Topics in Particle Physics and Astroparticles II

Guilherme Soares

Laboratory of Instrumentation and Experimental Particle Physics

31 March 2022

Index

Kinematic Properties

The required background levels at SHiP are of < 0.1 events over the course of the experiment.

Resources :

- PID
- **Kinematic Features**

Total Momentum
Transverse Momentum
Fraction of Transverse Momentum
Opening Angle
Impact Parameter
Distance of Closest Approach (DOCA)
Coordinates of the Decay Vertex

Tutorial 3 - Data Analysis with Pandas

The 3^{rd} tutorial can be found on the usual places, with the name Template_Data_Analysis.ipynb

This template will show you the basics on handling dataframes with pandas.

The topics covered will be the following :

- Printing the DataFrame
- Reading Entries one at a time
- Concatenating DataFrames
- Adding and Removing Columns
- Bulk Dataset Cuts
- Plotting Variables
- Normalizing Data

Tutorial 3 - Data Analysis with Pandas

In order to start messing around with Pandas dataframes, first we need some data to import. Available on the "Data" folder are 2 files with the names Signal.dat and Background.dat. These will be the files that we will look at with this template.

Both .dat files are saved in a .csv format. This means that each line corresponds to an event, and variables are separated by commas ",", with the first line having the name of each variable. This format can be imported to Pandas dataframes directly through pandas : :readcsv("path/to/file").

The first two blocks of code are very simple commands : **pandas : :head()** and **pandas : :tail()**. This commands function in a similar way to a print(), but only show the first and last 5 entries of the dataframe, respectively, alongside the variable names.

Although these commands are not very helpful while working the data, they are great ways to verify the structure of the dataframe, and quick check to know that everything is going well when altering the dataframe in bulk.

In order to read the dataframe, usual iteration methods work fine. Notice that to iterate along a column you need to write a variable explicitly in an analogous way to .root files
$\rightarrow$ *for entry in Signal["variable"] :*

Tutorial 3 - Data Analysis with Pandas

Merging dataframes is very useful. Pandas does this comfortably with the pandas : :concat([dataframe1,dataframe2],ignore_index=True).

If we want to add a column with a flag that distinguishes 2 different dataframes before we merge them, we can also add it with another simple command dataframe['NewFlag'] = 1. This creates a new column where every entry has the value 1.

If for some reason you want to remove a column from your dataset, you can also do it in bulk by utilizing the pandas : :dataframe.drop() function. Although the example shown in the code only refers to eliminating columns, the dataframe.drop() function is much more powerful and will be utilized in our future Machine Learning classes.

One feature to take into account when working with pandas dataframes is that most functions have an *inplace* option. This refers to alterations "inplace", which means in the dataframe itself. This option is False by default, as most of the time you want to create a new dataframe without altering the previous one, to avoid overwriting crucial data.

The last great function that pandas provides is to perform bulk cuts on the dataframes, based on specific variables. Utilizing the following operator *newdataframe = dataframe[dataframe["variables"] > 2]* makes it so that newdataframe will be a copy of dataframe, but only events where our variable is bigger than 2 will be copied.

Tutorial 3 - Plotting Pandas Dataframes

Drawing Plots with Matplotlib :

In order to plot datasets I use 2 different methods. The first is through the old reliable matplotlib. The second is through the seaborn library.

While no library is strictly better, I prefer using seaborn when overlapping distributions from different datasets (or samples), as it presents an automatic way of distinguishing the distributions, already normalized. Apart from this, it can also take the points argument in a flexible way, as either a traditional array or as a column from a pandas : :dataframe, saving a couple of lines of code.

Although I use seaborn to perform one-dimensional plots, matplotlib is the quintessential python plot drawing tool, and I use it when performing 2-dimensional histograms, as it can be customized to give a TH* look. Hence I present a 2-dimensional plot in the template.
In order to do this, we first create a figure (similar to the ROOT TCanvas). While it is not needed here, it is helpful in order to save it using matplotlib.savefig("path_to_save_to"). Afterwards we need to translate our data from the pandas dataframe to classic python lists through a loop.

Tutorial 3 - Plotting Pandas Dataframes

Drawing Plots with Matplotlib :

Then we just create a matplotlib::pyplot::hist2d object as follows :
matplotlib.hist2d(datax, datay, bins=[binsx,binsy], cmap=plotstyle, range=[[xmin,xmax],[ymin,ymax]])
Most arguments are self-explanatory, and the only non-intuitive argument is **cmap**, which defines the plot style. Additionally you may notice that in the template I wrote **bins=100**, which defines both axes as having 100 bins.
The last point to make is that adding a line with matplotlib.colorbar() creates the side bar with the z axis scale.

Tutorial 3 - Plotting Pandas Dataframes

Drawing Plots with Seaborn :

The second option I mentioned is seaborn.
Overlapping distributions can be done through the seaborn::distplot() function :
ax = sns.distplot(x, hist_kws={"range" : [0,250]}, norm_hist = True, kde=False, label="Signal")
Here, **x** is the list of points we want to put into the histogram, and can be defined as simply as *x = Signal['variable']*, **norm_hist** is an option to normalize the distribution to a total area of 1, **kde** gives an automatic fit to a gaussian distribution, **label** assigns a specific label to that distribution and **hist_kws** is the true crown jewel in the options, as it allows you to insert any options available for the matplotlib counterparts (in this case only the matplotlib option range=[0,250] was used).
Drawing a second overlapping distribution is very easy, as it simply requires you to utilize the previous function with a different dataset **y**. Although this would usually overwrite most objects in python, here the new distribution **y** is added to the previously created distplot object.

Tutorial 3 - Drawing Correlation Plots

The last part of this tutorial is related to our next project involving Machine Learning.

While doing a separation analysis you need to be aware that some variables are more correlated with each other than others, and going over features that are highly correlated in order to separate samples is not very efficient.

Pandas provides an automatic way of looking at the correlation matrix of dataframes, through the pandas : :dataframe.corr() function. This method gives the pearson correlation matrix by default, but the method might be changed through a **method** option.

Tutorial 3 - Normalizing Data

Normalizing data can be helpful in a plethora of cases. We will be performing a data normalization in order to apply machine learning methods.

We will be utilizing Pipelines, which are objects from SKLearn that take as inputs transformation functions and perform them in a pipeline.

Our pipeline contains 2 different functions.
The first is the StandardScaler, which renormalizes all of data in order to obtain a gaussian distribution with an average value of 0 and a standard deviation of 1.
The second function is a Principal Component Analysis (PCA). This one transforms the data so that you get components that are as independent of each other as possible, while maximizing the variance, with the idea of reducing the amount of dimensions needed in order to represent a space, while losing the least amount of information possible.

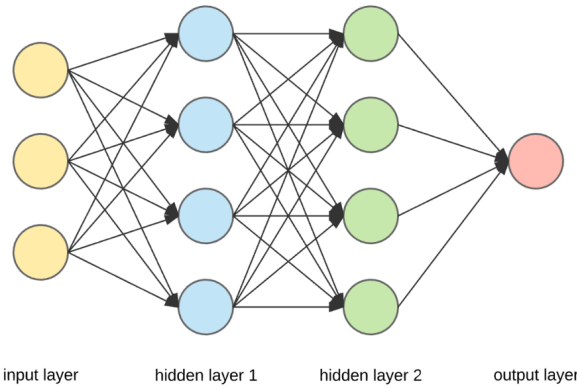
In order to do the normalization, we just need to define a subset of our data with the features that we want to transform and utilize the pipeline.fit_transform(subset) function.
Notice that the output of this function is an array (of floats in our case since we deemed it so in the options).

Machine Learning

Machine Learning algorithms have an advantage over traditional selection methods : **They take into account multi-variable correlations** in a non-trivial way.

We will be utilizing **Multi-layer Perceptrons**, which are feedforward Neural Networks.

Neural Networks are interconnected groups of nodes (neurons) arranged in layers, where each node processes information and passes the result on to the next layer
Input layer is the first layer, that receives the data, and has to have 1 node for each feature of the data



Output layer is the last layer, and usually has as many neurons as the classes that we are trying to classify our data into. The one exception is for binary problems, where just 1 neuron is enough (the chance of not being from a set is the maximum score possible minus the score attained, hence low scores are a measure of being of the other set)

The connections between nodes are defined by **Weights** that are adapted through the training phase, and each of the nodes might have a threshold or a bias. Alongside the structure of the network, the weights are what define the network

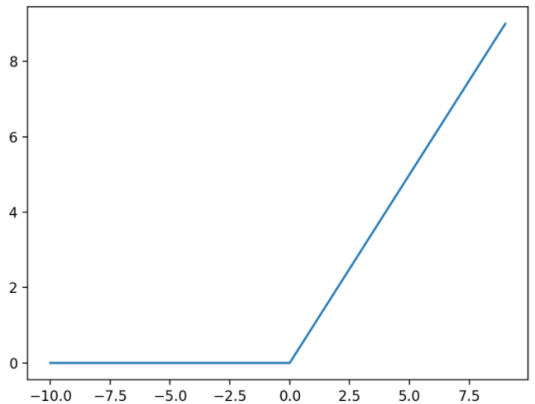
Machine Learning - Activation Functions

Each node processes the received information before transmitting it to the next layer. The biggest difference between MLPs and simple NNs is that the **Activation Functions** for MLPs are non-linear for all layers besides the first, which allows for backpropagation.

This implies that MLPs always have at least 3 layers, since all linear functions can be simplified through linear algebra in a way such that we would only need 2 layers : input and output. The layers in between these are called **Hidden Layers**

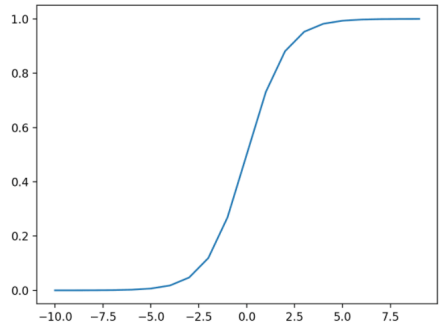
The more complex and bigger these hidden layers are, the longer it takes to process information. As such the activation functions need to be simple, fast to calculate and limited output ranges are preferred.

- Rectified Linear Unit (ReLU)

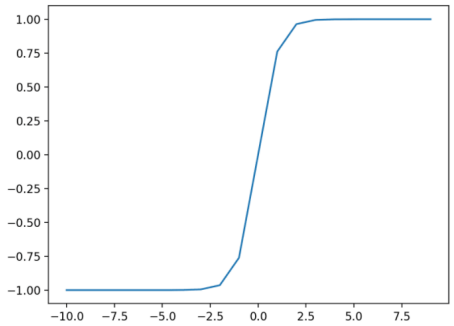


Machine Learning - Activation Functions

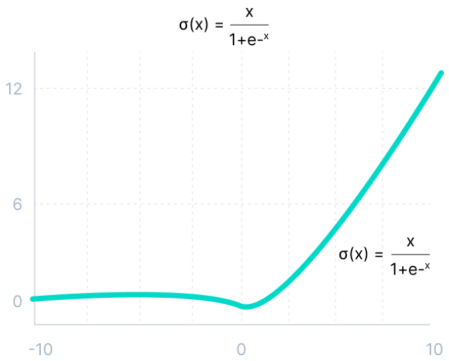
- Sigmoid



- Hiperbolic Tangent (Tanh)



- Swish ($x \times$ Sigmoid)



Machine Learning - More Basic Concepts

MLPs are NNs with **Supervised Learning**, which means that they train by processing examples with clearly defined inputs and results (features of the data, and we know it what is signal/noise à priori), forming a probability weighted association between the two.

This probability (weights) is updated according to a **Loss-Function**, which is commonly the **Cross-Entropy**. This is just a logarithmic function that is skewed towards correct predictions by heavily punishing incorrect predictions

$$\sum_i t_i log\left(p(x_i)\right) + (1 - t_i) log\left(1 - p(x_i)\right)$$

Here t_i is the true value of the event i (usually 1 for signal 0 for background) and $p(x_i)$ is the probability that the MLP attributed to said event.

Since it is impossible to find the optimal value for the weights of NNs, we try to find close to optimal solutions with **Optimization Algorithms**. These are usually stochastic gradient descent algorithms.

Each time the algorithm sees the whole dataset once is called an **Epoch**, but this does not necessarily define the rate at which the **weights** are updated. That rate is known as a **Batch**.

Machine Learning - Last Basic Concepts

The last thing to mention is that **there is** such a thing as **Overfitting**. This happens when the NN becomes so optimized for our data that it is turned obsolete when faced with new data.

In order to circumvent this there are several tools. The first one is setting aside a sample of our data that we do not use in training, but rather use to validate our training after each **epoch**, called **Validation Sample**. This way, we can let our NN run freely for as long as we want, and we can save the weights that give the best performance for the validation sample, that the NN is blind to, and as such should have no biases towards.

The second tool is varying the structure of our MLP. If the MLP is too complex for a problem, there is no avoiding overfitting our data, as it is akin to trying to use a pneumatic hammer to drive a nail.

The third tool is to try and mask information as we progress deeper into the MLP. This can be done through a function that we have already seen in Pandas : **drop()**. This ensures that random fractions of our data are removed, in order to try and avoid biases stemming from a high degree of omniscience.

The last thing to note is that due to all the randomness involved in ML algorithms, which can be as simple as being lucky in the early guesses of the optimization algorithms regarding the optimal solution, even while maintaining everything equal in one NN, training it several times will always yield different results, although they will never be too dissimilar provided we are constructing the Network correctly.