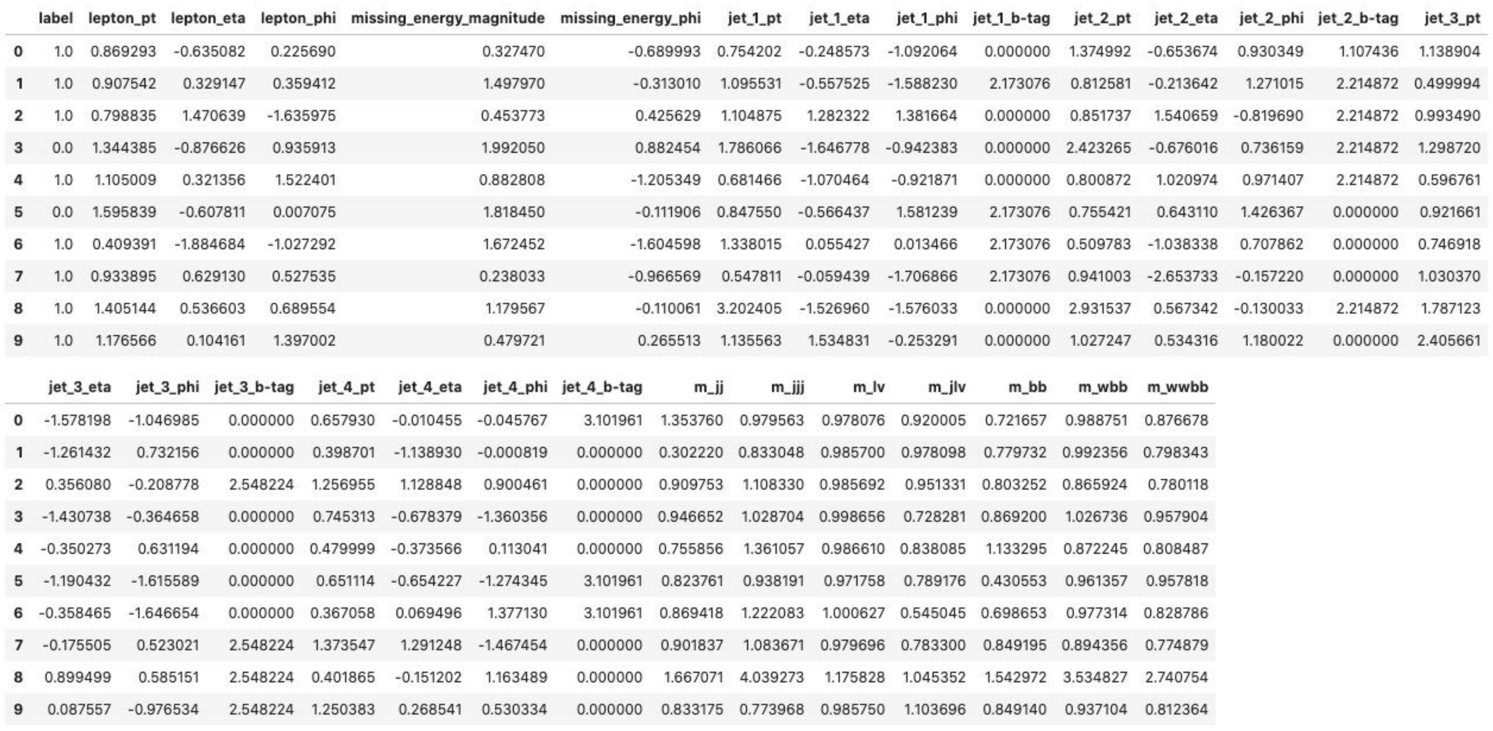
Exploration & Visualisation

This section presents data exploration and visualisation. The purpose of both parts is to get a good understanding of the distribution data and any potential issues that need to be fixed during data preprocessing.

The original data set consists of measurements of 11,000,000 events, which is substantial. In this capstone project I have chosen to use different parts of this set in order to speed up the workflow. During model development, I decided to use a set of the first 50,000 events to more quickly iterate over different possible model architectures. For hyperparameter tuning I used the first 250,000 events. And finally I used 1,000,000 events to retrain the tuned model.

[6] http://archive.ics.uci.edu/ml/datasets/HIGGS

The table below shows the values of all features for the first 10 events.



	label	lepton_pt	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	jet_1_pt	jet_1_eta	jet_1_phi	jet_1_b-tag	jet_2_pt	jet_2_eta	jet_2_phi	jet_2_b-tag	jet_3_pt
0	1.0	0.869293	-0.635082	0.225690	0.327470	-0.689993	0.754202	-0.248573	-1.092064	0.000000	1.374992	-0.653674	0.930349	1.107436	1.138904
1	1.0	0.907542	0.329147	0.359412	1.497970	-0.313010	1.095531	-0.557525	-1.588230	2.173076	0.812581	-0.213642	1.271015	2.214872	0.499994
2	1.0	0.798835	1.470639	-1.635975	0.453773	0.425629	1.104875	1.282322	1.381664	0.000000	0.851737	1.540659	-0.819690	2.214872	0.993490
3	0.0	1.344385	-0.876626	0.935913	1.992050	0.882454	1.786066	-1.646778	-0.942383	0.000000	2.423265	-0.676016	0.736159	2.214872	1.298720
4	1.0	1.105009	0.321356	1.522401	0.882808	-1.205349	0.681466	-1.070464	-0.921871	0.000000	0.800872	1.020974	0.971407	2.214872	0.596761
5	0.0	1.595839	-0.607811	0.007075	1.818450	-0.111906	0.847550	-0.566437	1.581239	2.173076	0.755421	0.643110	1.426367	0.000000	0.921661
6	1.0	0.409391	-1.884684	-1.027292	1.672452	-1.604598	1.338015	0.055427	0.013466	2.173076	0.509783	-1.038338	0.707862	0.000000	0.746918
7	1.0	0.933895	0.629130	0.527535	0.238033	-0.966569	0.547811	-0.059439	-1.706866	2.173076	0.941003	-2.653733	-0.157220	0.000000	1.030370
8	1.0	1.405144	0.536603	0.689554	1.179567	-0.110061	3.202405	-1.526960	-1.576033	0.000000	2.931537	0.567342	-0.130033	2.214872	1.787123
9	1.0	1.176566	0.104161	1.397002	0.479721	0.265513	1.135563	1.534831	-0.253291	0.000000	1.027247	0.534316	1.180022	0.000000	2.405661

	jet_3_eta	jet_3_phi	jet_3_b-tag	jet_4_pt	jet_4_eta	jet_4_phi	jet_4_b-tag	m_jj	m_jjj	m_lv	m_jlv	m_bb	m_wbb	m_wwbb
0	-1.578198	-1.046985	0.000000	0.657930	-0.010455	-0.045767	3.101961	1.353760	0.979563	0.978076	0.920005	0.721657	0.988751	0.876678
1	-1.261432	0.732156	0.000000	0.398701	-1.138930	-0.000819	0.000000	0.302220	0.833048	0.985700	0.978098	0.779732	0.992356	0.798343
2	0.356080	-0.208778	2.548224	1.256955	1.128848	0.900461	0.000000	0.909753	1.108330	0.985692	0.951331	0.803252	0.865924	0.780118
3	-1.430738	-0.364658	0.000000	0.745313	-0.678379	-1.360356	0.000000	0.946652	1.028704	0.998656	0.728281	0.869200	1.026736	0.957904
4	-0.350273	0.631194	0.000000	0.479999	-0.373566	0.113041	0.000000	0.755856	1.361057	0.986610	0.838085	1.133295	0.872245	0.808487
5	-1.190432	-1.615589	0.000000	0.651114	-0.654227	-1.274345	3.101961	0.823761	0.938191	0.971758	0.789176	0.430553	0.961357	0.957818
6	-0.358465	-1.646654	0.000000	0.367058	0.069496	1.377130	3.101961	0.869418	1.222083	1.000627	0.545045	0.698653	0.977314	0.828786
7	-0.175505	0.523021	2.548224	1.373547	1.291248	-1.467454	0.000000	0.901837	1.083671	0.979696	0.783300	0.849195	0.894356	0.774879
8	0.899499	0.585151	2.548224	0.401865	-0.151202	1.163489	0.000000	1.667071	4.039273	1.175828	1.045352	1.542972	3.534827	2.740754
9	0.087557	-0.976534	2.548224	1.250383	0.268541	0.530334	0.000000	0.833175	0.773968	0.985750	1.103696	0.849140	0.937104	0.812364

The first column contains the label and is binary. The values of the features in this sample are quite small. Most are either roughly between -2 and +2 or between 0 and 5. The values for the features jet_1_b-tag, jet_2_b-tag, jet_3_b-tag and jet_4_b-tag appear to have discrete values (e.g. jet_1_b-tag is either 0.0 or 2.173076). There don't seem to be any obvious patterns in the features that would distinguish between Signal or Background.

As part of the data exploration and visualization following operations were performed:

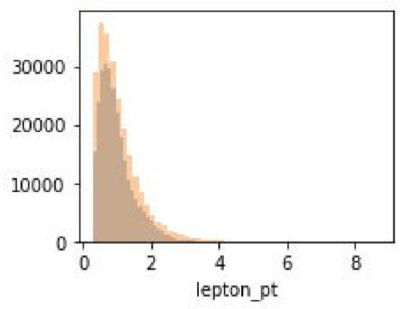
- For further analysis and training it is important to know if the data set is balanced. An imbalanced data set can introduce a bias in the model. Fortunately this is not the case. The data set is *nearly balanced.*

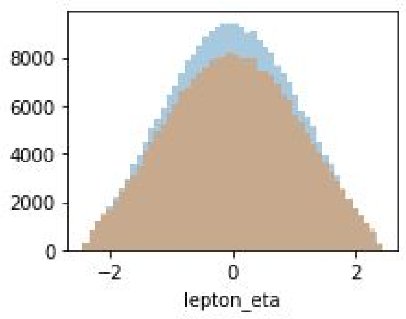
Label	**Data set size**		
	50,000	**250,000**	**1,000,000**
Background	23,435 (46.87 %)	118,058 (47.22%)	470,327 (47.03 %)
Signal	26,565 (53.13 %)	131,942 (52.78%)	529,673 (52.97 %)

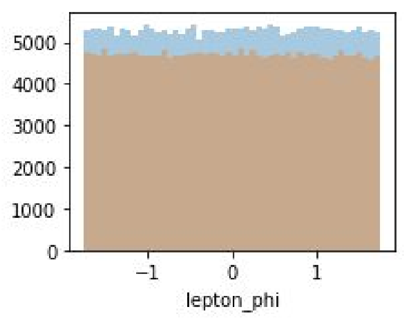
- The data set must be checked for missing data. This can easily be done using the pandas DataFrame function `isnull()`. Fortunately this data set does not have missing values.

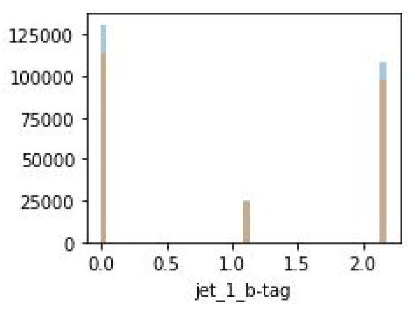
- Visualizations and histograms are a great way to check the distribution of the values of the different features. When comparing the distributions of Signal vs Background in a histogram per feature it immediately becomes clear there is very little difference between Signal and Background events. A few repeating patterns in the histograms also emerge. Following groups of features emerge:
    - Features lepton_pt, jet_1_pt, jet_2_pt, jet_3_pt, jet_4_pt and missing_energy_magnitude have a poisson distribution.
    - Features lepton_eta, jet_1_eta, jet_2_eta, jet_3_eta and jet_4_eta have a normal distribution.
    - Features lepton_phi, jet_1_phi, jet_2_phi, jet_3_phi, jet_4_phi and missing_energy_phi have a uniform distribution.
    - Features jet_1_b-tag, jet_2_b-tag, jet_2_b-tag and jet_4_b-tag have discrete values.

Below are visualisations illustrating each of these groups (blue: Signal; orange: Background).

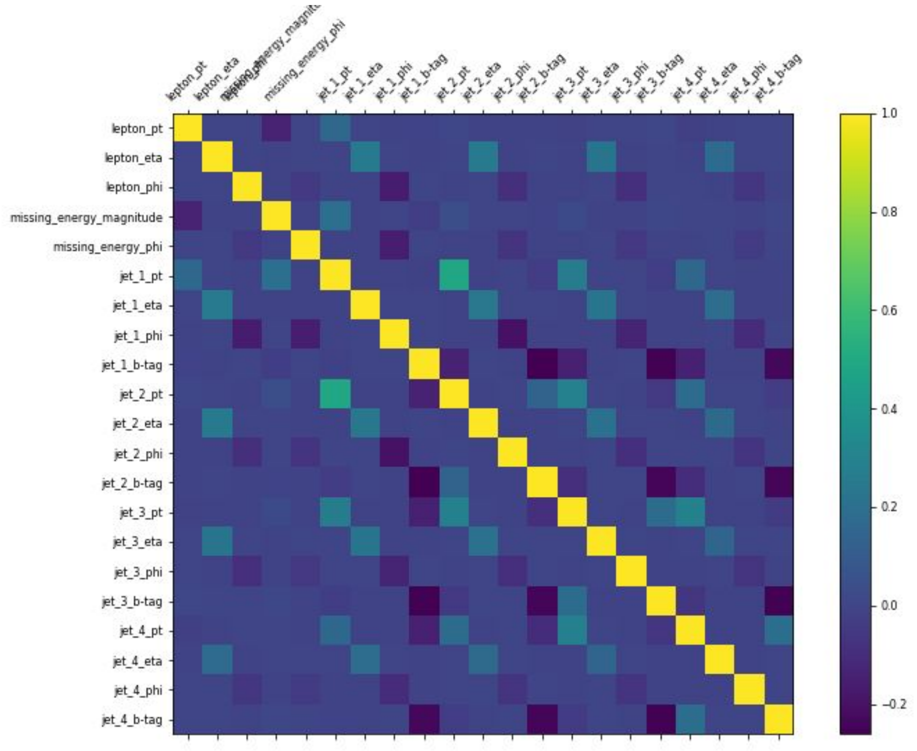








- Correlation between features. The correlation map below shows very little correlation between the different features. The single exception are features jet_1_pt and jet_2_pt, which have a correlation of about 0.48.



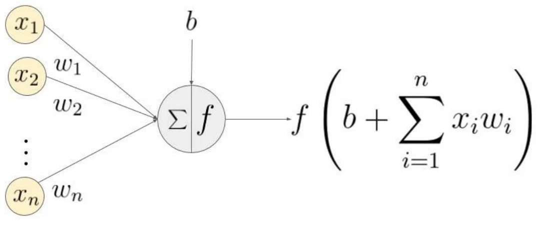
Algorithms and Techniques

I chose 2 different algorithms to train the classifier model: XGBoost and Deep neural network.

XGBoost is an ensemble machine learning method based on decision trees. Its primary goals are speed, scalability and model performance. In general it performs very well in various machine learning competitions and is easy to use. XGBoost is available in Amazon SageMaker as a built-in algorithm, which makes it very easy to use. The intention of this experiment was to create a baseline model with minimal effort and see how well it performs.

A neural network (NN) is an artificial representation of how the brain works. It consists of neurons (nodes) organized into layers. A NN can have few or many layers.

Shallow NNs typically have 1-2 layers, whereas Deep NNs (DNNs) can have tens or hundreds of layers. Each neuron has a set of inputs (with weights associated to it), an activation function and 1 output. The neuron will calculate the output using the input values, input weights, optional bias and the activation function as defined in the image below[7].



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

The inputs in the first layer are the values of the features of the item on which the network should make a prediction. The output of the neurons of a layer are fed as input to the neurons of the next layer. The output of the final layer is the prediction that the network makes.

During training, items are fed into the network and the output is compared to expected output. The difference between the expected and actual output is used to determine the loss. A high loss means that the network is performing poorly (making many wrong predictions). The optimizer uses the loss function and the current parameters of the network (weights & biases) to determine a new set of parameters, which are updated via a mechanism known as back propagation.

One of the potential issues with neural networks is overfitting, which means that the neural network becomes so familiar with the training set that it can perfectly “predict” for events in the training set, but performs poorly for other items. The techniques to reduce or prevent overfitting are called Dropout (at random ignore links between neurons) and BatchNormalization (normalize the inputs of each neuron).

Despite their complexity DNNs are very powerful in detecting patterns in data sets and can achieve high accuracy.

[7] Image used from https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/

Benchmark

The research paper referred to in the data set compares a couple of different models. As mentioned in the project proposal, the results in this paper would be the benchmark to compare my model with. The values in the table below represent the AUC score for different models under 3 different scenarios (only low-level features, only high-level features and all features).

Model	**Low-level features**	**High-level features**	**All features**
Boosted Decision Trees	0.73	0.78	0.81
Shallow NN	0.73	0.78	0.82
Deep NN	0.88	0.80	0.88

Methodology

Data Preprocessing

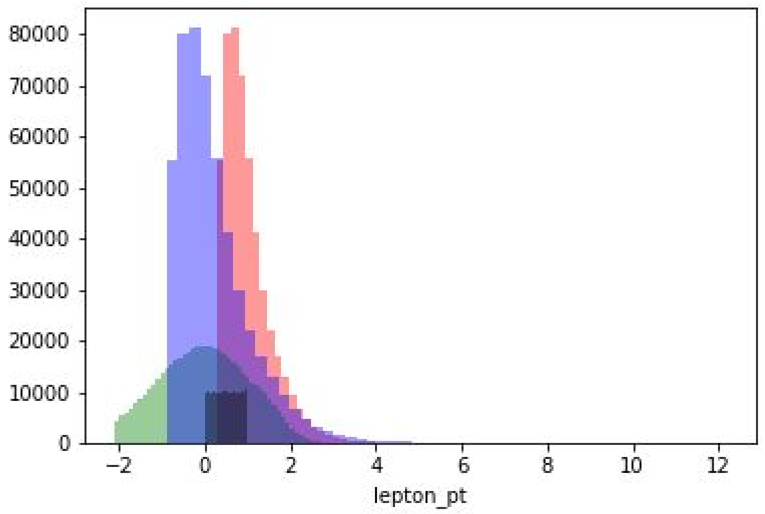
Data exploration showed that the data set is complete. There are no missing values or outliers that would require e.g. imputation. The data set is also fairly well balanced. Following operations will have to be performed on the data set.

Feature engineering

As features lepton_phi, missing_energy_phi, jet_3_phi and jet_4_phi have a uniform distribution they will be dropped. Initial tests with and without these features showed no difference.

Feature scaling

The features lepton_pt, missing_energy_magnitude, jet_1_pt, jet_2_pt, jet_3_pt and jet_4_pt are skewed. This can have a negative impact on the performance of the model. The sklearn preprocessing package contains a number of non-linear scalers that can transform the values on these features so they adhere better with a normal distribution: RobustScaler, PowerTransformer and QuantileTransformer.



The histogram above shows the distribution of the feature lepton_pt after applying each of the different non-linear scalers (red: original values; blue: RobustScaler; green: PowerTransformer; black: QuantileTransformer). When comparing the outcome of these scalers, PowerTransformer provides the best approximation of a normal distribution and as such this scaler will be used to transform the skewed features.

A StandardScaler will be used on all features to normalize the values across the different features.

Splitting the data

The final preprocessing step is splitting the data set into 3 parts:

- Training set (70%)
- Validation set (15%)
- Test set (15%)

The values between the round bracket indicate the percentage of each sub set.

Implementation

For both approaches I created separate Jupyter notebooks: Model_XGBoost and Model_NN. Both notebooks start by importing the necessary python packages, setting the environment and uploading the data set from the notebook storage onto Amazon S3. Both the train and validation set contain the features and labels (in 1st column), whereas the test set only contains the features.

XGBoost

SageMaker offers a couple of different XGBoost versions. I choose to use the most recent version (1.0-1) in algorithm mode[8] during training. As objective I used 'binary:logistic' with 'auc' as the evaluation metric.

As the data set is intuitively more complex and differences between Signal and Background appear to be subtle, I chose to train longer (with an early stopping if validation doesn't improve) while setting the hyperparameters to be more conservative in every round for the initial classifier. The details and description of each hyperparameter is well documented in Amazon's documentation[9]. The table below specifies the different hyperparameters, the value I set during the initial training, the ranges I set for hyperparameter tuning and the value of the tuned model.

Hyperparameter	Initial value	Tuning range	Tuned value
booster	gbtree (default)		
number of rounds	500		
early stopping rounds	15		
max depth	10	1 - 20 (integer)	16
alpha	0 (default)	0 - 2 (continuous)	1.7561
eta	0.2	0 - 1 (continuous)	0.1036
gamma	4	0 - 10 (continuous)	9.8276
min child weight	6	1 - 10 (continuous)	1.5400
subsample	0.8	0 - 1 (continuous)	0.9476

Compared to the initial model, there are a few differences to notice. The tuned model is substantially more complex (max_depth 10 vs 16) and most of the tuned hyperparameters (alpha, eta, gamma, subsample) make the learning process even more conservative than the initial model. The only exception is the min child weight, which decreased from 6 to 1.54.

[8] See https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html#xgboost-supported-versions
[9] See https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html

Deep neural network

Similar to XGBoost I also used Amazon SageMaker to develop and train the DNN. As the framework for the DNN I decided to use PyTorch, which is a relatively young but flexible and powerful machine learning library. The python SDK for SageMaker provides support for PyTorch via the class sagemaker.pytorch.estimator.PyTorch.

When using the PyTorch framework in SageMaker a training script must be provided that takes care of the actual training. It must accept a number of input parameters (), create the model, load the data sets, run the training and finally save the model. I based my script on the code provided in the plagiarism detection project in which PyTorch was used. I've included 3 files containing the necessary code:

- Model.py defines the structure of the DNN and to feedforward data through the network. I used the torch.nn.Sequential container to define all layers, which simplifies the code for the feedforward function.
- Train.py is executed during model training. It starts by parsing the command line arguments (such as location of the training and validation sets, location where the trained model should be stored and values for hyperparameters), determines if the container running the training has GPU capabilities in order to dramatically increase performance, loads the training and validation sets, creates the model, the learning optimizer and loss function, runs the training and finally saves the trained model.
- Predict.py is executed once the model is deployed. It's discussed in a later section of this report.

During each epoch of the training process, the script will load the training data in batches and make the model make predictions. For each batch the outputs (predictions) are compared with the actual labels. The average of the losses is calculated. Next the model is put in "evaluation" mode and predictions for the validation set are made. The average loss on the validation set is compared to the best loss on the validation set. If improved (i.e. less) the model in that epoch is considered better and saved. If the model didn't improve for 10 epochs or the maximum number of epochs has been reached, training stops.

A DNN can have many different parameters. When using a large data set, it can be very time consuming (and costly) to try out many combinations on the entire data set (even with GPUs). Instead I found that starting with a simple network and a small subset of the data allowed me to test different values for each of the network parameters (number of neurons per hidden layer, activation function, optimizer, weight initialization) in order to get an idea which combinations of values for parameters worked better. Initially I started with a shallow network without dropout and batch normalization and experimented with different activation functions (ReLU, PReLU and Tanh), different optimizers (SGD and Adam) and different methods to initialize the weights of the neural network. This showed me that a combination of Tanh, Adam and a uniform weight initialization function worked best.

As the network started overfitting, I added batch normalization and dropout which reduced the performance of the network. This can be expected as dropout causes the network to ignore (drop) outputs from some neurons. As I increased the data set, I added more layers to allow the network to extract more information from the data. This iterative approach allowed me to create a more complex network architecture. To speed up this iterative process, I changed the instance type of my SageMaker Notebook Instance to have GPUs and use SageMaker local mode to be able to train locally[10] in my notebook to further refine the model architecture.

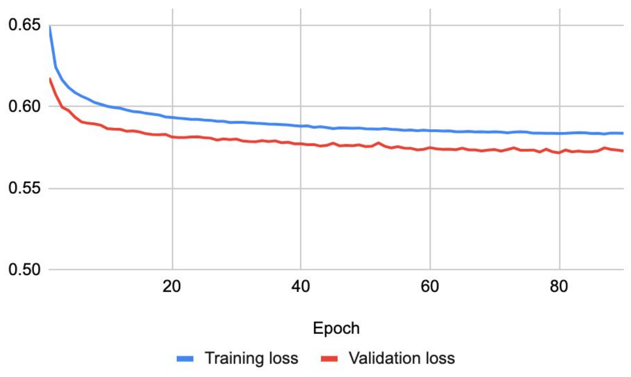
Once satisfied with the model architecture, I moved on to tuning hyperparameters, such as batch size, dropout rate, learning rate and size of the hidden layers. The approach using SageMaker local model no longer worked as SageMaker Hyperparameter tuning is not supported in SageMaker local mode. However the advantage of Hyperparameter tuning in SageMaker is that it can train multiple models in parallel and as such test more combinations of the hyperparameters in a shorter timeframe. SageMaker uses a Bayesian search approach to optimize the hyperparameters. Upon deciding which values to use for the hyperparameters it uses all knowledge obtained from previous training jobs (part of the Hyperparameter Tuning job). From that perspective it’s best to use moderate parallelism in relation to the total number of training jobs. In my Hyperparameter Tuning job I allowed SageMaker to run 4 jobs in parallel with a total of 30 jobs.

[10] See https://aws.amazon.com/blogs/machine-learning/use-the-amazon-sagemaker-local-mode-to-train-on-your-notebook-instance/

The table below shows the different values:

Hyperparameter	**Initial value**	**Tuning range**	**Tuned value**
Batch size	300	300, 512, 768, 1024	300
Dropout rate	0.5	0.3, 0.4, 0.5, 0.6, 0.7	0.3
Learning rate	0.001	0.0001, 0.0003, 0.0005, 0.001	0.001
Hidden neurons	300	510, 595, 680, 765, 850	680

After hyperparameter tuning, the model was trained a final time using a larger data set of 1 million collision events. The graph below shows evolution of the training and validation loss during training.

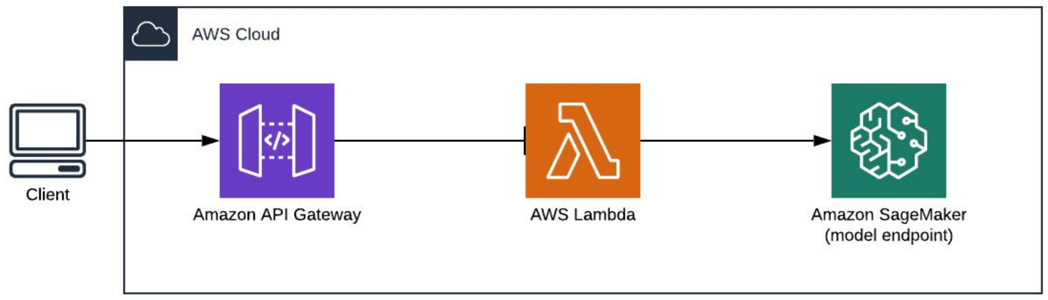


Prediction system

Architecture

The goal of my capstone project was twofold: a trained model and a REST Api using this model to make predictions of collision events. The model was trained using Amazon SageMaker. After training and testing it was deployed as an endpoint in SageMaker.

The REST Api consisted of an Amazon API Gateway RestApi combined with an Amazon Lambda Function, which communicated with the SageMaker endpoint for prediction as illustrated in the image below.



The Rest API expects that the measurements of collision events were passed in csv format. The Lambda function in itself will simply pass the data to the endpoint. The prediction code of the endpoint will apply feature scaling (using the scalers that were used to perform feature scaling on the initial data set) and make the actual prediction. The endpoint will return either 0 (for Background) or 1 (for Signal). Both Lambda and API Gateway will pass this result back to the client as csv.

Implementation

The RestApi in Amazon API Gateway was configured to integrate with Amazon Lambda in proxy mode. API Gateway will pack the HTTP request in a specific event format[11] and pass this to Amazon Lambda. It expects to receive the result from Lambda wrapped in a specific dictionary, containing the HTTP status code and body for the response.

Due to limitations in Amazon Lambda, I've deliberately kept the Lambda code simple. Initially I intended to perform feature scaling in Lambda, however this requires including python packages numpy and pandas in the Lambda function. The size of a deployment package (containing the code running in the Lambda Function) is limited. Including numpy and pandas made the package too big and would have required the use of Lambda Layers. As this complicates deployment, I decided to move this processing to the SageMaker endpoint. This reduced the Lambda Function to a proxy layer. In hindsight this choice was conceptually better as the parameters of the scalers and the model are stored together in a single deployable artifact. It also allowed for the model to be retrained using a different data set without having to update the code of the Lambda Function. The Lambda code makes use of the sagemaker-runtime client in the boto3 package to invoke the SageMaker endpoint.

[11] See https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html

The final piece is the SageMaker endpoint, which is deployed from the notebook using the Model.deploy() call. The endpoint uses a separate script (predict.py) containing the code to make the actual prediction. The endpoint requires 4 functions to be created:

- Model_fn creates the neural network, loads the state dict (to set weights and biases in the network) and loads the scalers.
- Input_fn deserializes the data received by the endpoint (from the Lambda function via the invoke_endpoint() call). It converts the input data from csv to a pandas DataFrame, applies scaling and returns a NumPy array.
- Predict_fn makes the actual prediction. As arguments it receives the NumPy array (from the input_fn) and the model. It will invoke the model to make the prediction and return this as a NumPy array (rounded to integers).
- Output_fn serializes the prediction(s) to a format expected by the Lambda function (csv in this case).

As an example, let's invoke the Rest API with the data of the first 5 items in the data set.

```
$ curl -X POST https://xxxx.execute-api.eu-west-1.amazonaws.com/live/test -d
'0.869293212890625,-0.6350818276405334,0.327470064163208,0.7542022466659546,-0.248573139309883
15,-1.0920639038085935,0.0,1.3749921321868899,-0.6536741852760315,0.9303491115570068,1.1074360
609054563,1.138904333114624,-1.5781983137130735,0.0,0.657929539680481,-0.010454569943249224,3.
1019613742828365'
1
$ curl -X POST https://xxxx.execute-api.eu-west-1.amazonaws.com/live/test -d
'0.9075421094894408,0.3291472792625427,1.4979698657989502,1.09553062915802,-0.5575249195098877
,-1.5882297754287718,2.1730761528015137,0.8125811815261839,-0.2136419266462326,1.2710145711898
804,2.2148721218109126,0.4999939501285552,-1.2614318132400513,0.0,0.39870089292526245,-1.13893
00823211668,0.0'
1
$ curl -X POST https://xxxx.execute-api.eu-west-1.amazonaws.com/live/test -d
'0.7988347411155698,1.4706387519836421,0.45377317070961,1.104874610900879,1.282322287559509,1.
3816642761230467,0.0,0.8517372012138367,1.540658950805664,-0.8196895122528077,2.21487212181091
26,0.9934899210929872,0.3560801148414612,2.548224449157715,1.256954550743103,1.128847599029541
,0.0'
0
$ curl -X POST https://ftwlqev1nj.execute-api.eu-west-1.amazonaws.com/live/test -d
'1.344384789466858,-0.8766260147094725,1.992050051689148,1.786065936088562,-1.6467777490615845
,-0.9423825144767758,0.0,2.4232647418975835,-0.6760157942771912,0.7361586689949036,2.214872121
8109126,1.2987197637557986,-1.430738091468811,0.0,0.7453126907348634,-0.6783788204193115,0.0'
1
 curl -X POST https://xxxx.execute-api.eu-west-1.amazonaws.com/live/test -d
'1.1050089597702026,0.3213555514812469,0.8828076124191283,0.6814661026000977,-1.07046389579772
```

```
95,-0.9218706488609314,0.0,0.8008721470832824,1.020974040031433,0.9714065194129944,2.214872121
8109126,0.5967612862586975,-0.35027286410331726,0.0,0.4799988865852356,-0.373565524816513,0.0'
1
```

The correct labels were 1, 1, 1, 0, 1. So the model was able to correctly classify only 60%.

Refinement

For this capstone project I trained 2 different models: a baseline model using XGBoost and a Deep Neural Network. The XGboost model was very easy to create, train and tune using the built-in algorithms of Amazon SageMaker. Designing the Deep Neural Network was far more complicated and labor intensive.

The table below presents the AUC score for each of the different models.

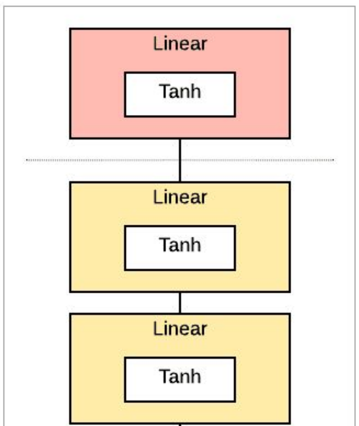
Model	**AUC score**
XGBoost (initial)	0.66
XGBoost (tuned)	0.67
Shallow NN	0.54
Deep NN (initial)	0.67
Deep NN (tuned)	0.70

Results

Model Evaluation and Validation

The training led me to the following final network:

- Input layer: Fully-connected layer with 17 neurons, tanh activation function, batch normalization and dropout.



- Hidden layers: 3 fully-connected hidden layers with 680 nodes, tanh activation function, Batch normalization and dropout.
- Output layer: fully-connected layer with 1 node and sigmoid activation function.
- Dropout: 30%
- Optimizer: Adam
- Learning rate: 0.001
- Loss function: Binary cross entropy

The model was tested against the test set in order to determine its final performance. The AUC score was 0.70.

Justification

The benchmark results were obtained from the research paper[12] referred to in the data set. The values in the table below represent the AUC score for different models under 3 different scenarios (only low-level features, only high-level features and all features).

Model	**Low-level features**	**High-level features**	**All features**
Boosted Decision Trees	0.73	0.78	0.81
Shallow NN	0.73	0.78	0.82
Deep NN	0.88	0.80	0.88

As one of the goals was to train a model that would not require the high-level features, I focused only training only with the low-level features. The table below contains the AUC score on the test set.

Model	**AUC score**
XGBoost (initial)	0.66
XGBoost (tuned)	0.67

[12] Baldi, P. et al. Searching for exotic particles in high-energy physics with deep learning. Nat. Commun. 5:4308 doi: 10.1038/ncomms5308 (2014)

Shallow NN	0.54
Deep NN (initial)	0.67
Deep NN (tuned)	0.70

Obviously the performance of the models is not sufficient when compared with the benchmark. Even the tuned deep NN does not perform to the same level as the benchmark model using either Boosted Decision Trees or Shallow NNs (0.70 vs 0.73).

There are a few explanations for this difference that I can think of.

- I only used part of the entire data set. Every time during training I increased the size of the data set, I noticed an increase in model performance. I used at most about 9% of the entire data set due to the amount of time it took to train (the final retraining of the model using 9% of the data set took about 45min using a GPU accelerated training instance. Extrapolating the training time on the entire data set would take 8h+ for the current model).
- The model architecture was perhaps too simplistic and as such could not capture all details or information from the data set to classify better.