The Jensen-Shannon Divergence and Machine Learning Toolbox

Nicholas Carrara, Jesse Ernst

August 11, 2017

Contents

1 Introduction

This documentation provides explanations for the Jensen-Shannon divergence and machine learning (JSDML) package for python. The first couple sections of this document cover the theoretical implications of the Jensen-Shannon divergence and its practical use in classification problems where there are a high number of variables or when one wants to use a machine learning algorithm as a statistic. For a more thorough discussion see the corresponding paper by Carrara and Ernst[1]. The current version of the JSDML package contains a neural network implementation in Keras[18] which is defaulted to use Tensorflow[19] as the backend. The mutual information (or JSD) estimation method follows from Kraskov[4] and is implemented using code borrowed in part from NPEET[7] (Non-Parametric Entropy Estimation Toolbox). This documentation was written along side version 0.1 of the package which will be updated to a working version 1.0 when additional functionality has been worked out by the authors. For now, the main results of the paper[1] can be reproduced using this code.

2 The Jensen-Shannon Divergence

The Jensen-Shannon Divergence, or JSD, is an intrinsic measure of the distinguishability between two probability distributions. In classification problems it is typically our desire to tell these two distributions apart from one another as best as possible. Because we don't necessarily have access to the probability distributions (we may only have data and the space of variables may be large) it is better to use a sort of meta-statistic or super-statistic (not to be confused with superstatistics in statistical mechanics) such as the JSD to determine the intrinsic distinguishability of these distributions. One may attempt to argue that there are other superstatistics that should also give us a measure of distinguishability, the JSD is the one that is most consistent with reducing our errors in classification. These points are more thoroughly outlined in the corresponding paper[1], but we will briefly cover them here. The Jensen-Shannon Divergence between two distributions $p(x|s)$ and $p(x|b)$, where $\theta = (x, b)$ denote the distribution and x is the dependent variable, is given by

$$\mathcal{JS}[p(x|s), p(x|b)] = \mathcal{S}[p(x)] - p(s)\mathcal{S}[p(x|s)] - p(b)\mathcal{S}[p(x|b)] = \int dx \left(p(x|s) \log \frac{p(x|s)}{p(x)} + p(x|b) \log \frac{p(x|b)}{p(x)} \right) \tag{1}$$

where $p(x) = \sum_\theta p(x, \theta) = p(s)p(x|s) + p(b)p(x|b)$ is the marginal distribution over θ and $\mathcal{S}[p(x)]$ is the Shannon entropy of $p(x)$. The JSD is related to the mutual information[8], which is useful for proving some aspects of the JSD, in the following way

$$\mathcal{JS}[p(x|s), p(x|b)] = \mathcal{M}[x; \theta] = \int dx d\theta p(x, \theta) \log \frac{p(x, \theta)}{p(x)p(\theta)} \tag{2}$$

When the mutual information is cast in this form where θ is a class label, then it is equivalent to the JSD between the two different class distributions. The JSD can behave as a metric since it is symmetric and obeys a triangle inequality. The JSD has been studied extensively in several papers[21],[22],[23],[24],[25]. An important property of the JSD, or the mutual information, is it's invariance under a sufficient statistic of the data, a point proven in the corresponding paper[1]. This can be shown in either of two ways. First consider that the data is subject to a Markov chain;

$$\theta \to x \to y \tag{3}$$

where the class label θ is codified by the variables x, and the variables $x \in \mathbb{R}^N$ are sent through some inferential algorithm or function that produces a variable $y \in \mathbb{R}^M$ such that $M \leq N$. In the case of binary classification it is desirable to have $M = 1$. The Markov chain implies that

$$p(\theta, x, y) = p(\theta)p(x|\theta)p(y|x) \tag{4}$$

Consider now the mutual information between the class variables $p(\theta)$ and the joint distribution of the x's and y's $p(x, y)$ which can be broken up in two different ways

$$\begin{aligned} \mathcal{M}[\theta; x, y] &= \mathcal{M}[\theta; y] + \mathcal{M}[\theta; x|y] \\ &= \mathcal{M}[\theta; x] + \mathcal{M}[\theta; y|x] \end{aligned} \tag{5}$$

Since the answer θ is independent of y given x, the last term in (5) is zero; i.e. $\mathcal{M}[\theta; y|x] = 0$. Since the mutual information is positive definite; i.e. $\mathcal{M}[\theta; x|y] \geq 0$, we have that

$$\mathcal{M}[\theta; x] \geq \mathcal{M}[\theta; y] \tag{6}$$

which means that the mutual information of the original discriminating variables and the answer (class label) is an upper bound and only when $y = f(x)$ is a sufficient statistic for the variables is the mutual information equal; i.e. when

$$p(\theta, x, y) = p(\theta)p(x|\theta)p(y|x) = p(\theta)p(y|\theta)p(x|y) \tag{7}$$

or written another way

$$p(\theta|x, y) = p(\theta|x) = p(\theta|y) \tag{8}$$

Why is this important? Well, whenever an inferential algorithms output is a sufficient statistic, then we know that the type-one and type-two errors given by the output will be the same as the input. We can see this by looking at the likelihood ratio

$$\xi(x) = \frac{p(x|s)}{p(x|b)} = \frac{p(b)}{p(s)} \frac{p(s|x)}{p(b|x)} \tag{9}$$

which by the Neyman-Pearson Lemma is the maximal test, i.e. it gives the minimum type one and type two errors. Whenever the inferential algorithm output is a sufficient statistic then we know from (8) that

$$\xi(y) = \frac{p(b)}{p(s)} \frac{p(s|y)}{p(b|y)} = \frac{p(b)}{p(s)} \frac{p(s|x)}{p(b|x)} = \xi(x) \tag{10}$$

therefore the type-one and type-two errors will also be minimal on the output. While in principle we could just have check each of the $\xi(x)$'s and $\xi(f(x))$'s to see if they were equal, but we don't have access to the actual probability distributions in practice. Thus is we can approximate something like the mutual information, this is a proxy to determining how sufficient of a statistic the inferential algorithm output is.

3 The KSG Estimator

One of the tools we will use in calculating the mutual information is a non-parametric estimation derived by Kraskov[4] and implemented by Ver Steeg in his package called NPEET[7]. We borrow some of the functionality from NPEET in our toolbox but also implement an improvement which is designed specifically for classification problems. The estimation given by Kraskov is the following;

$$\mathcal{M}[x; \theta] = \psi(k) + \psi(N) - \langle \psi(n_x) + \psi(n_\theta) \rangle \tag{11}$$

where $\psi(x)$ is the digamma function which for x equal to an integer can be written as

$$\psi(n) = \sum_{k=1}^{n-1} \frac{1}{k} - \gamma \tag{12}$$

where $\gamma = 0.5772156649$ is the Euler-Mascheroni constant. N is the number of samples, k is the number of nearest neighbors used to find the average Euclidean distance between points in the joint space of x and θ and (n_x, n_θ) are the number of points in the marginal spaces of x and θ which lie within the nearest neighbor distances calculated in the joint space. When the space θ is a binary discrete space, the calculation above simplifies greatly.

3.1 Improvements to the KSG Estimator

We can think of the estimation method above as something which takes the data x and projects it into a high dimensional space made of two sheets given by the values of θ being either signal or background. So each set of signal or background points get separated into their relative sheet. From here, we find the nearest neighbor (or average nearest neighbor) distance for each point on each sheet. Then we project the

sheets back together and search in the region we just calculated to be the nearest neighbor distance. If more points end up being in that region, it gives us an indication of how much the answer (θ) told us about the variables x. If no additional points land in any of the nearest neighbor distances, then the variable θ gives us no information about which distribution x belongs to, since we will never mistake any points in its vicinity with the wrong distribution. If however every point on average leaves us completely confused whenever we project the two sheets back together, then the answer (θ) gave us a maximal amount of information.

Kraskov's estimator was built with continuous distributions in mind. If we were to try and calculate $\psi(n_\theta)$ using the algorithm in NPEET (a KDTree), we would be left sitting for quite a while. Instead, we can make use of the arbitrariness of the values of the class labels and do the following. If we set the values of the binary class label to be such that the distance between them is always larger than any nearest neighbor distance calculated in the joint space (x, θ), then the value of $\psi(n_\theta)$ will be uniquely determined without having to do a search. This is because the nearest neighbors for each n_θ will just be the number of each class label that exists. Therefore we can calculate the quantity

$$\langle \psi(n_\theta) \rangle = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{m=1}^{n_{\theta_i}} \frac{1}{m} \right) - \gamma \tag{13}$$

which we can calculate before very quickly. This improves the speed of the algorithm by at least a third and is implemented in our version of the code.

4 Installing Python and Related Packages

I will provide here the necessary instructions to get started with Keras[18]. I will assume that users have a Linux based system or a Mac. On **Linux** machines this is easily done by installing both Python3[**?**] and pip3[**?**] using apt-get

```
1 $ sudo apt-get install python python-pip
```

That's it. On a **Mac** we will first need to install homebrew[11] by issuing the following command in a terminal

```
1 $ /usr/bin/ruby -e
2 "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

and then running

```
1 $ brew install python
```

This will install python3 along with pip. For **Windows** users, you can try the following, although I haven't tested it.

```
1 $ python -m pip install -U pip setuptools
```

which should install pip and setup tools.

4.1 Related Packages

Now we will need to install the following packages, some of which may already come with homebrew;

- **NumPy**[12] - This is a standard numerical package for python.
- **SciPy**[13] - This is also a standard scientific package for python.
- **Matplotlib**[14] - This contains all of the basic plotting effects that we need.
- **Keras**[18] - Our over-head neural network implementation.

- **TensorFlow**[19] - This is one of the backends that Keras uses for the implementation.
- **Theano**[20] - This is the other backend.

To install these packages we simply need to use pip3 in the terminal by entering the command

```
1 $ pip3 install numpy scipy matplotlib keras tensorflow theano
```

5 Auxiliary and Special Functions

5.1 Importing Data

Importing data for use in this program is pretty straightforward. There is currently only function for importing data given by the following

5.1.1 create_feature_sets_and_labels

The function **create_feature_sets_and_labels(signal, background, num_of_vars=1, test_size = 0.1, file_size=1.0, var_set=0.0)** takes several arguments. The first two should be the individual signal and background files in a **.csv** format. Make sure that the discriminating variables are in the same columns for both files. The function will output four arrays, **train_x, train_y, test_x, test_y**. The **train_x** array contains the ratio of training data specified by the user which is set to default at **90%** but is given by the argument **1 - test_size**. The array **test_x** is the testing data given by the ratio specified by **test_size** which is defaulted to **10%**. The arrays **train_y** and **test_y** are the arrays of answers corresponding to **train_x** and **test_x**. A full description of the parameters are given below;

- **signal** - This is the file which contains the signal data, it should be in a **.csv** format with each column being a separate discriminating variable.
- **background** - Same as signal except this is the background file.
- **var_set** - This should be an array specifying the columns (or variables) that the user wishes to extract from the signal and background files. An example would be **var_set = [0,1,2,4]** which would take the 1st, 2nd, 3rd and 5th columns from both signal and background.
- **num_of_vars** - If **var_set** is not specified in the function argument then **num_of_vars** can be used instead which takes the first N variables in the signal/background files specified by this argument. **var_set** will take priority over **num_of_vars** but if one sets **num_of_vars = 5** for example without specifying **var_set**, it will take the first five variables in the signal and background files.
- **test_size** - This specifies the amount of testing/training ratio to split up the data into. The default is set to 10% but the user can enter any ratio they desire.
- **file_size** - The user can specify the amount of the signal/background files they wish to use with this function, the default of which is set to 100%.

When this function imports the data file, it grabs the entire set of data files and randomizes the data. The amount of the signal and background files are taken to be in equal proportion constrained by the **file_size** parameter. While different amounts of signal and background can be used in machine learning, it does not generally lead to good results, especially with the use of a neural network. Despite the amount of background or signal that may be expected in any experiment, you will want to train the neural network to recognize both in equal proportion to avoid biasing the network towards one regime or the other.

5.1.2 acceptance_rejection_cuts

This function takes a set of signal and background outputs from the neural network and generates a set of CDF's and inverse CDF's by making cuts on the data. There is an option for making the cuts at equal intervals of signal, which is usually done, and then use those values to find the corresponding background acceptance. The function header is given by; **acceptance_rejection_cuts(signal, background, cut_density=0.1, symmetric_signal=True)**. The individual function attributes are as follows.

- **signal** - The signal distribution from the neural network output
- **background** - The background distribution from the neural network output
- **cut_density** - This tells the function how finely to make the cuts. The default reduces to a number of points $n = 1001 \times \text{cut\_density} = 101$. The user can adjust this by changing its input value.
- **symmetric_signal** - This attribute is defaulted to true, which tells the function that the user wants equal divisions of signal, so cut at equal intervals. If set to false the function will cut according to equal intervals in (max - min).

The function will make various cuts and generate the four CDF's and inverse CDF's; signal acceptance, background acceptance, signal rejection, background rejection, and then return these to the user. These arrays can then be used to generate ROC curves with the plotting functions of the neural network class.

5.1.3 avgdigamma2

This is a variation on the **avgdigamma** function that appears in Greg Ver Steegs package NPEET. One of the contributions of the associated paper to this code was to make the mutual information calculation faster when one of the variables is a class label. This is explained briefly in section 3.1 The function header is given by; **avgdigamma2(points,dvec)**. The two arguments are given by;

- **points** - This is the array of class labels θ; e.g. $[1.0, -1.0, \dots]$.
- **dvec** - These are the list of nearest neighbor distances calculated in the joint space of (x, θ).

What the function does is to ask for each point in **points** whether or not the arbitrary distance between class labels, $\Delta = \theta_i - \theta_j$, is larger than the corresponding distance in **dvec**. If not, then the number of nearest neighbors is simply the number of class labels θ_i corresponding to the label associated with **dvec**. If it is larger than Δ, then the number of nearest neighbors is the total number of **points**. This will return $\langle \psi(n_\theta) \rangle$, the average of the digamma function evaluations on those numbers of nearest neighbors.

5.2 The Jensen-Shannon Divergence

We have several options when it comes to calculating the JSD, or mutual information. If our problem is only one-dimensional, we can simply bin the data and calculated the discrete form of the JSD. This tends to be accurate enough when the data isn't terribly sparse. When we have more than one dimension however, we need to appeal to better methods which include the Kraskov estimator that is implemented in part by a package called NPEET.

5.2.1 discrete_JSD

This function calculates a one-dimensional JSD by binning the input values and using the Shannon entropy definition

$$JS[p(x|s), p(x|b)] = S[p(x)] - p(s)S[p(x|s)] - p(b)S[p(x|b)] \tag{14}$$

where $p(x) = \sum_\theta p(x, \theta) = p(s)p(x|s) + p(b)p(x|b)$ is the marginal over the answer (or class labels). The function header is given by; **discrete_JSD(signal, background, num_bins=50)**. The individual arguments are given by;

- **signal** - This is the signal distribution for the one-dimensional problem (either neural network output or a single discriminating variable).

- **background** - The corresponding background for the signal distribution.
- **num_bins** - The number of bins the user wishes to use in calculating the JSD. The default is set to 50.

The function takes the signal and background sample distributions and combines them into the union $(x|s)\bigcup(x|b)$, which is just the normalized distribution $p(x)$ (the marginal). Then based on the number of bins the user specifies, the function calculates

$$S[p(x)] = -\sum_x p(x)\log p(x) \tag{15}$$

for all three distributions $p(x)$, $p(x|s)$ and $p(x|b)$. For any individual probabilities which happen to be $p(x) = 0$, the entropy term ends up also being zero so these values are skipped in the sum. The function returns the JSD value.

5.3 Variations of NPEET

This section briefly touches on the few functions borrowed from NPEET, as well as one modified one **mi**. This function was modified to be optimized for class labels and is at least one third faster than using the original version. For more details see the paper[1] and section 3. For more info on NPEET see [7].

5.3.1 mi

This is a modified version of the original **mi** function found in NPEET. It uses the **avgdigamma2** function outlined earlier which cuts down on the computation time. The function header is given by; **mi(x,y,k=3,base=2)**. The arguments are given by;

- **x** - The set of discriminating variables
- **y** - The set of class labels
- **k** - The number of specified nearest neighbors to average over for finding distances. The default is set to three.
- **base** - An overall scale factor depending on the base of the logarithm the user wants to use. The default is set to two, which is sufficient for classification problems.

This function implements the Kraskov method outlined in section three with a modification that reduces the computation time by at least a third. The function returns the mutual information value, which in this case happens to also by the JSD.

5.3.2 mi2

This is an unmodified NPEET function, except for the name, which calculated the mutual information between two continuous distributions.

5.3.3 avgdigamma

This is also an unmodified NPEET function that finds the average value of the digamma evaluations for $\langle\psi(n)\rangle$, where n is the number of nearest neighbors within a certain distance in the marginal space.

5.3.4 zip2

This utility function is also an unmodified NPEET function that forms the Cartesian product of two variables.

6 Neural Network Class

The neural network implementation in this program comes from Keras which uses Tensorflow as an overhead. One can change this overhead to Theano by changing **line 42** in the main file to read **os.environ['KERAS_BACKEND']='t** instead of **os.environ['KERAS_BACKEND']='tensorflow'**. The current version of this implementation of Keras does not allow for arguments that generalize the type of training algorithm, cost function or other parameters. Most other parameters remains fixed except for the size of the network and a few others. To begin let's look at the constructor.

6.0.1 constructor

The neural network constructor is given by the declaration **def __init__(self, layer_vector, learning_rate=0.1, decay_value=1e-6, momentum_value=0.9, nest=True)**. This function returns a network object that has several other functions to be used with it. The main body of the constructor arranges the appropriate topology of the network and compiles it. The network is arranged as a **fully connected** network with **tanh()** activation functions. The cost function is the **mean squared error** and the training method is given by **batch gradient descent** with some possible parameters including a **momentum** term, **Nesterov accelerated descent**, **learning rate** and a **decay rate**. The network object is ready for training after declaration. Each of the function arguments are explained below.

- **layer_vector** - This is a vector of node numbers. The first entry in the vector should be the number of inputs, the last entry the number of outputs and any middle numbers will be entered as hidden layers. So for example the vector **layer_vector = [4,10,3,1]** will setup a network with four input nodes, one output node and two hidden layers of ten and three nodes each.
- **learning_rate** - This specifies the learning rate for the gradient descent method and is defaulted to **learning_rate = 0.1**.
- **decay_value** - A specification of a decay rate for the learning rate, which is defaulted at **decay_rate = 1e-6**.
- **momentum_value** - This specifies a momentum term in the gradient descent algorithm which is defaulted to the value **momentum_value = 0.9**.
- **nest** - This is a boolean argument which specifies whether or not to use the Nesterov accelerated descent. It is defaulted to **nest = True**.

Apart from these specified arguments there are other parameters of any network object which are the following.

- **parameters** - These are the weight values of the network which can be entered manually or imported from a previous session.
- **normalization** - This is an array of normalization parameters that are found during training.
- **model** - This is the actual Keras object that we are inheriting in this network class.

6.0.2 get_parameters

This function returns a parameter variable that contains the weight values for the network. These are set with Keras' built in functions for setting and getting weight values.

6.0.3 set_parameters

This function takes as argument **set_parameters(self, param)** a param variable which is used to set the parameter values of the network using Keras' built in functionality.

6.0.4 get_normalization

This returns an array object for the normalization parameter saved to the network object

6.0.5 set_normalization

This allows the user to specify the normalization parameters by hand by passing through an array object.

6.0.6 find_normalization_parameters

This is a somewhat involved function that finds the normalization parameters for a particular dataset such that the data is scaled for both the neural network and the mutual information calculation simultaneously. The only parameter for this function is the data you wish to scale to; **find_normalization_parameters(self,data)**. This is usually done before the training and the testing data must also be scaled according to the parameters that are found in this step. The scaling is performed in several steps. First, a copy of the data is created so as not to affect the actual data that was passed to the function. Then standardization is applied to the data such that each variable $x_i \in X$ gets transformed as

$$x_i' = \frac{x_i - \mu_i}{\sigma_i} \tag{16}$$

where (μ_i, σ_i) are the mean and standard deviation for variable i. After this we apply a sigmoid transformation which pulls any outliers in the data away from the edges.

$$x_i'' = \left(1 + e^{-x_i'}\right)^{-1} \tag{17}$$

The third step in this can be slightly cumbersome and requires searching through each variable to find nearest neighbors. KDTree's can be employed here, but instead we brute force the nearest neighbor search on a subset of the data (1000 events). We perform a nearest neighbor search on each point and evaluate the average Euclidean distance in each dimension $\langle |x_i'' - x_{i_{\text{nearest}}}''| \rangle$. We then scale each dimension so that the variable with the largest average Euclidean distance is transformed to that its average Euclidean distance is unity; i.e. the variable with the largest average Euclidean distance is scaled as

$$x_{\langle \max \rangle}''' = \frac{x_{\langle \max \rangle}''}{\langle |x_{\max}'' - x_{\max_{\text{nearest}}}''| \rangle} \tag{18}$$

The other dimensions are scaled as

$$x_i''' = x_i'' \frac{\langle |x_i'' - x_{i_{\text{nearest}}}''| \rangle}{\langle |x_{\max}'' - x_{\max_{\text{nearest}}}''| \rangle} \tag{19}$$

Finally, we subtract the mean from each dimension to move it to zero

$$x_i'''' = x_i''' - \mu_i''' \tag{20}$$

In principle, one can separate the two scalings for the mutual information and the neural network and hand the data to each of those methods independently. We chose to do it all in one step to not have to worry about keeping different normalizations around, but one is certainly able to do it that way if they wish. The parameters that are saved are the values of $(\{\mu_i\}, \{\sigma_i\})$, $(\{\langle |x_i'' - x_{i_{\text{nearest}}}''| \rangle\}, \langle |x_{\max}'' - x_{\max_{\text{nearest}}}''| \rangle)$ and $\{\mu'''\}$.

6.0.7 normalize_data

This function applies the transformation laid out in the previous function given by the normalization parameters that where found there.

6.0.8 train_network

This function takes as argument a training set and the corresponding class labels (answers) and trains the network according to a specified number of epochs and batch size for batch gradient descent. The function has the form **train_network(self, training_data, training_answer, num_epochs=1, batch=256)**. The following are the function arguments.

- **training_data** - This is the collection of training data for the network to be trained on.
- **training_answer** - These are the set of class labels corresponding to the training data which are generally given by a one-dimensional array of values; i.e. $[1.0, -1.0, 1.0, \ldots, 1.0]$.
- **num_epochs** - This specifies the number of epochs desired for the training session. The default is set to one.
- **batch** - This specifies the number of events for the batch gradient descent and can be changed to any value by the user. The default value is 256, but for the paper we used 100.

Performance can vary depending on how one batches the gradient descent algorithm[**?**]. A batch value of one is what's called stochastic gradient descent[**?**]. The number of appropriate epochs can also depend on the type of problem you are trying to solve. Simple low dimensional problems typically don't require a large number of epochs to find the minimum, but more complicated sets of variables can take a lot longer.

The training function normalizes the data upon handing a data set. This normalization will always be recalculated every time the **train_network** function is called so we urge caution with how it is used in conjunction with the **evaluate_network** function. Testing data should be normalized according to the same parameters that were used on the training data. After training is complete the network parameters are saved to the parameters attribute of the object.

6.0.9 evaluate_network

This function passes a set of testing data to the network which evaluates it and returns the network output in an array called **results**. The function header is given by **evaluate_network(self, testing_data, testing_answer, score_output=True)**. The function arguments are given by

- **testing_data** - The signal/background data to be evaluated.
- **testing_answer** - The corresponding class labels for the testing data.
- **score_output** - This option will evaluate a score and error function for the testing data if set to True. It is set to True by default.

The array that is returned by this function contains tuples of the network output and the actual class label specified by **testing_answer** which are something like $[[0.5834, 1.0], [-.34, -1.0], [0.00134, 1.0], \ldots]$. Some analysts are interested in getting predictions from the network, but for this particular piece of software we just need the output of the network which is a statistic that is hopefully sufficient. We can take these results and split them up according to their signal background class labels and then pass them through the JSD calculation to compare with the input JSD value.

6.0.10 split_binary_results

This function takes the output from the **evaluate_network** function and splits it into separate signal and background arrays.

6.0.11 network_output_jsd

This function does some handling with the data to prepare it for a mutual information calculation and then returns the mutual information. The function header is given by; **network_output_jsd(self, signal, background, neighbors=3)** where the individual arguments are given by

- **signal** - The signal distribution from the network output.
- **background** - The background distribution from the network output.
- **neighbors** - The number of nearest neighbors to be used in the mutual information calculation.

The function returns the JSD for these signal and background distributions.

6.0.12 plot_network_output

This function takes the output from the **evaluate_network** function and generates several plots. The function header is given by **plot_network_output(self, results, save_cuts=True)**. The function arguments are given as

- **results** - This is a tuple that contains the neural network output value and the class label. This is usually given as the output of the **evaluate_network** function.
- **save_cuts** - This option allows one to save the cut values from the acceptance and rejection cuts to a file.

There are three plots generated by this function. The first is a histogram of the outputs for signal and background which looks something like the following;

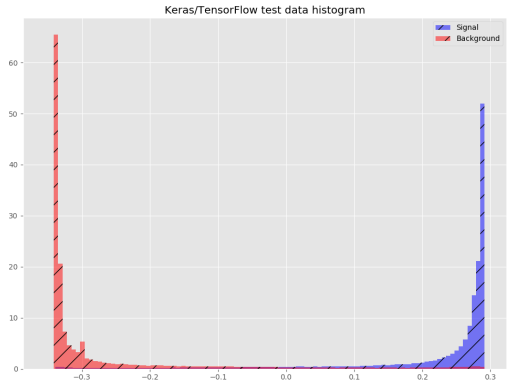

Figure 1: Example of network output histogram for signal and background.

This gives the user an idea of the amount of optimization obtained by the network by showing the amount of overlap in the two output distributions. The function also passes the results to the function **acceptance_rejection_cuts** which returns four arrays of values **signal acceptance**, **background acceptance** and their reciprocal values. From these we can construct the two ROC (receiver operator characteristic) curves **background rejection vs. signal acceptance** and **background acceptance vs. signal acceptance** which we show in the following examples.

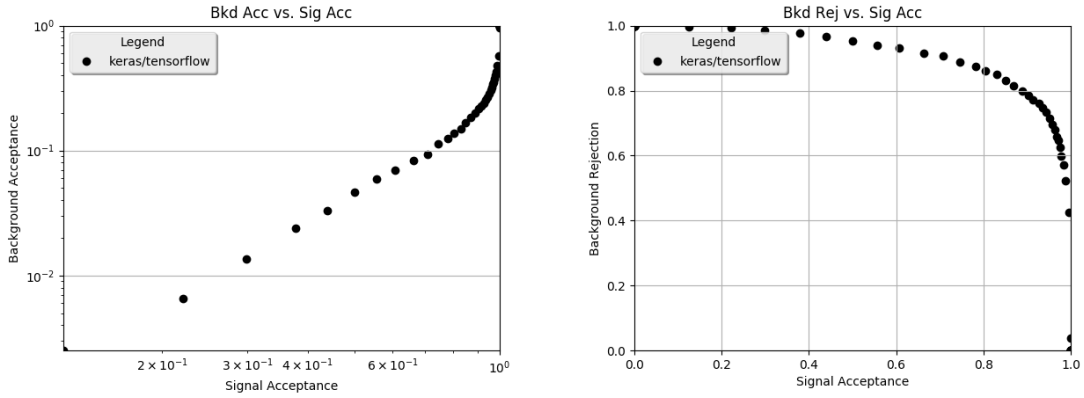

Figure 2: Example ROC curves background rejection vs. signal acceptance and background acceptance vs. signal acceptance.

6.0.13 save_network_score_to_file

This function saves the network score value $y = f(x)$ along with the corresponding values of x in a csv file. The function header is given by; **save_network_score_to_file(self, data, results, file)**, where the individual arguments are given by;

- **data** - This is the actual set of data points x that were used to generate the network score.
- **results** - This is the output from the network evaluated on the data points x given by **data**.
- **file** - The name of the file that the user wants to save the information to.

This function allows the user to analyze the relationship between the variable space and its projection into the statistic space $y = f(x)$.

7 Built In Examples

There is one built in example in the main part of the code which uses the SUSY data from this paper[**?**]. The data can be found in its original form at the UC Irvine repository[3]. The data included with the code is split into different signal and background files. This section will walk through the steps taken in this example. Simpler examples are given in the examples.py script that is also included with this code.

7.1 SUSY Example

This example looks at a set of low-level variables from a SUSY search[**?**] and calculates the JSD before the sample is sent through a neural network, and compares it with the JSD of the neural network output once it has been sufficiently optimized. While it is not entirely necessary, the example code here breaks up the SUSY data into ten subsets to get some statistics on the calculation of MI and the performance of the network. This breaking up of the data requires a lot more lines of code than would be necessary for a single comparison. In any case, the first thing that we do is import the data by calling

```
542 train_x, train_y, test_x, test_y = create_feature_sets_and_labels( "SUSYSignal.csv",
543                                                                    "SUSYBackground.csv",
544                                                                    num_of_vars=8,
545                                                                    test_size=0.001,
546                                                                    file_size=0.999,
547                                                                    var_set=[0,1,2,3,4,5,6,7] )
```

We've specified the signal and background files to be "**SUSYSignal.csv**" and "**SUSYBackground.csv**" respectively. We want to take essentially the entire file, so we use the argument **file_size=0.999** but only set the testing size to **test_size=0.001**. This is because we will specify the testing size later since we are breaking up the data into ten subsets. The variables we want are the eight low-level ones which are given by columns **var_set=[0,1,2,3,4,5,6,7]**.

The next set of lines define our ranges for the ten subsets to be defined

```
548 num_points = len(train_x)
549 skip_val = len(train_x) / 10
550 starting_val = 0
551 ending_val = skip_val - 1
552 JSD_list = list()
553 AUC_list = list()
```

This just keeps track of where the different subsets start and end. We also initialize a list that will keep track of the different JSD and AUC values for each subset as we go through them. We then define our loop, which is the following

```
557 for j in range(0,10):
558     temp_scores = list()
559     layer_vector = [8, 14, 7, 1]
560     network = nnet( layer_vector, learning_rate=0.05, decay_value=1e-5 )
561     temp_train_x = [train_x[i] for i in range(starting_val,ending_val)]
562     temp_train_y = [train_y[i] for i in range(starting_val,ending_val)]
563     #   Now separate the training and testing by a 70 / 30 random split
564     testing_ratio = int( .3 * skip_val )
565     new_train_x = list( temp_train_x[:-testing_ratio] )
566     new_train_y = list( temp_train_y[:-testing_ratio] )
567     new_test_x = list( temp_train_x[-testing_ratio:] )
568     new_test_y = list( temp_train_y[-testing_ratio:] )
569     #   Now train the network on the training set
570     network.train_network( new_train_x, new_train_y, num_epochs=100, batch=100 )
571     #   Then, evaluate the network on the testing results
572     results = network.evaluate_network( new_test_x, new_test_y )
573     #   We must break up the testing results into separate signal/background
574     #   so that we can calculate the mutual information
575     signal, background = network.split_binary_results( results )
576     temp_train_y = [[temp_train_y[i]] for i in range(len(temp_train_y))]
577     #   Getting the area under curve for background rejection versus signal efficiency
578     sig_eff, sig_rej, bkd_eff, bkd_rej = acceptance_rejection_cuts(signal, background)
579     AUC = network.area_under_ROC_curve(sig_eff,bkd_rej)
580     #   Now calculating the before and after mutual information
581     mutual = mi(network.normalize_data(temp_train_x),temp_train_y,k=1)
582     new_mutual = network.network_output_jsd(signal, background, neighbors=1)
583     #   Now we save all the values we've calculated for this set of sample data
584     temp_scores.append(mutual)
585     temp_scores.append(new_mutual)
586     JSD_list.append(temp_scores)
587     AUC_list.append(AUC)
588     print "trial:     ", j
589     print "MI before:", mutual
590     print "MI after: ", new_mutual
591     print "AUC:       ", AUC
592     starting_val += skip_val
593     ending_val += skip_val
```

We loop through all ten subsets of the data and split up the training and testing according to **testing_ratio** on line 564. The JSD on the input variables is calculated using the training samples, while the JSD on the network output comes from the testing samples. Line 559 defines **layer_vector = [8,14,7,1]** which is the network topology we decide to use here. Lines 561-568 define the training and testing subsets for each

iteration. Line 570 trains the network on the training subset for 100 epochs using a batch size of 100. The results are then given by the network looking at the testing data in line 572.

We then split up the results into individual signal and background arrays in line 575 which are then passed to the **acceptance_rejection_cuts** function, in line 578, that returns the four CDF's and inverse CDF's. Then the area under curve is calculated on the signal acceptance and background rejection pair of CDF values in line 579. We then calculate the before and after JSD from the discriminating variables and the network output in lines 581 and 582. The results are printed after each subset is evaluated and then saved into the lists **JSD_list** and **AUC_list** to keep track of them. After all ten subsets are gone through we run the last set of commands

```
594 #   Now take the mean and standard deviation of the JSD's and AUC's and print them to screen
595 before_list = [JSD_list[i][0] for i in range(len(JSD_list))]
596 after_list = [JSD_list[i][1] for i in range(len(JSD_list))]
597 before_mean = sum(before_list) / 10
598 after_mean = sum(after_list) / 10
599
600 before_std = np.std(np.array(before_list))
601 after_std = np.std(np.array(after_list))
602
603 AUC_mean = sum(AUC_list) / 10
604 AUC_std = np.std(np.array(AUC_list))
605
606 print "MI before mean:      ",before_mean
607 print "MI before std:       ",before_std
608 print "MI after mean:       ",after_mean
609 print "MI after std:        ",after_std
610 print "eff/rej AUC mean:    ",AUC_mean
611 print "eff/rej AUC std:     ",AUC_std
```

This calculates out the means and standard deviations of both lists and prints the results out to the screen.

7.2 Gaussian Example

The Gaussian example is given in a separate script called gauss_example.py. It is similar to the SUSY example in that it breaks up the data into ten subsets and trains a network on each of them individually. The main body of the loop is identical to lines 557-593 of the SUSY example, the difference being the preparation of the data and some plotting that happens afterwards. The beginning section of the data is the following

```
10 #   We make a set of 5D gaussians using numpy
11 signal_mu = 1.0
12 background_mu = -1.0
13 sigma = 1.0
14 num_samples = 100000
15
16 signal = [[np.random.normal(signal_mu,sigma,1)[0],
17            np.random.normal(signal_mu,sigma,1)[0],
18            np.random.normal(signal_mu,sigma,1)[0],
19            np.random.normal(signal_mu,sigma,1)[0],
20            np.random.normal(signal_mu,sigma,1)[0]] for i in range(num_samples)]
21
22 background = [[np.random.normal(background_mu,sigma,1)[0],
23                np.random.normal(background_mu,sigma,1)[0],
24                np.random.normal(background_mu,sigma,1)[0],
25                np.random.normal(background_mu,sigma,1)[0],
26                np.random.normal(background_mu,sigma,1)[0]] for i in range(num_samples)]
27
28 #   Now we'll make some redundant variables
29 signal_6 = [[np.exp(signal[i][1] + signal[i][2])] for i in range(num_samples)]
30 signal_7 = [[signal[i][1] + signal[i][2]] for i in range(num_samples)]
```

```
31 background_6 = [[np.exp(background[i][1] + background[i][2])] for i in range(num_samples)]
32 background_7 = [[background[i][1] + background[i][2]] for i in range(num_samples)]
33 #   And then add them to the lists
34 signal = np.concatenate((signal,signal_6,signal_7),axis=1).tolist()
35 background = np.concatenate((background,background_6,background_7),axis=1).tolist()
36
37 data = []
38 answer = []
39 for i in range(len(signal)):
40     data.append(signal[i])
41     answer.append(1.0)
42     data.append(background[i])
43     answer.append(-1.0)
```

The Gaussians in this example are five-dimensional with means either $\mu_s = 1.0$ or $\mu_b = -1.0$ for signal or background respectively. The variances are all set to $\sigma = 1.0$. Some additional redundant variables are created which are simply functions of variables two and three; $x_6 = \exp[x_2 + x_3]$ and $x_7 = x_2 + x_3$. The point of this exercise is to show that the redundant variables do not add any discriminating power between the signal and background distributions. The code will iterate through each variable, adding them one at a time and keeping track of their mutual informations. The next chunk of code that does this is the following.

```
47 num_vars = []
48 sample_size = int(num_samples/10)
49 Total_JSD_list = list()
50 for t in range(len(data[0])):
51     JSD_list = list()
52     starting_val = 0
53     ending_val = sample_size - 1
54     #   Here we pick the variables one at a time in succession
55     num_vars.append(t)
56     temp_data = [[data[j][l] for l in num_vars] for j in range(len(data))]
```

This loops through all seven variables adding them one at a time to the active data set. After this is done we then go through the main loop for each set of variables breaking up the data into ten subsets as to get some statistics. The main body of the loop is identical to the SUSY example except for the number of training epochs being changed to 25 and the **layer_vector** being adjusted to the appropriate number of input variables for each iteration. The differences come after the main body of the loop which is given by the lines;

```
103     Total_JSD_list.append([before_mean,before_std,after_mean,after_std])
104
105
106 Num_of_vars = [1,2,3,4,5,6,7]
107 Variable_means = [Total_JSD_list[i][0] for i in range(len(Total_JSD_list))]
108 Variable_errors = [Total_JSD_list[i][1] for i in range(len(Total_JSD_list))]
109 Network_means = [Total_JSD_list[i][2] for i in range(len(Total_JSD_list))]
110 Network_errors = [Total_JSD_list[i][3] for i in range(len(Total_JSD_list))]
111
112 plt.figure()
113 plt.plot(Num_of_vars, Variable_means, color='k', label='jsd_before k=1', linestyle='--')
114 plt.plot(Num_of_vars, Network_means, color='k', label='jsd_after k=1')
115 plt.errorbar(Num_of_vars, Variable_means, yerr=Variable_errors,fmt='',color='k',linestyle='--',ca
116 plt.errorbar(Num_of_vars, Network_means, yerr=Network_errors,fmt='',color='k',capsize=2)
117 plt.xticks(np.arange(0, 8, 1.0))
118 plt.grid()
119 plt.xlabel('Number of Variables')
120 plt.ylabel('JSD')
121 plt.legend( loc='lower right' )
122 plt.title('5D Gaussian + red. var. JSD vs. Num. of Var.')
123 plt.show()
```

For each variable that we add in succession, we keep track of the JSD means and standard deviations before and after the network. Then we plot the results which look something like the following.

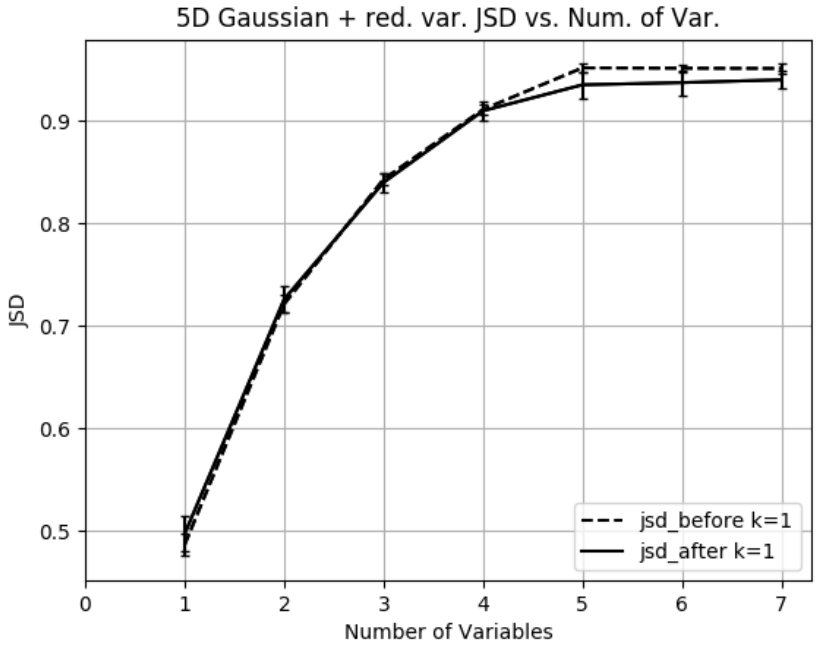

Figure 3: JSD vs. number of variables for the Gaussian example. The first five variables are Gaussians with $(\mu_s, \sigma_s) = (1.0, 1.0)$ for the signal and $(\mu_b, \sigma_b) = (-1.0, 1.0)$ for the background. The sixth and seventh variables are redundant and are given by $x_6 = \exp[x_2 + x_3]$ and $x_7 = x_2 + x_3$.

References

[1] N. Carrara and J. Ernst, *On the Upper Limit of Separability*

[2] N. Carrara and J. Ernst, *The Jensen-Shannon Divergence and Machine Learning Toolbox*

[3] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[4] A. Kraskov, H. Stögbauer, P. Grassberger, *Estimating mutual information*, Phys. Rev. E 69, 066138, (2004).

[5] L.F. Kozachenko, N.N. Leonenko, *A statistical estimate for the entropy of a random vector*, Problemy Peredachi Informatsii 23, (1987), 916.

[6] S. Delattre, N. Fournier, *On the Kozachenko-Leonenko entropy estimator*, arXiv:1602.07440,

[7] G. Ver Steeg, *NPEET: Nonparamateric entropy estimation toolbox*, `https://github.com/gregversteeg/NPEET`

[8] T.M. Cover, J.A. Thomas, *Elements of Information Theory*, Wiley, New York, 1991.

[9] G. van Rossum, **Python tutorial**, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica(CWI), Amsterdam, May 1995

[10] pip - Python Installation Package, `https://pip.pypa.io/en/stable/`

[11] M. Howell, **Homebrew**, `https://brew.sh/`

[12] Stfan van der Walt, S. Chris Colbert and Gal Varoquaux, **The NumPy Array: A Structure for Efficient Numerical Computation**, Computing in Science and Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37 (publisher link)

[13] Jones E, Oliphant E, Peterson P, et al. **SciPy: Open Source Scientific Tools for Python**, 2001-, http://www.scipy.org/ [Online; accessed 2017-05-11].

[14] John D. Hunter, **Matplotlib: A 2D Graphics Environment**, Computing in Science and Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55 (publisher link)

[15] Wes McKinney, **Data Structures for Statistical Computing in Python**, Proceedings of the 9th Python in Science Conference, 51-56 (2010) (publisher link)

[16] Fabian Pedregosa, Gal Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, douard Duchesnay, **Scikit-learn: Machine Learning in Python**, Journal of Machine Learning Research, 12, 2825-2830 (2011) (publisher link)

[17] Fernando Prez and Brian E. Granger, **IPython: A System for Interactive Scientific Computing**, Computing in Science and Engineering, 9, 21-29 (2007), DOI:10.1109/MCSE.2007.53 (publisher link)

[18] Franois Chollet, **Keras**, `https://github.com/fchollet/keras`

[19] Google Research, **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**, `http://tensorflow.org/`, Software available from tensorflow.org,

[20] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio, **Theano: A CPU and GPU Math Expression Compiler**, Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX

[21] J. Lin, *Divergence Measures Based on the Shannon Entropy*, IEEE Trans. on Info. The. Vol. 37, no.1, 1991.

[22] G. Crooks, *Measuring Thermodynamic Length*, arXiv:0706.0559v2

[23] A. Majtey, P.W. Lamberti, M.T. Martin and A. Plastino, *Wootters' distance revisited: a new distinguishability criterium*, arXiv:quant-ph/0408082v2 2004.

[24] A.P. Majtey, P.W. Lamberti and D.P. Prato, *Jensen-Shannon divergence as a measure of distinguishability between mixed quantum states*, arXiv:quant-ph/0508138v3 2005.

[25] F. Nielsen, *A family of statistical symmetric divergences based on Jensen's inequality*, arXiv:1009.4004v2 2011.