Guide to Machine Learning scripts

Niamh Holland

September 2021

1 Introduction

This document is a guide to using the machine learning scripts written for reducing the backgrounds in the WATCHMAN detector. The machine learning scripts can be found at: https://github.com/niamh-h/MLEARN .

The input files of data should be in the form of ROOT data trees. The useful data is extracted into text files using the extract macro file. These text files are then input into the machine learning scripts. 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 method uses two machine learning models; one classifies fast neutron events against the rest of the data, another classifies lithium-9 events against the data. These models will then be combined on the final validation data, that has been previously fed through the Likelihood Analysis. Thus, ending with data that has 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 will be 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 all of these are removed by the fast neutron model. The model used in both machine learning scripts is called AdaBoost and is a boosted decision tree.

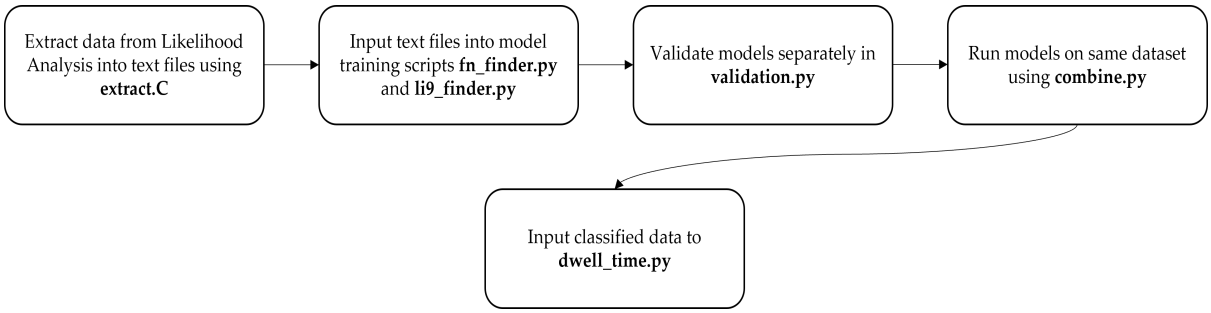

Figure 1: Process of using the machine learning scripts.

2 Extracting the data

2.1 Using extract.C

The simulated data will be in the form of a ROOT data tree. This must be extracted into a usable format to input into the Python machine learning scripts. The extract macro file allows you to choose the variables to extract and inputs them into a text file. It does this for the signal and the sources of background being simulated. These are currently: Lithium-9, Nitrogen-17, Fast Neutrons, Geoneutrinos, Worldwide reactors (excluding Hartlepool, Heysham and Torness which are simulated separately so different signal scenarios can be easily selected). The corresponding header file **extract.h** is needed for **extract.C** to work, and should be edited to look for one of the data files being used (edit lines 199 and 201). If this doesn't work, a new header file can be generated and used alongside **extract.C** with the following commands in ROOT:

```
 TFile myfile("name_of_a_data_file.root")
data->MakeClass("extract")
```

To extract another variable, for example 'mc_energy', add a variable to the list at the beginning:

```
Double_t x;
```

Then add a reference to the branch this data is on:

```
tr->SetBranchAddress('mc_energy', &x);
```

Then add the variable to the input stream into the text file, making sure there is a single space between it and the previous variable:

```
... ' ' << x << ' ' << ... ;
```

Repeat this for every data file, so every source of background will have the same variables. This is essential for when it comes to loading these textfiles into the Python script as dataframes as they must all have the same dimensions (Eg must all have 14 variables corresponding to 14 columns in a dataframe). Once the variables and file paths are appropriately input to the file, it is run using the following commands in root:

```
.L extract.C
extract a
a.Loop()
```

This will generate text files of the chosen data for each source of background/signal, in the chosen file path.

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**.

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. Currently, the simulations are run as pairs, so one event in the root file corresponds to a positron and a neutron (in the case of ibd). So, the total mc events is $nevents \times 2$. The value of *nevents* can be accessed either by running

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

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

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 text files that have been extracted from the root data files. 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. The separate data 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. For example, in the **fn_finder.py** file:

```
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 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 tested on the 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. Or described as the 'distance' a prediction is from the the decision boundary – the further it is, the higher the confidence in the decision. Can be used to plot the distribution of signal/background separation achieved by the model.

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. Finally the testing set is saved to an csv file and the trained model to a sav file.

The **li9_finder.py** script works in the exact same way, the only differences are the lack of fast neutron data and the lithium-9 data is re-labelled as 1.

3.1.2 Figures

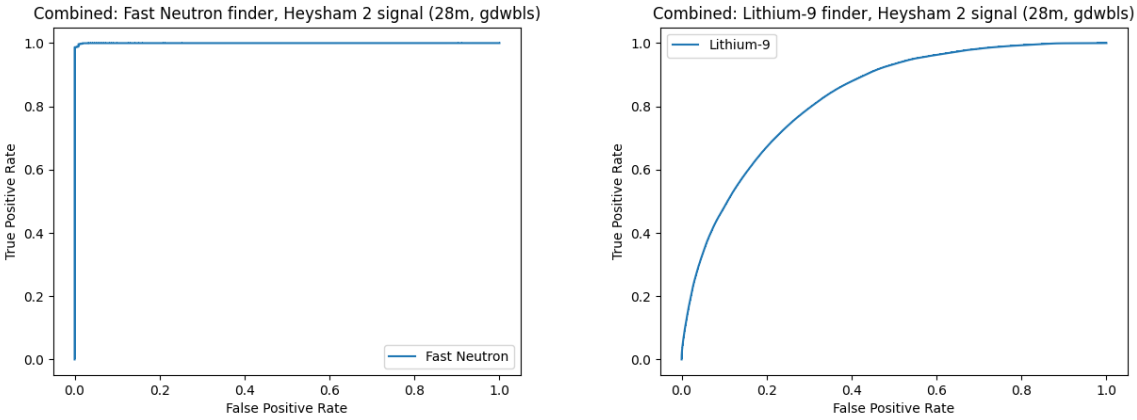

Figure 2: Two examples of ROC curves for the fast neutron model (left) and the lithium-9 model (right). Produced on the same data in **combine.py**, where the fast neutron model is applied first, followed by the lithium-9 model.

There are multiple methods of assessing the 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) that are correctly and incorrectly classified. 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. i.e 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. Finally, an ROC curve is 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 Figures 3 and 2.

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

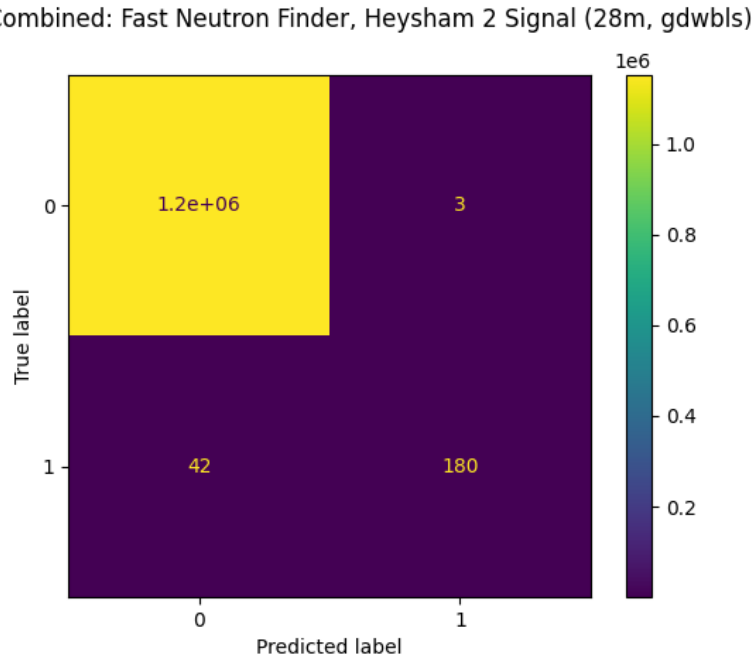

Figure 3: A confusion matrix, where fast neutron data is labelled as 1 and all other data is labelled as 0.

3.2 Validation

The validation script is for good practise but does not necessarily need to be done as the combine file also acts as a validation script. However some important steps are involved including adding another column to the data that labels the exact source of each event. So, events have a label, 1 or 0, that the model predicts, and a source, 1-7, 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. 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. The same method is used on the Monte Carlo energy (which also **should not** be used as an input variable); it is added and removed before the model is applied, and then re-added afterwards. This is useful for seeing the energy of the events that are kept by the model, allowing the potential use of an energy fitter to be assessed.

3.3 Combination of models

The combine file works in the same way as the validation file – running the model on the entire dataset. However, first it runs the fast neutron model, outputting the classified data into a csv and generating

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

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

the usual figures, then it feeds forward the data kept by the neutron model to the lithium model. The final classified dataset is saved to another csv file. The only changes to be made to run the file are the correct file paths to the data and to save the new data, and the figure titles and file paths.

4 Dwell Time Calculation

The **dwell_time.py** file can be used to calculate the dwell time that the classification has achieved. The only edit needed to the file is the file path to the classified data outputted at the end of the **combine.py** script. When run, the file will ask for the detector size in metres, the number of remaining lithium-9 events with Monte Carlo energy above 6 MeV(this is why mc_energy is added despite not being an input variable), and the total simulated Monte Carlo events for each event type (the guide to find this value is given in section 2.3). The number of events above 6 MeV can be found by inputting the final classified data into the **decision_function.py** script, which will print the values. It also creates a plot of the classified data based on the confidence of the model, i.e. the decision function, which is another useful figure. The number of lithium events above 6 MeV are subtracted from the total remaining lithium events, as an energy fitter can be used to remove these. The size of the tank is needed to set the rates of each event and the total MC events are needed to calculate the efficiency of the model at removing each source of background. 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

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

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 energy fitter isn't actually applied and the error used is a binomial calculation on the efficiency. However, it is useful to gauge roughly how well the models are performing.