Guide to Machine Learning scripts

Niamh Holland

November 2021

Contents

1 Introduction

This document is a guide to using the machine learning scripts written for reducing the backgrounds in the WATCHMAN detector. These can be found at: `https://github.com/niamh-h/MLEARN`. It will outline the steps needed to use the scripts and briefly cover how they work. The model used in both machine learning scripts is called AdaBoost which is a boosted decision tree classifier (see Appendix A).

The input files of data should be in the form of ROOT data trees. The overall process of using machine learning models on this data is is in two main steps. The first step is to train the models, where they "learn" how to use the input variables to classify the events. The next step is then running these models on unseen data to validate their performance.

The current models are trained to identify either fast neutron events or Lithium-9 events in the whole data-set (out of all other events in the detector including the signal). This is different to how classification models are usually used to separate the signal from background events. In the remote reactor study only the fast neutron model was used, but both are included here. The two models can then be combined on the validation data - which may have already been fed through LEARN. The result is data with the majority of fast neutron events removed and a useful amount of the Lithium-9 events removed, with minimal loss of signal events. These models are combined in "series", with the better performing fast neutron model applied first followed by the Lithium-9 model. As such, the Lithium-9 model is not trained on data from fast neutron events as it's assumed that when combined, all of these are removed by the fast neutron model first.

2 Data

2.1 Data Format

The data after reconstruction is in ROOT files. The first step of the machine learning is to open the files containing the event data being used. This should include events from Lithium-9, Nitrogen-17, Geoneutrinos, Fast Neutrons, the relevant reactors for the chosen scenario, and the worldwide reactor background. The relevant data variables are extracted and put into Pandas DataFrames. Simple cuts are also applied here - most basically $n100 > 0$ to filter any events that failed the reconstruction. More cuts can easily be added to this part and will be applied to all the data being extracted. The module used to open the ROOT files is called Uproot, information on using it can be found here `https://github.com/scikit-hep/uproot4`.

To run the script the data must follow the following conventions:

- They must be in ROOT file format.
- They must all be in the same directory as each other (not the working directory).
- They must be the only ROOT files in that directory.
- They must be named with the source of the events as the first word, e.g. for a training file of Hartlepool A-1 events the file must begin with "hartlepool1_", so could be named "hartlepool1_training.root". This is essential for the script to ensure that all sources are present and label the data correctly.

Currently, the scripts look for both Hartlepool signals and a Heysham and Torness signal, alongside the other correlated background sources. If a different scenario is being looked at such as Heysham-2, this needs to be modified.

When the scripts are run the first thing they ask for is the path to the data. This should be inputted from the /home directory (or whatever this corresponds to) and should end with "*.root". This indicates the script to look for the ROOT files within the listed directory. For example, if the data is kept in a file called "training_data", the path could be inputted as `/home/user/training_data/*.root`.

2.2 Choice of variables

The input data currently being used are the following 22 variables:

- n100 (and n100_prev) for gd-wbls, n9 and n9_prev for gd-water
- dt_prev_us
- inner_hit (and inner_hit_prev)
- beta variables (1 through 6, including beta_prev)
- good_pos (and good_pos_prev)
- closestPMT (and closestPMT_prev)
- drPrevr

These vary in their level of use between the fast neutron and Lithium models. However, the models must be fed the same dimension of data that they were trained on, so in order to put all the data together and run multiple models, the same data must be extracted for both. The less helpful variables do not hinder the model so this has no effect on the success. **This also means that a new model must be created if new variables are added - i.e. re-run the training script and save another model for the correct data dimensions**.

The suitability of the variables can be assessed from a comparison plot of their distributions for each event type, as seen in Figure 1. The difference in shape shows that the machine learning can use the variable to separate the event types.

2.3 Total Monte Carlo events

The total Monte Carlo events that were originally simulated in each of the files is needed for the dwell time calculation. If the simulations are run as pairs, one event in the ROOT file corresponds to prompt event and a delayed event. So, the total mc events is $nevents \times 2$. If the value of $nevents$ is unknown, it can be accessed either by running

```
data->Show(0)
```

in ROOT, or by creating a TBrowser in the directory containing them and opening the *nevents* leaf for each file.

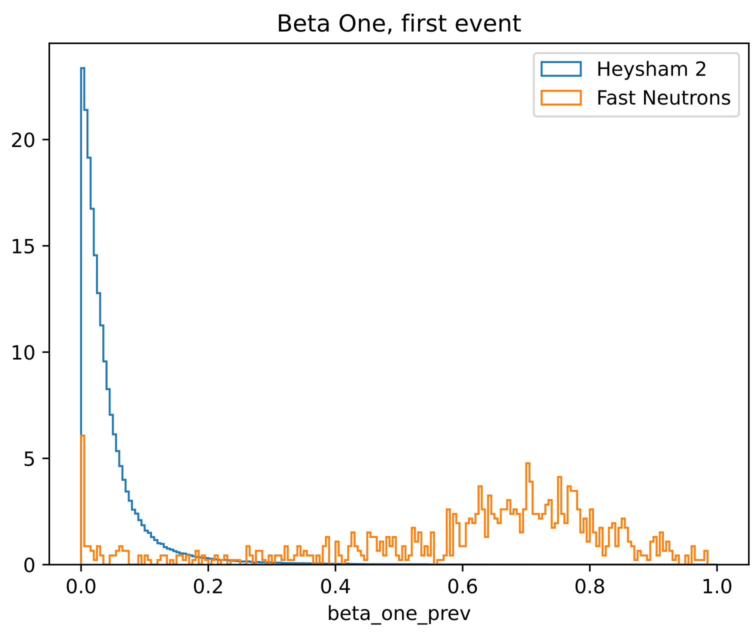

Figure 1: A plot of the first Beta variable for the prompt event for two sources: Inverse Beta Decay from Heysham-2 and a fast neutron. The difference in shape shows the variable is a useful input to AdaBoost.

3 Model training and validation

3.1 Training

The training file for the fast neutron model is **fn_finder.py**, and for Lithium-9 it is **li9_finder.py**.

3.1.1 Steps in the script

The structure of these scripts is essentially the same. They both take as their input the output files from FRED in ROOT format. These are input into Pandas DataFrames for easy manipulation in the script. An additional label column is added to the data:
In the fast neutron finder:

Label = 0 means the data is ***not*** fast neutrons,

Label = 1 means it ***is*** fast neutrons.

In the Lithium-9 finder:

Label = 0 means the data is ***not*** Lithium-9,

Label = 1 means it ***is*** Lithium-9.

The label column acts as the target values and is what the model actually predicts each event to be (either 1 or 0). This is why this is called a binary classification. The data frames are then concatenated into one DataFrame, hence the need for all the data to have the same dimensions.

The label column is then separated from the rest of the data and they are named y (labels) and X (rest of data). This is the form of the data that is fed to the model to train it – X contains the data for the model to use to learn to classify events, y contains the true classification of each event and acts as a cross check.

Before feeding into the model, the data is further split into a training set and a testing set. This is standard good practice as the success of the model can be judged on unseen data from the testing set once it has been trained on the training set.

```
train_X, test_X, train_y, test_y = train_test_split(X, y)
```

The model is then created

```
clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2),
                         n_estimators=100, learning_rate=0.1)  .
```

max_depth refers the the number of branches in each decision tree. *n_estimators* refers to the number of trees in the forest, *learning_rate* controls how each tree contributes to the final classification of an event. These parameters can be adjusted. There is an optional **GridSearch.py** script available that takes multiple variations of these parameters and returns the mean accuracy score each one would yield. However, little difference has been seen between these results and the current values.

The trained model is then applied to the remaining testing data:

```
pred = clf.predict(test_X)
```

Returns the predicted classifications of each event in the testing set – in the form of an array of 1 or 0 for every event in the testing set.

```
prob = clf.predict_proba(test_X)
```

Returns the probability (determined by the classifier) an event is 1 and a probability an event is 0. The probabilities of every event being 1 are used to plot the Receiver Operating Characteristic (ROC) curve later on.

```
scores = clf.decision_function(test_X)
```

Returns the level of confidence in the predicted class by the model. Sometimes described as the 'distance' a prediction is from the decision boundary, the further this is, the higher the confidence in the decision. This is used to plot the distribution of signal/background separation achieved by the model, called the decision function.

The trained model is saved into the current working directory as a SAV file. This will be used in the validation script. The rest of the script creates multiple figures and scores that indicate the level of success the model has achieved in classifying the testing data set, which are saved in the working directory as PDFs. These will be explained in the next section.

3.1.2 Figures

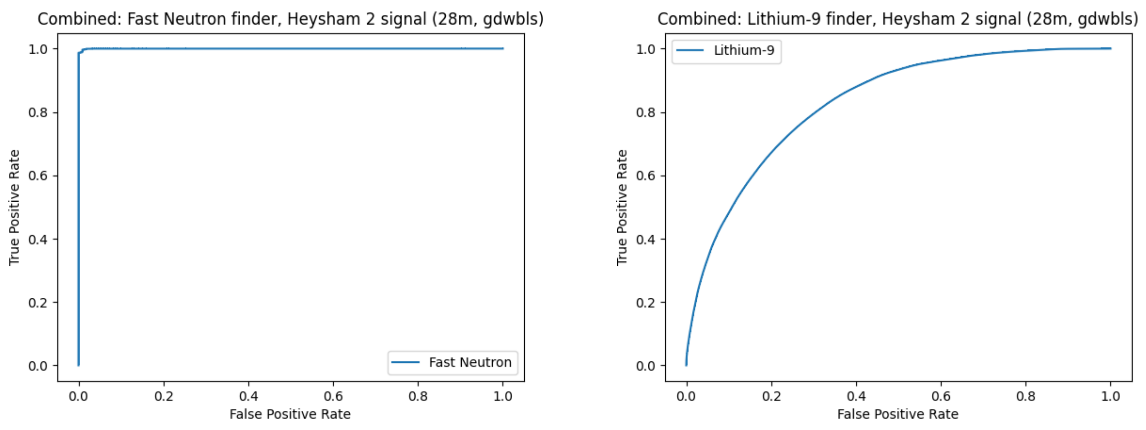

Figure 2: Two examples of ROC curves for the fast neutron model (left) and the Lithium-9 model (right). Produced on the same data in **combine.py**, where the fast neutron model is applied first, followed by the Lithium-9 model.

There are multiple methods of assessing the success of the models. In the scripts, a confusion matrix is printed and put into a figure. This gives the raw numbers of events from each label (in the testing set in the case of the training scripts) that are correctly and incorrectly classified. An example of a

Confusion Matrix plot can be seen in Figure 4. A classification report is also created, which gives the values of precision and recall for each label. Below, tp refers to the raw number of true positives, fp to false positives, tn to true negatives, and fn to false negatives. So, an event labelled as 1 when it is 1 = true positive, labelled as 1 when it is 0 = false positive, and vice versa for true/false negatives.

$$Precision = \frac{tp}{tp + fp} \tag{1}$$

or $\frac{tn}{tn+fn}$ depending on which class one is looking at.

$$Recall = \frac{tp}{tp + fn}. \tag{2}$$

The f1 score is the harmonic mean of these:

$$f1score = \frac{2 \times precision \times recall}{precision + recall}. \tag{3}$$

The classification report also gives the accuracy score of the model overall. An ROC curve is also created. First the predicted probabilities of being 1 (or positive) for every event and their true labels are used to calculate the true positive rate, tpr, and the false positive rate fpr:

$$tpr = \frac{tp}{tp + fn}(= Recall), \tag{4}$$

$$fpr = \frac{fp}{tn + fp}. \tag{5}$$

In this line:

```
signal = prob[:,1]
```

the predicted probability of each event being from the positive class are put into an array. This is used with the true labels to calculate tpr and fpr

```
fpr, tpr, _ = roc_curve(y, signal) .
```

The area under the ROC curve is also calculated for further comparison between curves

```
auc = auc(fpr, tpr) .
```

These are plotted against each other for different thresholds of the probability value. The ideal threshold is where the true positive is maximum while the false positive rate is minimum. Usually, an ideal ROC is a step-like graph, corresponding to a threshold that gives a tpr of 1 and fpr of 0. Examples of these are seen in Figure 2.

The final figure (Figure 3) created is the plot of the confidence scores of the events given by the classifier - the decision function. Since events classified as 0 are given a negative confidence score while those classified as 1 a positive score, this plot can show the level of separation achieved by the classifier. It can also show the level of crossover. Since the use of the classifier is to find a specific event (fast neutrons or Lithium-9), the crossover would be any "other" source incorrectly labelled as the specified event, or the specified event incorrectly labelled as "other". To demonstrate, the labels on the fast neutron plot are as follows:

True fast neutron $\longrightarrow$ label is 1 and classifier predicts 1.

False fast neutron $\longrightarrow$ label is 0, classifier predicts 1 (this could be other sources of background that do not get through, or lost signal).

True "other" $\longrightarrow$ label is 0 and classifier predicts 0.

False "other" $\longrightarrow$ label is 1, classifier predicts 0 (these are fast neutrons that get through).

Once the **fn_finder.py** and **li9_finder.py** files have been run and the models have been created and saved, the next step is to run the **validation.py** script.

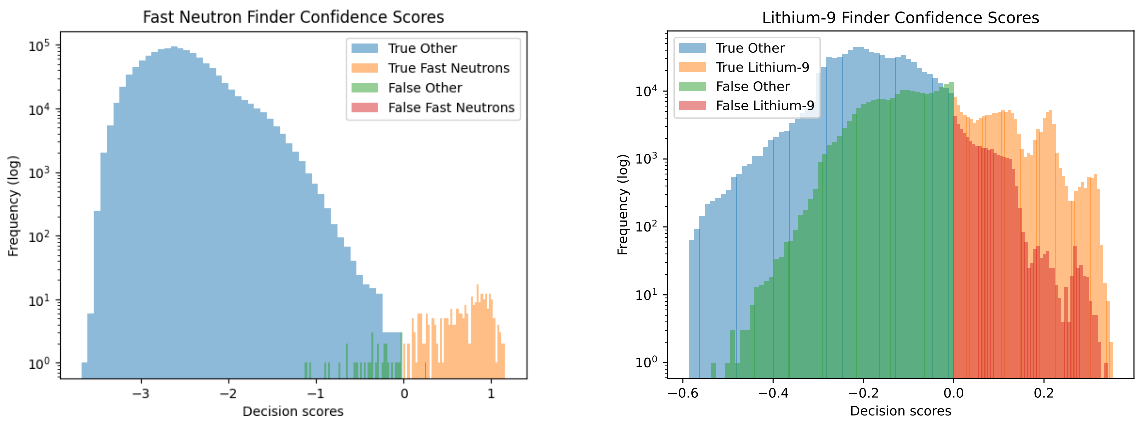

Figure 3: Two examples of confidence score plots for the fast neutron model (left) and the Lithium-9 model (right). Produced on the same data in **validation.py**..

3.2 Validation

The validation script follows the same procedure as the training scripts except there is no splitting of the data as the model has already been trained. It is imported, along with the validation (unseen) data. The label column is added the same as before, but an additional column is added that corresponds to the exact source of each event. So, events have a label, 1 or 0, that the model predicts, and a source, 1-9, that specifies it's actual original source. This source column is created and then removed and saved before the model is applied. It is **not** an input variable. This means that all the event sources listed must be input (again if looking at a different scenario then this should b modified). Once the data is classified, it is added back to the data-frame alongside the classifier's predictions, probabilities, and scores. The corresponding numbers for each source are shown in Table 1. These source numbers are consistent between all models and data.

Source	**Number**
Fast Neutrons	1
Geoneutrinos	2
Hartlepool-1	3
Hartlepool-2	4
Heysham	5
Lithium-9	6
Nitrogen-17	7
Torness	8
World	9

Table 1: Correlated background sources and their corresponding number used to label them in **validation.py** and **combine.py**.

The model is then applied as in the training scripts and the same figures are created. The final step is the input data files are copied into the working directory and then updated with the machine learning information. This includes the predictions, probabilities, and confidence scores for each event. The data is also put into a csv file as a backup. So, when the copied data files are opened in ROOT there will be an addition tree containing this information.

There may be a warning about the number of bytes when the copied data files are re-opened in ROOT. This cannot currently be avoided, hence the data files are copied to avoid disrupting original files. However, this should not be a problem for accessing the data, as long as the Uproot module is up to date. Another reason the data is copied is because files can only be updated in this way once. The tree then already exists so cannot be created again, so the script could not be re-run. If the files are re-copied then the tree is deleted and can then be re-created.

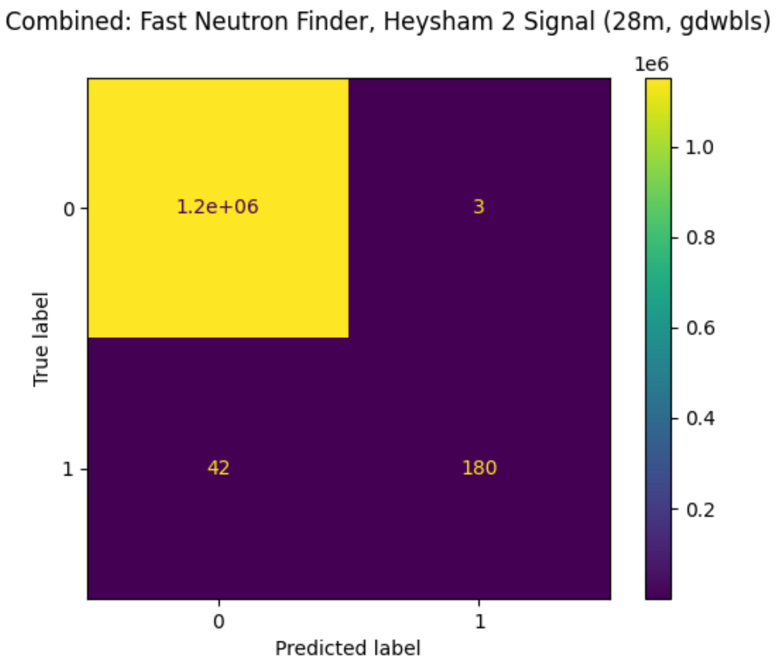

Figure 4: A confusion matrix, where fast neutron data is labelled as 1 and all other data is labelled as 0.

3.3 Combination of models (Optional)

The combine file works in the same way as the validation file – running the model on the entire data-set. However, first it runs the fast neutron model and generates the usual figures, then it feeds forward the data kept by the fast neutron model to the Lithium-9 model. The machine learning data from this run is added to copied versions of the data files as before, and the final classified data set is saved to another csv file as a backup.

4 Dwell Time Calculation (Optional)

The **dwell_time.py** file can be used to calculate the dwell time that the classification has achieved. When run, the file will ask for which classifier last ran on the data, the detector size in metres and the total simulated Monte Carlo events for each event type (the way to find this value is given in section 2.3). The size of the tank is needed to set the rates of each event and the total Monte Carlo events are needed to calculate the efficiency of the model at removing each source of background. Currently, the tank sizes can be 16m, 22m, or 28m - so this cannot be used to make dwell time estimates for the LEC detector yet (the rates of each event are needed for this). It will also ask for the source of the signal - Hartlepool, Heysham-2, Heysham + Torness, or Heysham full. The script then calculates the dwell time using the following equation

$$N_\sigma = \frac{st}{\sqrt{\sum_i b_i t}}, \tag{6}$$

rearranged for t gives

$$t = \frac{N_\sigma^2 \sum_i b_i}{s^2}. \tag{7}$$

The value of N_σ is set to 3 for the discovery scenario, this can be easily changed within the script if needed. The rates of each event after the model is applied are calculated by finding the "efficiency" of the model for each source:

$$efficiency = \frac{no.\ events\ left}{total\ no.\ simulated\ events}\ . \tag{8}$$

This value is calculated for each event source, and then multiplied by the original corresponding rate of the event to get the estimated rate of each event post-model application:

$$new\ rate = efficiency\ *\ original\ rate\ . \tag{9}$$

The new rates are then substituted into equation 7. This gives a rough idea of the dwell time given by the model as the error used is a binomial calculation on the efficiency and no systematic errors are included. However, it is useful for gauging roughly how well the models are performing.

A Boosted Decision Trees

The basic model of the algorithm is a decision tree. Beginning with the data sample as a whole, decisions are made based on the input information and lead to a split in the sample. The subsequent two samples are represented by being on two separate branches of the tree. For example, take an algorithm splitting some data into signal and background, the first input variable to be considered is the number of PMT hits. The first branch of the tree contains all the data then, at the first split (or node) a value of the PMT hits is used as a threshold to split the sample. On either resulting branch, there will be a certain number of signal and background data points. The value of the threshold is chosen by which gives the best separation of signal and background between the two branches. This then continues on the branch that increases this separation, using the next input variable to make the cut. On the other branch, a leaf is created and labelled as whichever class dominates in that sample (if signal dominates, it is a signal leaf and vice versa.). In this way, the data is continually split at nodes until either: a set number of splits is reached, all of the leaves are pure, or the data runs out. The measure of purity is what the quality of separation is based on, and involves giving each event in the data-set a weight, either individually or per class. The purity is defined in Equation 10 [1].

$$P = \frac{\sum_s W_s}{\sum_s W_s + \sum_b W_b} \tag{10}$$

Once this process is complete, the tree contains leaves that are labelled as either signal or background. A data point that passes through the tree and ends up one of these leaves is then classified accordingly. AdaBoost is a boosting algorithm, which is an example of an ensemble method. These combine the outputs of multiple weak estimators to give a more accurate final prediction [2]. Boosting involves changing the weights given to events based on how easy they are to classify by the weak classifiers. In this case, the weak classifiers are small decision trees and the final ensemble model is a boosted decision tree. The weak decision trees often only have around 1-8 branches and are referred to as decision stumps [3]. Data begins with equal weighting (different to the weights previously used to calculate the purity of a leaf), for example, $1/n$ each for n events. It then passes through the first decision stump; if an event is misclassified (i.e. a signal event ends on a background leaf), the weight of that event is increased. The now unequally weighted data points are then fed to the next decision stump, where the same process occurs. This process of increasing the weights on continually misclassified events means the more difficult data points are more likely to be correctly classified.

Fundamentally, this is how the binary AdaBoost classification algorithm works; however, the way it re-weights misclassified events is slightly different. Beginning with all n events having equal weights, $1/n$, they are fed to the first decision stump which creates a hypothesis. The error, ε_t, in this hypothesis is then calculated by summing the weights of the misclassified events. A variable called β is set as

$$\beta = \frac{\varepsilon_t}{1 - \varepsilon_t}. \tag{11}$$

The error on the decision stump must also be less than 0.5 (so it must only be slightly better than random guessing), meaning the value of beta will always be between 0 and 1 [4]. The events are then re-weighted by multiplying the weight by β if it was correctly classified or by 1 if it was incorrectly classified. The new weights are then renormalised before continuation to the next decision stump, where the process starts again with the next decision stump [4]. In this way, the correctly classified events are minimised, thus boosting the misclassified events. This is a qualitative description of how the binary AdaBoost classification algorithm works; for more detail see [4].

In binary classification, the data is classified into two possible categories. When a data point is fed through AdaBoost, it is given a score depending on what category leaf it ends on in each decision stump. If it lands on a signal leaf, it is given a score of 1, if a background leaf, it is given a score of -1. The final score of the event is given by the sum of the scores from each decision stump, often then normalised over the number of decision stumps. This score therefore acts as a confidence score of the classification of that event [1]. For example, if there are 40 trees, the maximum score an event can receive is 40 and the minimum is -40. The former corresponding to an event that is classified as signal by every weak learner, the latter as background by every weak learner. Similarly, if the scores are normalised, then the final scores would be between -1 and 1. An event with a score of 0 would show that it was classified as signal and background an equal number of times, suggesting low confidence in its final classification. These scores can be plotted, giving the show of separation between the two categories, demonstrated in Section 3.1.2.

References

[1] Byron P Roe et al. "Boosted decision trees as an alternative to artificial neural networks for particle identification". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 543.2–3 (2005), pp. 577–584.

[2] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[3] J. Brownlee. *Machine Learning Mastery.* 2021. URL: `https://machinelearningmastery.com/`.

[4] Yoav Freund, Robert E Schapire, et al. "Experiments with a new boosting algorithm". In: *icml.* Vol. 96. Citeseer. 1996, pp. 148–156.