Anomaly Prediction

using a Neural Network implemented in Keras

List of Abbreviations

API	Application Programming Interface
Label	binary anomaly indicator [0/1]
PV	page views
STD	standard deviation
value	number of page views per minute
#	number (amount)

List of Contents

Pre-Processing and preliminary Considerations

Approach and Structure

The task we would like to tackle is a binary classification problem. In order to predict a binary value of several feature sets, the task is to set up and customize a neural network. The ultimate goal is to achieve a high area under the precision-recall curve on the training set, and a high F-Score, precision and recall on the testing set. To achieve this target, we will carry out a number of steps. The structure of this report provides a baseline of our procedure. At the beginning, pre-processing and feature selection has been conducted. The following figure displays important steps at the stage of preprocessing:

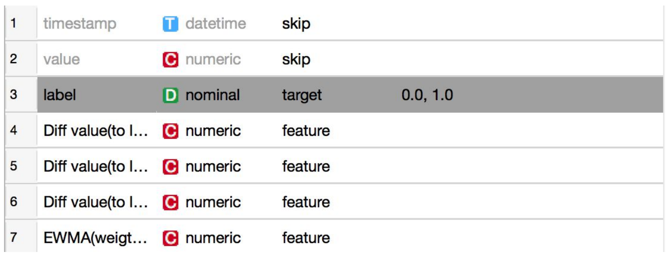
Ranking
Decision Tree
Data Preparation

The selection process is supported by building up a decision tree. This first steps are already leading the way since the accuracy of the network highly depends on the selected features. In the following, the data has to be pre-processed and selected according to the previous findings. Building upon these findings, a neural network shall be defined. The network will be improved and customized by methods like assemble learning and k-fold cross-validation. Moreover, the architecture of the neural network is of high significance. For constructing the neural network, we will employ the API Keras with a “Tensorflow” backend. Therefore, we mainly have to focus on a number of network characteristics like the number of layers, the chosen optimizers, the loss and activation functions and the epoch and batch size. Next, we will customize the network according to our needs and discuss different approaches. Finally, we critically assess the defined network with reference to the results and give a conclusion.

Feature Ranking

Feature Selection Method

Using the tool “Orange”, we analyzed the entire training data set. To read in the file, we skipped the value for “timestamp” and “page view value”. Next, we set “label” to be the target. In the following, we will detect the impact of the numerous features on the label. To do this, we will fall back on a number of statistical measures such Information Gain, the CHI2-test or the Gini coefficient.



1	timestamp	T datetime	skip	
2	value	C numeric	skip	
3	label	D nominal	target	0.0, 1.0
4	Diff value(to l…	C numeric	feature	
5	Diff value(to l…	C numeric	feature	
6	Diff value(to l…	C numeric	feature	
7	EWMA(weigt…	C numeric	feature	

Information Gain

Information gain provides us with comprehensive values telling which attribute carries the most useful information. Therefore, when trying to figure out the features that have the most impact on a target value or when setting up a decision tree, the attribute with the highest information gain shall be chosen as the root of the tree. Before calculating the information gain, a value of purity of a certain subset is required. Various statistical methods are in line for a measure of purity. For instance, Entropy impurity, Gini impurity or Misclassification impurity. In the following explanation, we will refer to Entropy impurity. The Entropy can be calculates as follows:

$$Entropy(N) = -\sum_j P(w_j)\log_2 P(w_j)$$

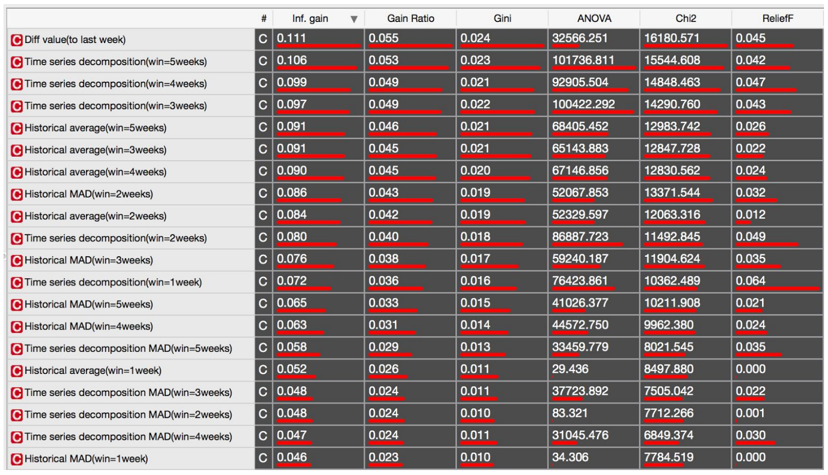
The final entropy value ranges between 0 and 1. For instance, a 1 bit entropy corresponds to a probability of 0.5. The entropy describes how many binary decisions have to be made to classify the information. To give an example, when trying to find 1 out of 2 entries, 1 bit is needed to determine either "found" or "not found", hence entropy is 1 while probability is 0.5.

Entropy is a prerequisite for determining the information gain. Information Gain will measure the change of impurity between different splits. The formula for Information Gain is given as follows:

$$Gain(S,A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

S stands for the original data set, whereas A depicts the attribute the split has been mode on. The formula brings out the expected reduction of entropy due to the split. From the formula, we can see that Entropy(X) generally ranges between 0 and 1, with 1 standing for very undetermined. 0 standing for clear and distinctive or on meta level, 0 means that zero binary decisions are needed to define to dataset, the dataset is pure already. The interpretation of Information Gain is vice-versa. Hence, a high information gain of 1 represents a high benefit. In the true sense of the word, it means a high gain of information which is good for our analysis. Information Gain cannot be negative, because when operating with data, we can only gain information but not loose.

For analyzing the training data, we employed the "Rank" tool and focused on the values for information gain. The observation yielded that for example the feature "Diff value(to last week)” and “Time series decomposition(win=5weeks)” significantly affect the binary value of label. This reasons our decision to use the mentioned features for building up the neural network.



	#	Inf. gain	Gain Ratio	Gini	ANOVA	Chi2	ReliefF
Diff value(to last week)	C	0.111	0.055	0.024	32566.251	16180.571	0.045
Time series decomposition(win=5weeks)	C	0.106	0.053	0.023	101736.811	15544.608	0.042
Time series decomposition(win=4weeks)	C	0.099	0.049	0.021	92905.504	14848.463	0.047
Time series decomposition(win=3weeks)	C	0.097	0.049	0.022	100422.292	14290.760	0.043
Historical average(win=5weeks)	C	0.091	0.046	0.021	68405.452	12983.742	0.026
Historical average(win=3weeks)	C	0.091	0.045	0.021	65143.883	12847.728	0.022
Historical average(win=4weeks)	C	0.090	0.045	0.020	67146.856	12830.562	0.024
Historical MAD(win=2weeks)	C	0.086	0.043	0.019	52067.853	13371.544	0.032
Historical average(win=2weeks)	C	0.084	0.042	0.019	52329.597	12063.316	0.012
Time series decomposition(win=2weeks)	C	0.080	0.040	0.018	86887.723	11492.845	0.049
Historical MAD(win=3weeks)	C	0.076	0.038	0.017	59240.187	11904.624	0.035
Time series decomposition(win=1week)	C	0.072	0.036	0.016	76423.861	10362.489	0.064
Historical MAD(win=5weeks)	C	0.065	0.033	0.015	41026.377	10211.908	0.021
Historical MAD(win=4weeks)	C	0.063	0.031	0.014	44572.750	9962.380	0.024
Time series decomposition MAD(win=5weeks)	C	0.058	0.029	0.013	33459.779	8021.545	0.035
Historical average(win=1week)	C	0.052	0.026	0.011	29.436	8497.880	0.000
Time series decomposition MAD(win=3weeks)	C	0.048	0.024	0.011	37723.892	7505.042	0.022
Time series decomposition MAD(win=2weeks)	C	0.048	0.024	0.010	83.321	7712.266	0.001
Time series decomposition MAD(win=4weeks)	C	0.047	0.024	0.011	31045.476	6849.374	0.030
Historical MAD(win=1week)	C	0.046	0.023	0.010	34.306	7784.519	0.000

Gain Ratio

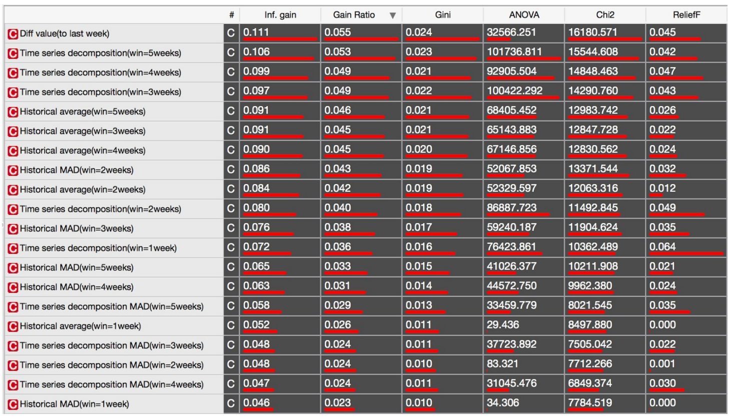
The next measure we took into consideration is the “Gain Ratio”. This measure reveals a similar ratio of factor dependencies like the information gain. However, the differences are more distinct and clear. Information gain ratio is a ratio of the information gain to the intrinsic information. It lowers down a bias that is caused by multi-valued attributes. For this reason, it is more distinct than information gain. Additionally to information gain, the intrinsic value has to be calculated using the following formula.

$$IV(Ex,a) = -\sum_{v \in values(a)} \frac{|\{x \in Ex | value(x,a) = v\}|}{|Ex|} \cdot \log_2\left(\frac{|\{x \in Ex | value(x,a) = v\}|}{|Ex|}\right)$$

Afterwards, the information gain is being divided by the intrinsic value. The result will be the gain ratio.

$$IGR(Ex,a) = IG/IV$$

The gain ratio generated the following results. In fact, the results are highly similar to the findings when applying “Information Gain”. Therefore, we do not have to particularly pay attention to the results.

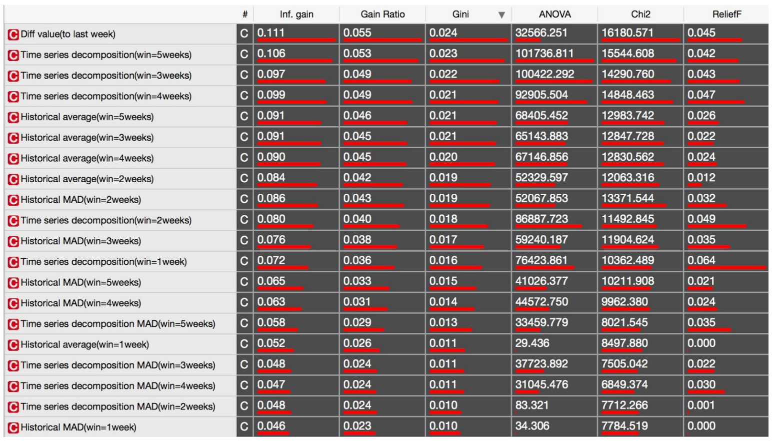


	#	Inf. gain	Gain Ratio ▼	Gini	ANOVA	Chi2	ReliefF
Diff value(to last week)	C	0.111	0.055	0.024	32566.251	16180.571	0.045
Time series decomposition(win=5weeks)	C	0.106	0.053	0.023	101736.811	15544.608	0.042
Time series decomposition(win=4weeks)	C	0.099	0.049	0.021	92905.504	14848.463	0.047
Time series decomposition(win=3weeks)	C	0.097	0.049	0.022	100422.292	14290.760	0.043
Historical average(win=5weeks)	C	0.091	0.046	0.021	68405.452	12983.742	0.026
Historical average(win=3weeks)	C	0.091	0.045	0.021	65143.883	12847.728	0.022
Historical average(win=4weeks)	C	0.090	0.045	0.020	67146.856	12830.562	0.024
Historical MAD(win=2weeks)	C	0.086	0.043	0.019	52067.853	13371.544	0.032
Historical average(win=2weeks)	C	0.084	0.042	0.019	52329.597	12063.316	0.012
Time series decomposition(win=2weeks)	C	0.080	0.040	0.018	86887.723	11492.845	0.049
Historical MAD(win=3weeks)	C	0.076	0.038	0.017	59240.187	11904.624	0.035
Time series decomposition(win=1week)	C	0.072	0.036	0.016	76423.861	10362.489	0.064
Historical MAD(win=5weeks)	C	0.065	0.033	0.015	41026.377	10211.908	0.021
Historical MAD(win=4weeks)	C	0.063	0.031	0.014	44572.750	9962.380	0.024
Time series decomposition MAD(win=5weeks)	C	0.058	0.029	0.013	33459.779	8021.545	0.035
Historical average(win=1week)	C	0.052	0.026	0.011	29.436	8497.880	0.000
Time series decomposition MAD(win=3weeks)	C	0.048	0.024	0.011	37723.892	7505.042	0.022
Time series decomposition MAD(win=2weeks)	C	0.048	0.024	0.010	83.321	7712.266	0.001
Time series decomposition MAD(win=4weeks)	C	0.047	0.024	0.011	31045.476	6849.374	0.030
Historical MAD(win=1week)	C	0.046	0.023	0.010	34.306	7784.519	0.000

Gini Coefficient

The Gini coefficient can be computed by taking the Lorenz curve into considerations. Namely, it is area between the diagonal and the Lorenz curve divided by the whole triangle. In mathematical terms, it would be G = A / (A + B). The Lorenz curve plots the cumulative value on the y axis. The x axis represents the number of units holding the value displayed on the y axis. The Gini coefficient ranges from 0 to almost 1, where 0 means total equality and a high number stands for complete inequality.

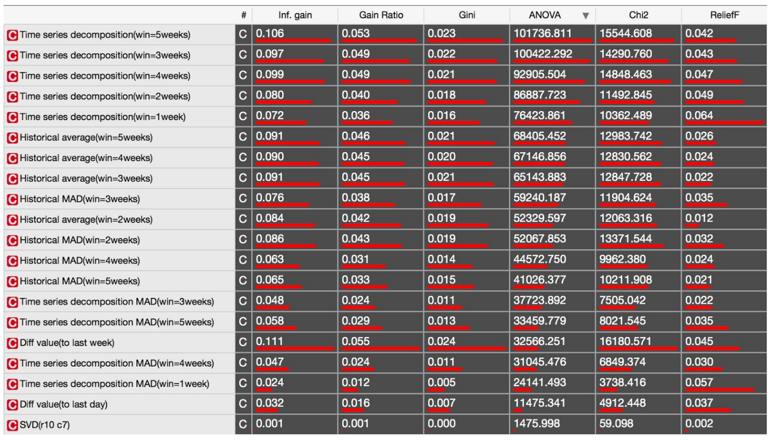
When applying Gini to the training data, setting label as the target, the results below are generated. The “Diff value(to last week)” is leading, followed by the different “Time series decomposition” and the “Historical Average”. Based on this findings, those features are preferably chosen.



	#	Inf. gain	Gain Ratio	Gini ▼	ANOVA	Chi2	ReliefF
Diff value(to last week)	C	0.111	0.055	0.024	32566.251	16180.571	0.045
Time series decomposition(win=5weeks)	C	0.106	0.053	0.023	101736.811	15544.608	0.042
Time series decomposition(win=3weeks)	C	0.097	0.049	0.022	100422.292	14290.760	0.043
Time series decomposition(win=4weeks)	C	0.099	0.049	0.021	92905.504	14848.463	0.047
Historical average(win=5weeks)	C	0.091	0.046	0.021	68405.452	12983.742	0.026
Historical average(win=3weeks)	C	0.091	0.045	0.021	65143.883	12847.728	0.022
Historical average(win=4weeks)	C	0.090	0.045	0.020	67146.856	12830.562	0.024
Historical average(win=2weeks)	C	0.084	0.042	0.019	52329.597	12063.316	0.012
Historical MAD(win=2weeks)	C	0.086	0.043	0.019	52067.853	13371.544	0.032
Time series decomposition(win=2weeks)	C	0.080	0.040	0.018	86887.723	11492.845	0.049
Historical MAD(win=3weeks)	C	0.076	0.038	0.017	59240.187	11904.624	0.035
Time series decomposition(win=1week)	C	0.072	0.036	0.016	76423.861	10362.489	0.064
Historical MAD(win=5weeks)	C	0.065	0.033	0.015	41026.377	10211.908	0.021
Historical MAD(win=4weeks)	C	0.063	0.031	0.014	44572.750	9962.380	0.024
Time series decomposition MAD(win=5weeks)	C	0.058	0.029	0.013	33459.779	8021.545	0.035
Historical average(win=1week)	C	0.052	0.026	0.011	29.436	8497.880	0.000
Time series decomposition MAD(win=3weeks)	C	0.048	0.024	0.011	37723.892	7505.042	0.022
Time series decomposition MAD(win=4weeks)	C	0.047	0.024	0.011	31045.476	6849.374	0.030
Time series decomposition MAD(win=2weeks)	C	0.048	0.024	0.010	83.321	7712.266	0.001
Historical MAD(win=1week)	C	0.046	0.023	0.010	34.306	7784.519	0.000

Anova

Another statistic measure we conducted is the so-called “ANOVA” test, standing for analysis of variance. In fact, it is combination of different methods to investigate the difference between differing groups. When performing an analysis of variance, the found variance is partitioned and classified to varying sources. Simply speaking, the measure generates a statistic of whether or not the means of several groups are equal, and therefore generalizes the t-test to more than two groups. This kind of procedure is particularly useful in our case because we are testing and comparing more than two features for statistical significance. The ANOVA test generated the following results.



	#	Inf. gain	Gain Ratio	Gini	ANOVA ▼	Chi2	ReliefF
Time series decomposition(win=5weeks)	C	0.106	0.053	0.023	101736.811	15544.608	0.042
Time series decomposition(win=3weeks)	C	0.097	0.049	0.022	100422.292	14290.760	0.043
Time series decomposition(win=4weeks)	C	0.099	0.049	0.021	92905.504	14848.463	0.047
Time series decomposition(win=2weeks)	C	0.080	0.040	0.018	86887.723	11492.845	0.049
Time series decomposition(win=1week)	C	0.072	0.036	0.016	76423.861	10362.489	0.064
Historical average(win=5weeks)	C	0.091	0.046	0.021	68405.452	12983.742	0.026
Historical average(win=4weeks)	C	0.090	0.045	0.020	67146.856	12830.562	0.024
Historical average(win=3weeks)	C	0.091	0.045	0.021	65143.883	12847.728	0.022
Historical MAD(win=3weeks)	C	0.076	0.038	0.017	59240.187	11904.624	0.035
Historical average(win=2weeks)	C	0.084	0.042	0.019	52329.597	12063.316	0.012
Historical MAD(win=2weeks)	C	0.086	0.043	0.019	52067.853	13371.544	0.032
Historical MAD(win=4weeks)	C	0.063	0.031	0.014	44572.750	9962.380	0.024
Historical MAD(win=5weeks)	C	0.065	0.033	0.015	41026.377	10211.908	0.021
Time series decomposition MAD(win=3weeks)	C	0.048	0.024	0.011	37723.892	7505.042	0.022
Time series decomposition MAD(win=5weeks)	C	0.058	0.029	0.013	33459.779	8021.545	0.035
Diff value(to last week)	C	0.111	0.055	0.024	32566.251	16180.571	0.045
Time series decomposition MAD(win=4weeks)	C	0.047	0.024	0.011	31045.476	6849.374	0.030
Time series decomposition MAD(win=1week)	C	0.024	0.012	0.005	24141.493	3738.416	0.057
Diff value(to last day)	C	0.032	0.016	0.007	11475.341	4912.448	0.037
SVD(r10 c7)	C	0.001	0.001	0.000	1475.998	59.098	0.002

Chi2

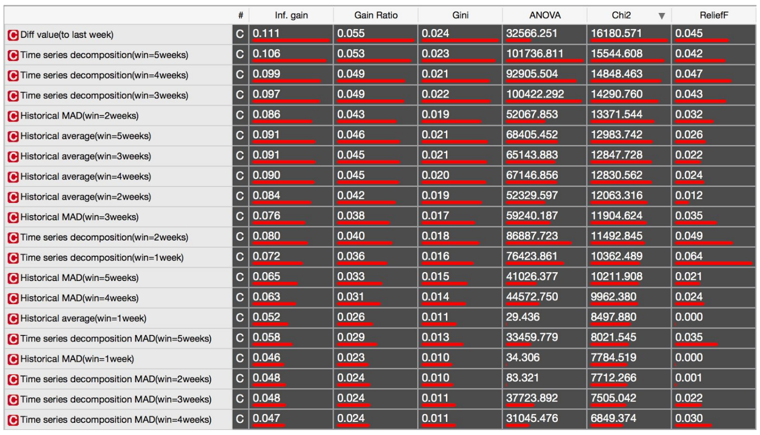
A prerequisite of conducting the Chi2 test, is normally distributed and independent data. The range of the Chi2 test is not predetermined.

$$\chi^2 = \sum_{i=1}^{k}\sum_{j=1}^{\ell}\frac{\left(h_{ij}-\tilde{h}_{ij}\right)^2}{\tilde{h}_{ij}} \quad \text{mit} \quad \tilde{h}_{ij} = \frac{h_{i\bullet}\cdot h_{\bullet j}}{n} \qquad \frac{(\text{observed} - \text{expected})^2}{\text{expected}}$$

Chi2 is calculated by computing the actual expected value of every “cell” of a table. The sum of these quantities over all of the cells is the test statistic. For a Chi2 Test, under the null hypothesis, it has approximately a chi-squared distribution whose number of degrees of freedom are:

$$(\text{number of rows} - 1)(\text{number of columns} - 1)$$

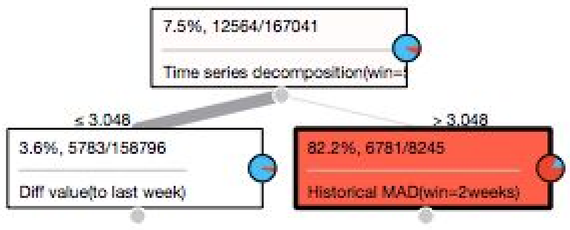
The following table outlines the influence of the different features on the target according to the CHI2 measure.Interestingly, we see the feature “Historical MAD(win=2 weeks) being more influential than in other measures.

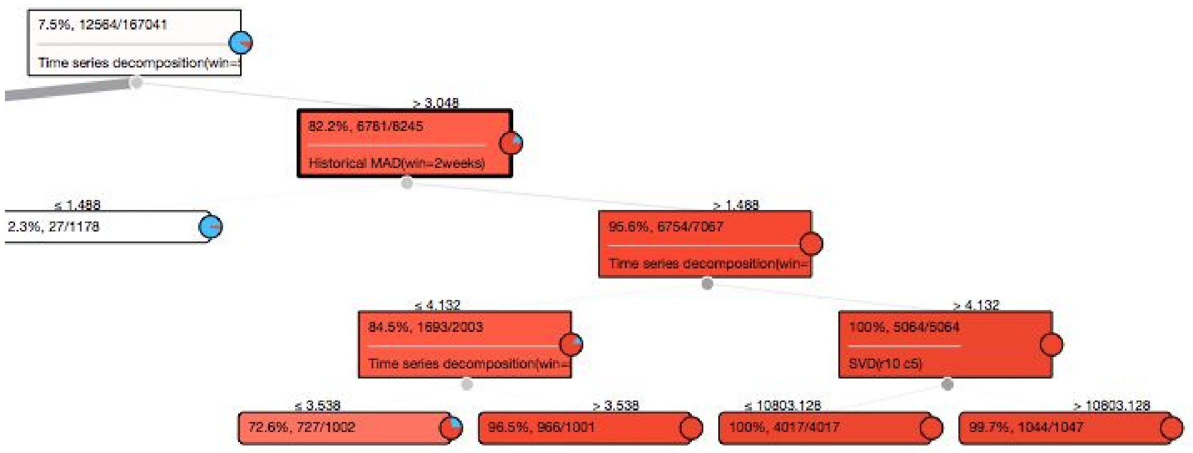


	#	Inf. gain	Gain Ratio	Gini	ANOVA	Chi2 ▼	ReliefF
Diff value(to last week)	C	0.111	0.055	0.024	32566.251	16180.571	0.045
Time series decomposition(win=5weeks)	C	0.106	0.053	0.023	101736.811	15544.608	0.042
Time series decomposition(win=4weeks)	C	0.099	0.049	0.021	92905.504	14848.463	0.047
Time series decomposition(win=3weeks)	C	0.097	0.049	0.022	100422.292	14290.760	0.043
Historical MAD(win=2weeks)	C	0.086	0.043	0.019	52067.853	13371.544	0.032
Historical average(win=5weeks)	C	0.091	0.046	0.021	68405.452	12983.742	0.026
Historical average(win=3weeks)	C	0.091	0.045	0.021	65143.883	12847.728	0.022
Historical average(win=4weeks)	C	0.090	0.045	0.020	67146.856	12830.562	0.024
Historical average(win=2weeks)	C	0.084	0.042	0.019	52329.597	12063.316	0.012
Historical MAD(win=3weeks)	C	0.076	0.038	0.017	59240.187	11904.624	0.035
Time series decomposition(win=2weeks)	C	0.080	0.040	0.018	86887.723	11492.845	0.049
Time series decomposition(win=1week)	C	0.072	0.036	0.016	76423.861	10362.489	0.064
Historical MAD(win=5weeks)	C	0.065	0.033	0.015	41026.377	10211.908	0.021
Historical MAD(win=4weeks)	C	0.063	0.031	0.014	44572.750	9962.380	0.024
Historical average(win=1week)	C	0.052	0.026	0.011	29.436	8497.880	0.000
Time series decomposition MAD(win=5weeks)	C	0.058	0.029	0.013	33459.779	8021.545	0.035
Historical MAD(win=1week)	C	0.046	0.023	0.010	34.306	7784.519	0.000
Time series decomposition MAD(win=2weeks)	C	0.048	0.024	0.010	83.321	7712.266	0.001
Time series decomposition MAD(win=3weeks)	C	0.048	0.024	0.011	37723.892	7505.042	0.022
Time series decomposition MAD(win=4weeks)	C	0.047	0.024	0.011	31045.476	6849.374	0.030

Decision Tree

For prediction and feature selection, decision trees can be a good tool. The general purpose of a decision tree is to support in learning the possibility of an event with determining combination of certain factors. In short, decision trees consist of a tree-like structure with every junction outlining a decision. The junctions are accompanied by the relative outcomes of choosing a certain branch. For instance, a possibility is given, as well as the specification on the splitting factor. Decision trees are a valuable and effective method when aiming at determining what specific combination of factors is most likely to lead to a certain outcome (in our case - the anomaly).
Below the root of the tree is provided. Since the red zone (the node with anomaly most likely to happen) is in the right leave, further we have considered only this part of the tree.



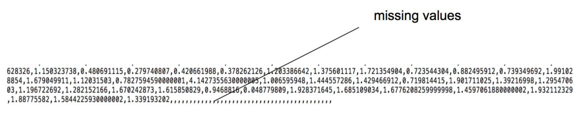


After examining the right node, we found out 8 features that could cause the highest probability of anomaly:

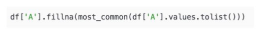
- Time series decomposition(win=5weeks),
- Historical MAD(win=2weeks),
- Time series decomposition(win=5weeks),
- Time series decomposition(win=1week),
- Diff value(to last week),
- Time series decomposition(win=4weeks),
- Time series decomposition(win=3weeks),
- SVD(r10 c5)

Data Preparation

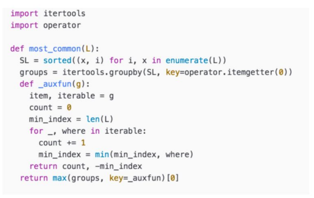
Unfortunately, in the test set, there were some values missing. For this reason we came up with a function to fill those gaps. The function “most_common” calculates the value that most often occurs in the column and returns it. Pandas function “fillna” can then be called to insert this found value to fill the gaps. This is an important step to ensure a working neural network.



```
628326,1.150323738,0.480691115,0.279740807,0.420661988,0.378262126,1.203386642,1.375601117,1.721354904,0.723544304,0.882495912,0.739349692,1.99102
8854,1.679049911,1.12031503,0.7827594590000001,4.1427355630000005,1.006595948,1.444557286,1.429466912,0.719814415,1.901711025,1.39216998,1.2954706
03,1.196722692,1.282152166,1.670242873,1.615850829,0.9468816,0.048779809,1.928371645,1.685109034,1.6776208259999998,1.4597061880000002,1.932112329
,1.88775582,1.5844225930000002,1.339193202,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```



```
df['A'].fillna(most_common(df['A'].values.tolist()))
```



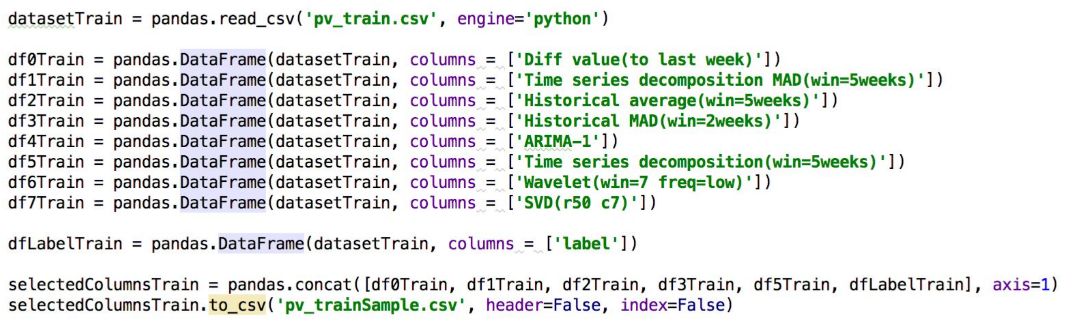
```
import itertools
import operator

def most_common(L):
  SL = sorted((x, i) for i, x in enumerate(L))
  groups = itertools.groupby(SL, key=operator.itemgetter(0))
  def _auxfun(g):
    item, iterable = g
    count = 0
    min_index = len(L)
    for _, where in iterable:
      count += 1
      min_index = min(min_index, where)
    return count, -min_index
  return max(groups, key=_auxfun)[0]
```

First Implications

Our research has identified a number of features to be particularly significant regarding the outcome of the label.

- Diff value(to last week)
- Time series decomposition MAD(win=5weeks)
- Historical average(win=5weeks)
- Historical MAD(win=2weeks)
- ARIMA-1
- Time series decomposition(win=5weeks)
- Wavelet(win=7 freq=low)
- SVD(r50 c7)

For this reason, we will use these features to set up the neural network. We were reading those features in and dropped other components of the data set. This kind of ore-processing was handled with the “Pandas” and the “Numpy” library.

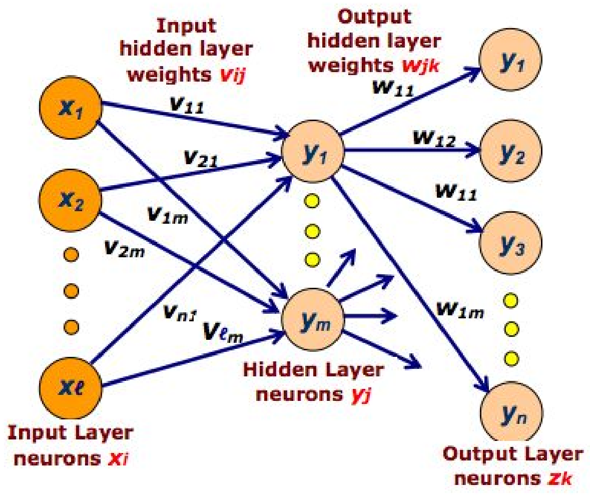


```
datasetTrain = pandas.read_csv('pv_train.csv', engine='python')

df0Train = pandas.DataFrame(datasetTrain, columns = ['Diff value(to last week)'])
df1Train = pandas.DataFrame(datasetTrain, columns = ['Time series decomposition MAD(win=5weeks)'])
df2Train = pandas.DataFrame(datasetTrain, columns = ['Historical average(win=5weeks)'])
df3Train = pandas.DataFrame(datasetTrain, columns = ['Historical MAD(win=2weeks)'])
df4Train = pandas.DataFrame(datasetTrain, columns = ['ARIMA-1'])
df5Train = pandas.DataFrame(datasetTrain, columns = ['Time series decomposition(win=5weeks)'])
df6Train = pandas.DataFrame(datasetTrain, columns = ['Wavelet(win=7 freq=low)'])
df7Train = pandas.DataFrame(datasetTrain, columns = ['SVD(r50 c7)'])

dfLabelTrain = pandas.DataFrame(datasetTrain, columns = ['label'])

selectedColumnsTrain = pandas.concat([df0Train, df1Train, df2Train, df3Train, df5Train, dfLabelTrain], axis=1)
selectedColumnsTrain.to_csv('pv_trainSample.csv', header=False, index=False)
```

Customizing and Tuning our Neural Network

Fundamental Neural Network Working Principles

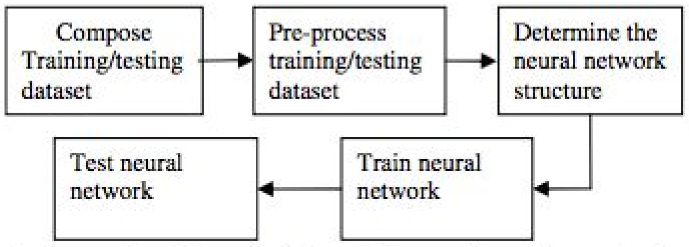
The basic idea behind a neural network is to simulate (copy in a simplified but reasonably faithful way) lots of densely interconnected brain cells inside a computer so you can get it to learn things, recognize patterns, and make decisions in a humanlike way. The amazing thing about a neural network is that you don't have to program it to learn explicitly: it learns all by itself, just like a brain.
A typical neural network has anything from a few dozen to hundreds, thousands, or even millions of artificial neurons called **units** arranged in a series of layers, each of which connects to the layers on either side (refer to the figure below).Some of them, known as **input units**, are designed to receive various forms of information from the outside world that the network will attempt to learn about, recognize, or otherwise process. Other units sit on the opposite side of the network and signal how it responds to the information it's learned; those are known as **output units**. In between the input units and output units are one or more layers of **hidden units**, which, together, form the majority of the artificial brain. Most neural networks are **fully connected**, which means each hidden unit and each output unit is connected to every unit in the layers either side. The connections between one unit and another are represented by a number called a **weight**, which can be either positive (if one unit excites another) or negative (if one unit suppresses or inhibits another). The higher the weight, the more influence one unit has on another.



- The hidden layer does intermediate computation before directing the input to output layer.
- The input layer neurons are linked to the hidden layer neurons; the weights on these links are referred to as input-hidden layer weights.
- The hidden layer neurons and the corresponding weights are referred to as output-hidden layer weights.

- A multi-layer network with *l* input neurons, *m1* neurons in the first hidden layers, and *n* output neurons in the output layers is (*l*-*m1*-*n*).

In order to build anomaly detection system using neural network, certain steps should be taken (Figure 1). First step is random collection of sessions from the given dataset to train and test neural network. Second step is preprocessing the collected data to a neural network readable format. Third step is determining the neural network structure, which is actually determining the number of hidden layers, number of hidden nodes in each layer,activation functions used in neural network and training algorithm. Fourth step is training neural network until a certain number of iterations or a certain RMSE value reached.Fifth and the final step is testing the neural network. The testing was conducted in three parts. In the preliminary experiment we just wanted to see when the neural network was properly trained to detect attacks and when it did not detect any attacks. The next experiment was done with a small amount of traffic, and in the end we conducted the final experiment where we used a higher amount of traffic. In these last two experiments we had normal traffic, known attacks and unknown attacks in three different files.


Compose Training/testing dataset
Pre-process training/testing dataset
Determine the neural network structure
Train neural network
Test neural network
Figure 1. Steps to be taken to build neural network based anomaly detection System

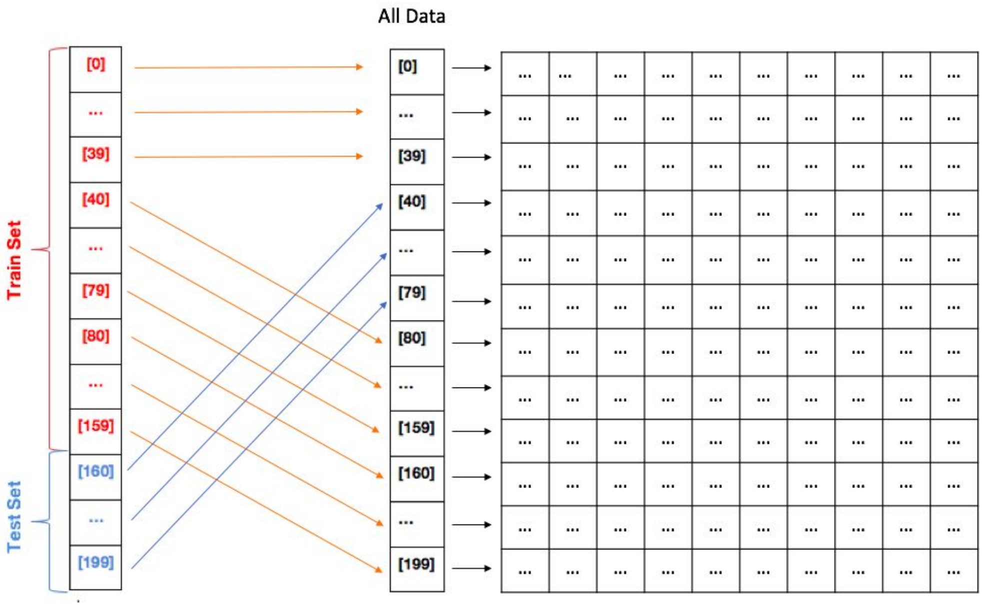
K-fold cross-validation

The mechanics of cross-validation are relatively simple, but the reasons why cross-validation is used with neural networks are a bit subtle. The ultimate goal of the classification problem is to find a set of neural network weights and bias values so that the input data generates output values that best match the target values. A simplistic approach would be to use all of the available data items to train the neural network. However, this approach would likely find weights and bias values that match the data extremely well (probably with 100 percent accuracy), but when presented with a new, previously unseen set of input data, the neural network would likely predict very poorly. This phenomenon is called overfitting. To avoid overfitting, the idea is to separate the available data into a training data set (typically 80 percent to 90 percent of the data) that's used to find a set of good weights and bias values, and a test set (the remaining 10 percent to 20 percent of the data) that is used to evaluate the quality of resulting neural network.
The simplest form of cross-validation randomly separates the available data into a single training set and a single test set (i.e. hold-out validation). But the hold-out approach is

somewhat risky because an unlucky split of the available data could lead to an ineffective neural network. One possibility is to repeat hold-out validation several times. This is called repeated sub-sampling validation. But this approach also entails some risk because, although unlikely, some data items could be used only for training and never for testing, or vice versa.

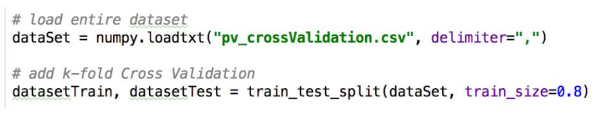
The idea behind k-fold cross-validation is to divide all the available data items into roughly equal-sized sets. Each set is used exactly once as the test set while the remaining data is used as the training set. The matrix (implemented as an array of arrays) labeled All Data has 200 data items. The number of folds has been set to 10. Each subset of data has 200/10 = 20 items. So subset [0] data ranges from indices [0] to [19], subset [1] ranges from indices [20] to [39], and the last subset [9] ranges from [180] to [199].

The k-fold cross-validation process iterates over the number of folds. For fold k=0, the 40-item test set would be data items [0] through [39], and the 160-item training set would be data items [40] through [199]. For fold k=1, (the scenario we illustrated on our example in the figure below), the test set would be data items [40] through [79] and the training set would be data items [0] through [39] and also [80] though [199]. Finally, for the last fold k=9, the test set would be items [160] through [199] and the training set would be items [0] through [159].

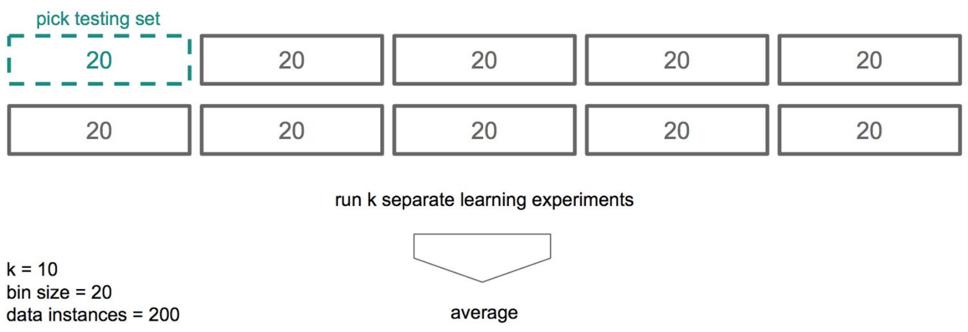


For efficiency the training and test data sets should be implemented as references to the data source in memory, rather than duplicating the data values. Also, the our case here is fortunate because the number of folds (10) divides evenly into the number of available data items (200). In situations where the number of folds doesn't divide evenly, the last data subset picks up the extra data items. For example, if the number of folds had been set to 11, the first three subsets would contain 200/11 = 18 data items and the last subset would contain the remaining 2 data items.

Please, see below the implementation of cross-validation to our data.



```
# load entire dataset
dataSet = numpy.loadtxt("pv_crossValidation.csv", delimiter=",")

# add k-fold Cross Validation
datasetTrain, datasetTest = train_test_split(dataSet, train_size=0.8)
```



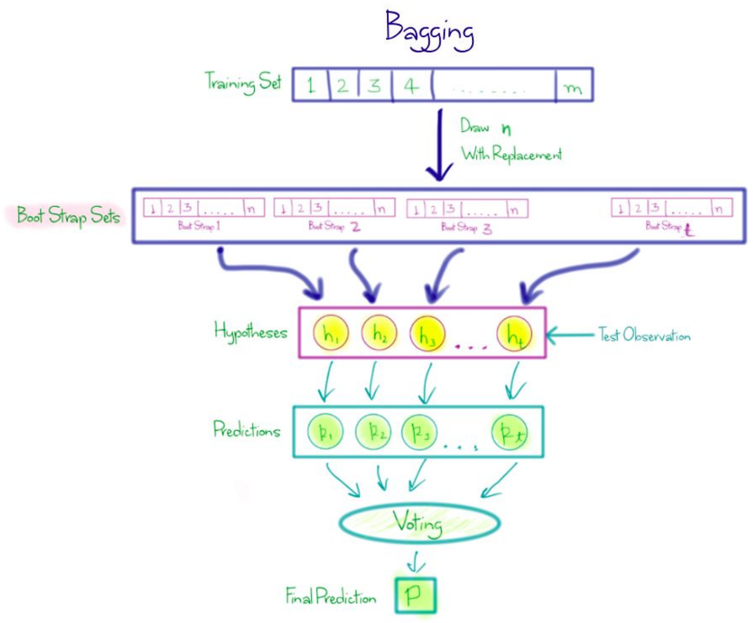
So, basically the idea is to break the training data into k subsets, where k is usually 10. Then you run your training algorithm (the three most common approaches are back-propagation, particle swarm optimization, and genetic algorithm optimization) 10 times. On the first training run you use the 9/10 of the training data to train, and then compute the network's accuracy using the 1/10 of the remaining data. This process is repeated, so that each 1/10 subset is used exactly once as the validation set. When finished you take the average of the 10 accuracies and use it as the overall estimate of the accuracy of the network. In short, k-fold cross-validation gives you an estimate of a neural network's accuracy when the network was constructed using particular values for number of hidden nodes and training parameters.
If you do k-fold cross-validation repeatedly, and during the training phase use different values for the training technique's parameters (different techniques have different parameters – back-prop needs learning rate and momentum, particle swarm needs inertia, cognitive and social weights, and so on) and also try different numbers of hidden nodes, you can find the best values for number of hidden nodes and training parameters. Then with these in hand you can finally train your network using all your data, with the best number of hidden nodes and training parameters.

Ensemble Learning

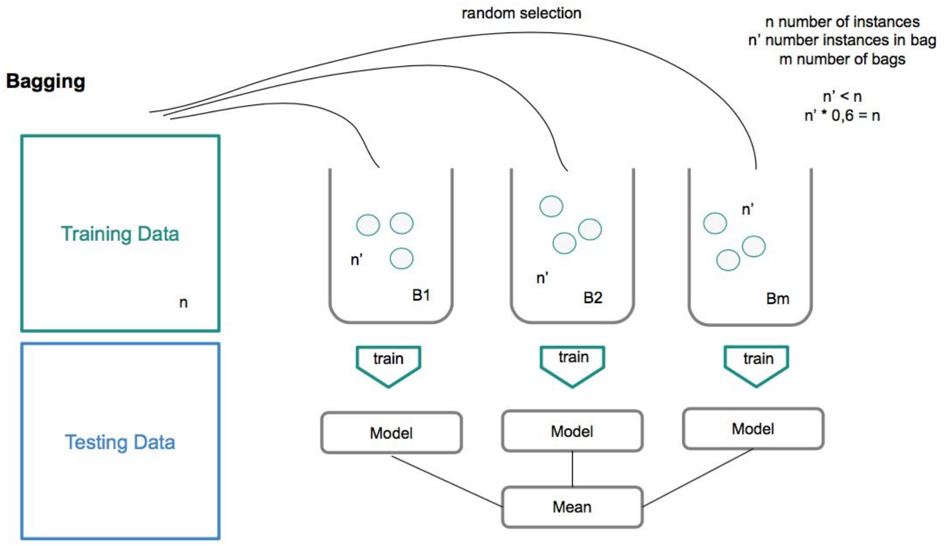
Bootstrap aggregating (Bagging)

Bootstrap aggregating (bagging) is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression.

It also reduces variance and helps to avoid over-fitting. Although it is usually applied to decision tree methods, it can be used with any type of method.

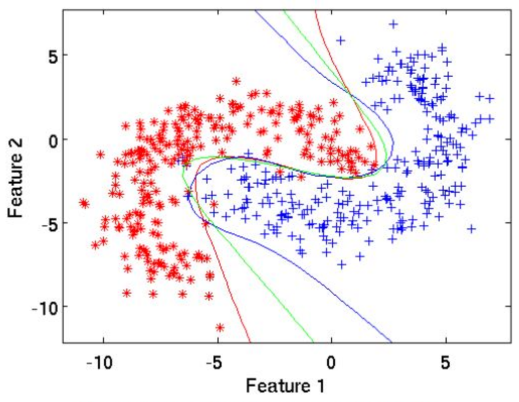


- Given:
  - Training set of N examples
  - A class of learning models (in our case neural network model)
- Method:
  - Train multiple (k) models on different samples (data splits) and average their predictions
  - Predict (test) by averaging the results of k models
- Goal:
  - Improve the accuracy of one model by using its multiple copies
  - Average of misclassification errors on different data splits gives a better estimate of the predictive ability of a learning method

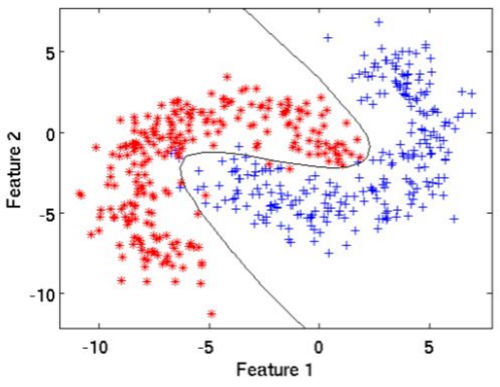


Assume we measure a random variable *x* with a *N(μ,σ2)* distribution. If only one measurement *x1* is done, the expected mean of the measurement is *μ* and variance is *Var(x1)=σ2*. If random variable *x* is measured *K* times $(x_1,x_2,\ldots x_k)$ and the value is estimated as $(x_1+x_2+\ldots+x_k)/K$. Mean of the estimate is still *μ*, but the variance is smaller now: $[Var(x_1)+\ldots Var(x_k)]/K^2=K\sigma^2 / K^2 = \sigma^2/K$.

Example



Three neural nets generated with default settings [bpxnc]

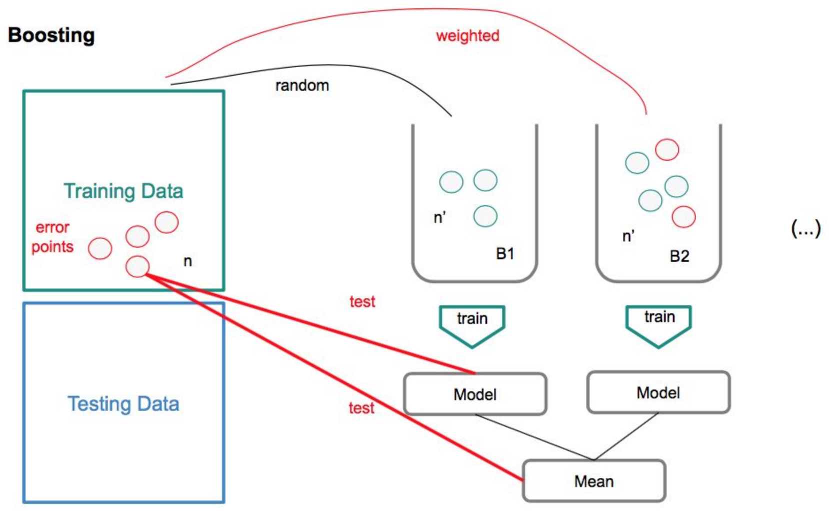


Final output from bagging 10 neural nets

Main property of Bagging is that it decreases variance of the base model without changing the bias due to averaging. So, bagging typically helps when applied with an over-fitted base model due to high dependency on actual training data. It does not help much with high bias (i.e. when the base model is robust to the changes in the training data).

Boosting

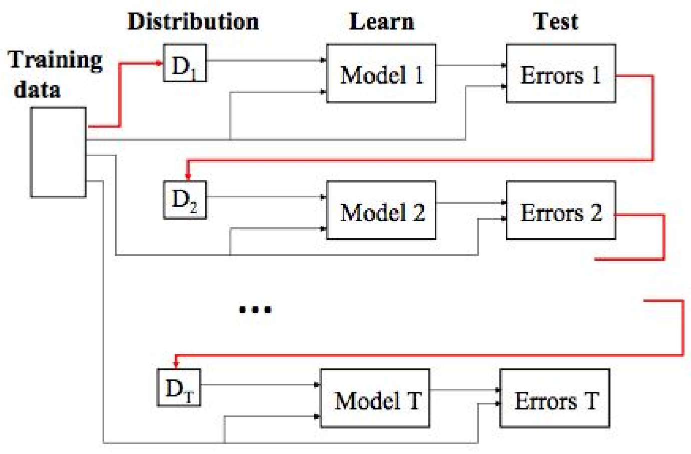
Boosting is a general method for improving the performance of learning algorithms. It is a method for finding a highly accurate classifier on the training set by combining weak hypotheses each of which needs only to be moderately accurate on the training set. As applied to an ensemble of neural networks using supervised learning, the algorithm proceeds as follows: Assume an oracle that generates a large number of independent training examples. First, generate a set of training examples and train a first network. After the first network is trained it may be used in combination with the oracle to produce a second training set in the following manner for anomaly detection (0 if normal, 1 if anomaly). If it is 1, pass outputs from the oracle through the first learning machine until the first network misclassifies a pattern and add this pattern to a second training set. Otherwise, pass outputs from the oracle through the first learning machine until the first network finds a pattern that it classifies correctly and add to the training set. This process is repeated until enough patterns have been collected. These patterns, half of which the first machine classifies correctly and half incorrectly, constitute the training set for the second network. The second network may then be trained.
The first two networks may then be used to produce a third training set in the following manner: Pass the outputs from the oracle through the first two networks. If the networks disagree on the classification, add this pattern to the training set. Otherwise, toss out the pattern. Continue this until enough patterns are generated to form the third training set. This third network is then trained. In the final testing phase, the test patterns (never previously used for training or validation) are passed through the three networks and labels assigned using the following voting scheme: If the first two networks agree, that is the label. Otherwise, assign the label as classified by the third network.

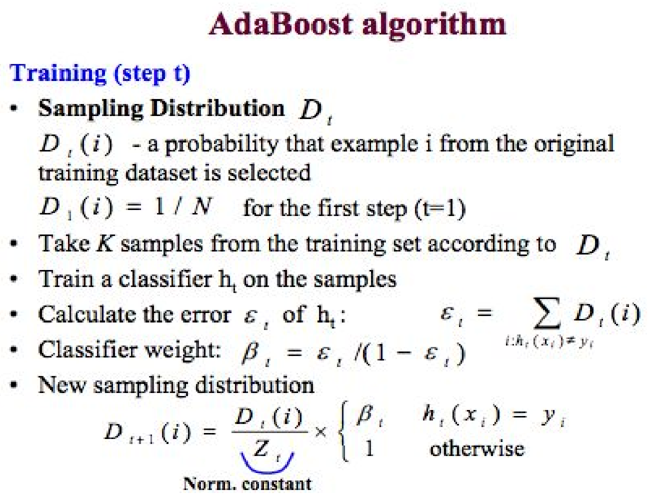


AdaBoost (boosting by sampling)

- Given:
  - A training set of N examples (attributes + class label pairs)
  - A “base” learning model (in our case a neural network)
- Training stage:
  - Train a sequence of T “base” models on T different sampling distributions defined upon the training set (D)
  - A sample distribution Dt for building the model t is constructed by modifying the sampling distribution Dt-1 from the (t-1)th step.
    - Examples classified incorrectly in the previous step receive higher weights in the new data (attempts to cover misclassified samples)

AdaBoost training





AdaBoost algorithm

Training (step t)

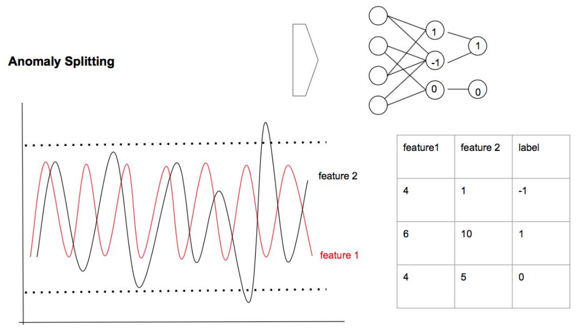
- **Sampling Distribution** D_t
  $D_t(i)$ - a probability that example i from the original training dataset is selected
  $D_1(i) = 1 / N$ for the first step (t=1)
- Take K samples from the training set according to D_t
- Train a classifier h_t on the samples
- Calculate the error ε_t of h_t: $\varepsilon_t = \sum_{i:h_t(x_i)\neq y_i} D_t(i)$
- Classifier weight: $\beta_t = \varepsilon_t/(1-\varepsilon_t)$
- New sampling distribution

$$D_{t+1}(i) = \frac{D_t(i)}{\underbrace{Z_t}_{\text{Norm. constant}}} \times \begin{cases} \beta_t & h_t(x_i) = y_i \\ 1 & \text{otherwise} \end{cases}$$

All in all, in boosting each classifier specializes on a particular subset of examples. Algorithm is concentrating on “more and more difficult” examples. Boosting can reduce variance (the same as Bagging), but also eliminate the effect of high bias of the weak learner (unlike Bagging).

Anomaly Splitting

Another concept we had in mind, was classifying the anomalies before operating the neural network. Currently, the anomalies are only divided into “1” and “0” with “1” standing for anomaly. Therefore, the network has to differentiate and learn that high values in a feature can lead to an anomaly but also very low values. This differentiation complicates the learning of the network. Therefore, we came up with the idea to further split anomalies into “high” and “low” anomaly and create a new label. That way we would end up having “-1” for low anomaly “0” for no anomaly and “1” for high anomaly. This approach could significantly simplify the training of the network and yield a higher f-value. The do this however, we have to analyse the data first and define thresholds for features.



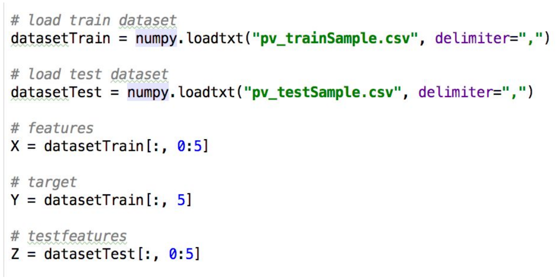
feature1	feature 2	label
4	1	-1
6	10	1
4	5	0

Sequential Neural Network and Prediction

Basic Network Architecture

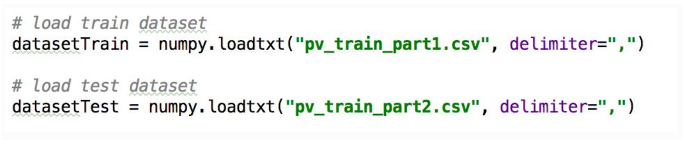
For implementing our neural network, we fell back on Keras. Keras is a high-level neural networks API that we used upon a Tensorflow backend. As a first step, we imported “keras.layers” to get access to the “Dense, Dropout” Layers. Moreover, we imported the “Sequential Model”. There are two types of models available in Keras: the Sequential model and the Model class used with functional API.

With support by the Numpy library, we loaded our pre-processed data samples which are the training and the testing data. The variable X is holding the features used for training, Y is the training targets. Z is the initialized with the testing data, used for prediction.

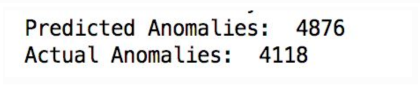


```
# load train dataset
datasetTrain = numpy.loadtxt("pv_trainSample.csv", delimiter=",")

# load test dataset
datasetTest = numpy.loadtxt("pv_testSample.csv", delimiter=",")

# features
X = datasetTrain[:, 0:5]

# target
Y = datasetTrain[:, 5]

# testfeatures
Z = datasetTest[:, 0:5]
```

We found it very helpful to train, customize and test the network using the training data only. Hence, we splitted the training data and used one half for training and one half for testing. This approach bered the advantage of exactly knowing the accuracy of the prediction because both data sets are labelled. We counted the number of “actual anomalies” counted in the data with the number of “predicted anomalies” calculated with the neural network.



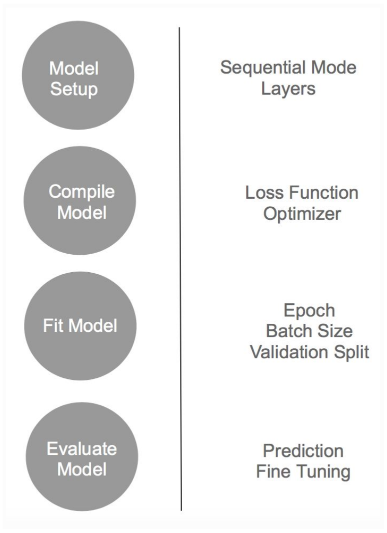
```
# load train dataset
datasetTrain = numpy.loadtxt("pv_train_part1.csv", delimiter=",")

# load test dataset
datasetTest = numpy.loadtxt("pv_train_part2.csv", delimiter=",")
```



```
Predicted Anomalies:  4876
Actual Anomalies:  4118
```

Another advantage of this approach is the avoidance of overfitting. Applying the whole training data set often resulted in an overfit with extremely high accuracy on the training data but low accuracy on the testing set. The problem of overfitting generally was one of the biggest issues.

In the following, we will discuss the process of building up the neural network and finding the right settings. The figure below, provides a basic framework.



Model Setup

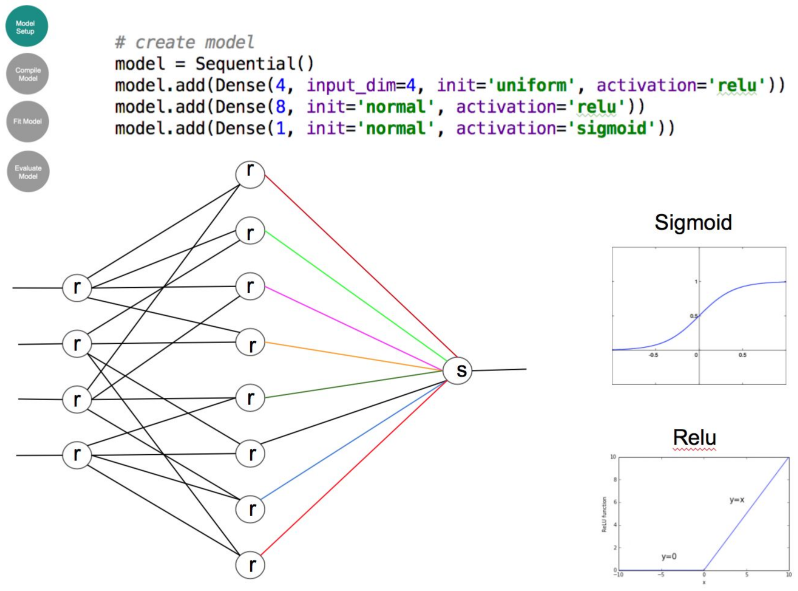
For setting up our neural network, we have chosen to make use of the Sequential Model. To keep things easy, we started from a very simple architecture with one input, one output layer and only one hidden layer in between. We noticed, that the initialization and the activation function is of very high importance. The network accuracy significantly drops when applying “uniform” initialization at the output layer. The difference between both kinds can be summarized as follows:

- uniform initializes all connection with the same initial weight.
- normal initializes the connections randomly according to a distribution function

The main purpose of finding the right initialization is to avoid both of the following cases and to achieve a trade-off between them:

- The weights are too small, and repeatedly multiplying with the weights dampens the signal so much that you essentially lose the input signal. If the input doesn’t flow through the network, then it becomes impossible to credit assignment (since a small change in the parameters still do not change the output).
- The norm of the weights is too big and repeatedly multiplying amplifies the signal too much such that it explodes. As we will see, this puts the network in a very bad position to start.

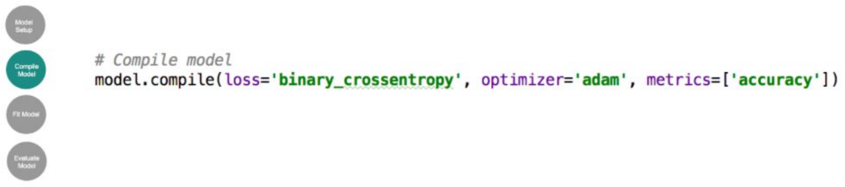
As important as the initialization is also the activation. In a neural network, the value that gets propagated is the product of weight multiplied with the actual informative numeric value. The activation function compresses and unifies the output of this multiplication. The two activation functions we applied are the rectified linear unit and the sigmoid function. Both functions are displayed below.



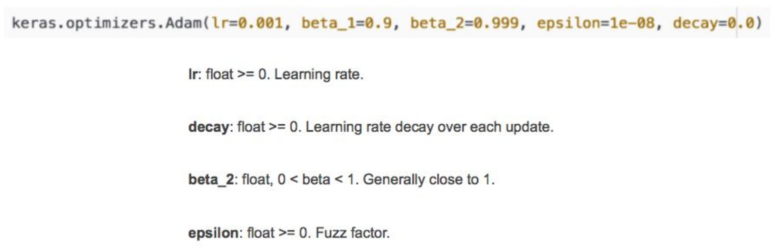
```
# create model
model = Sequential()
model.add(Dense(4, input_dim=4, init='uniform', activation='relu'))
model.add(Dense(8, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
```

Compile Model

All Keras models need to be compiled. As for compilation, there is a few parameters that have to be given. For our loss function, we have chosen to employ “binary_crossentropy”. This function is suggested when dealing with binary classification. The used optimizer is “adam”. All optimizers have characteristic specifications. For example, the learning rate provides information on how quick weights can change and adjust. We can also implement a decay on the learning rate to decrease the learning rate with every iteration. The metric the optimization is oriented on is the accuracy. A metric is a function that is used to judge the performance of your model. Metric functions are to be supplied in the metrics parameter when a model is compiled. A metric function is similar to an loss function, except that the results from evaluating a metric are not used when training the model.



```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```



```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
```

lr: float >= 0. Learning rate.

decay: float >= 0. Learning rate decay over each update.

beta_2: float, 0 < beta < 1. Generally close to 1.

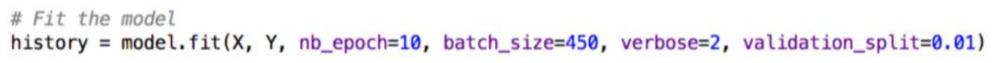
epsilon: float >= 0. Fuzz factor.

Fit Model

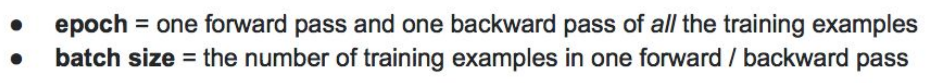
Since we have defined and compiled our model, it is time to execute the model on the data (i.e. adapt the weights on a training dataset).
We can train or fit our model on our loaded data by calling the fit() function on the model.

The training process runs for a fixed number of iterations through the dataset called epochs, that we specify using the nb_epoch argument. We also set the number of instances that are evaluated before a weight update in the network is performed, called the batch size and set using the batch_size argument.

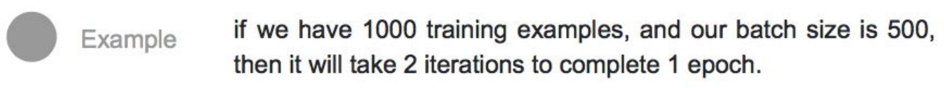
For this problem, we will run for a number of iterations equal to 10 and use a batch size of 450. These can be chosen experimentally by trial and error.



```
# Fit the model
history = model.fit(X, Y, nb_epoch=10, batch_size=450, verbose=2, validation_split=0.01)
```



- **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward / backward pass



Example if we have 1000 training examples, and our batch size is 500, then it will take 2 iterations to complete 1 epoch.

Fitting the network requires the training data to be specified, both a matrix of input patterns X and an array of matching output patterns y.

The network is trained using the backpropagation algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

The backpropagation algorithm requires that the network be trained for a specified number of epochs or exposures to the training dataset.

Each epoch can be partitioned into groups of input-output pattern pairs called batches. This define the number of patterns that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time.

```
# Fit the model
history = model.fit(X, Y, nb_epoch=10, batch_size=450, verbose=2, validation_split=0.01)
```

- **validation_split** = if validation_split is 0.1 (edit), the first 90% are selected as train data and the remaining 10% as validation data.

```
if 0 < validation_split < 1:
                do_validation = True
                split_at = int(len(ins[0]) * (1 - validation_split))
                (ins, val_ins) = (slice_X(ins, 0, split_at), slice_X(ins, split_at))
```

- **shuffle** = training data can optionally be shuffled after every epoch.

Here is the list of arguments that we have used in our fit() function:

- **X:** list of Numpy arrays taken from given train dataset
- **Y:** list of Numpy arrays taken from given train dataset
- **epochs:** integer, the number of times to iterate over the training data arrays.
- **batch_size:** integer. Number of samples per gradient update.
- **verbose:** 0, 1, or 2. Verbosity mode. 0 = silent, 1 = verbose, 2 = one log line per epoch.
- **validation_split:** float between 0 and 1: fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch.

Evaluate Model

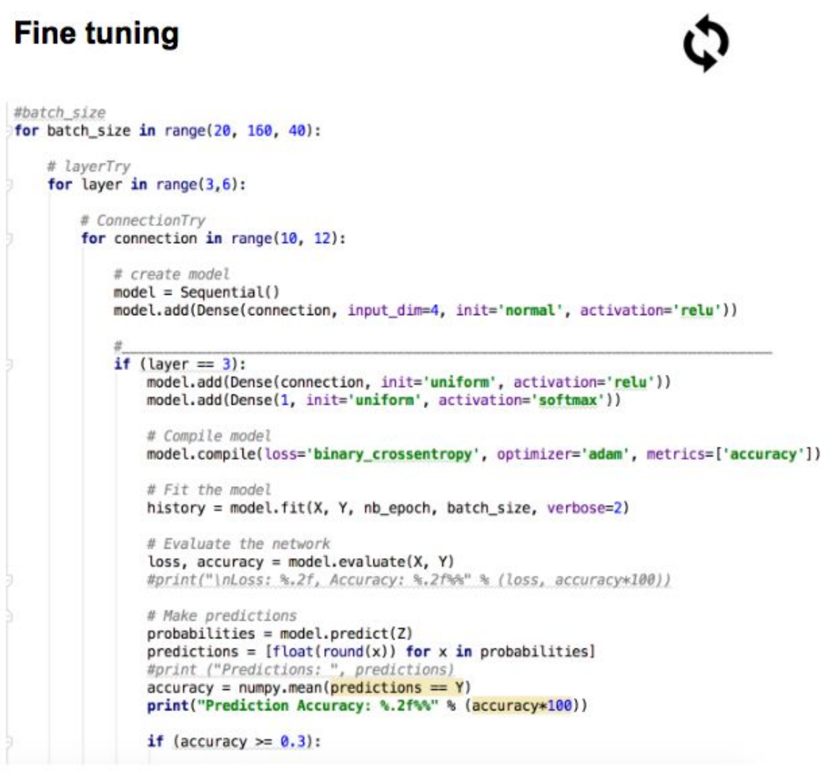
Once the network is trained, it can be evaluated.

The network can be evaluated on the training data, but this will not provide a useful indication of the performance of the network as a predictive model, as it has seen all of this data before.

We can evaluate the performance of the network on a separate dataset, unseen during testing. This will provide an estimate of the performance of the network at making predictions for unseen data in the future.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy. A list of evaluation metrics is returned.

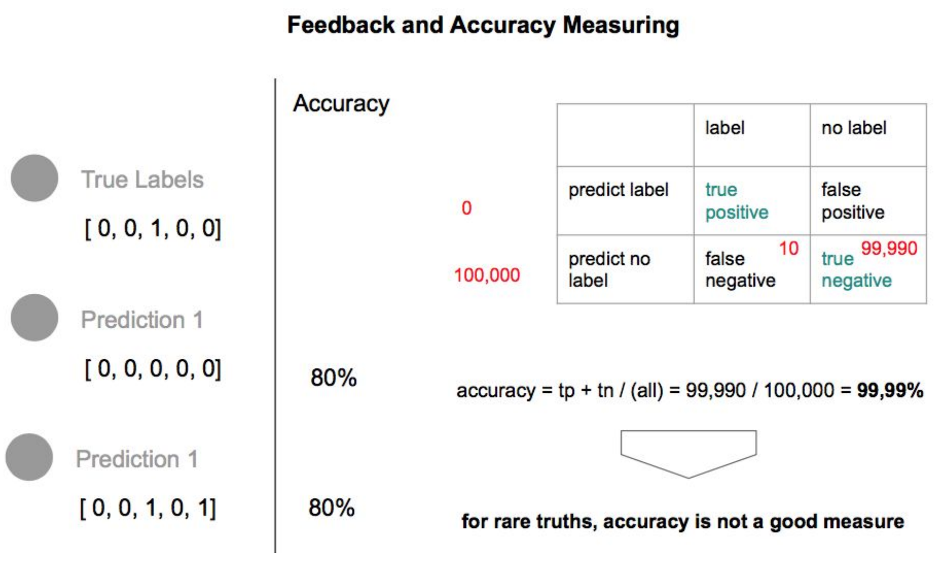
For example, for a model compiled with the accuracy metric, we could evaluate it on a new dataset as follows:



Fine tuning

```
#batch_size
for batch_size in range(20, 160, 40):

    # layerTry
    for layer in range(3,6):

        # ConnectionTry
        for connection in range(10, 12):

            # create model
            model = Sequential()
            model.add(Dense(connection, input_dim=4, init='normal', activation='relu'))

            #________________________________________________________________________
            if (layer == 3):
                model.add(Dense(connection, init='uniform', activation='relu'))
                model.add(Dense(1, init='uniform', activation='softmax'))

                # Compile model
                model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

                # Fit the model
                history = model.fit(X, Y, nb_epoch, batch_size, verbose=2)

                # Evaluate the network
                loss, accuracy = model.evaluate(X, Y)
                #print("\nLoss: %.2f, Accuracy: %.2f%%" % (loss, accuracy*100))

                # Make predictions
                probabilities = model.predict(Z)
                predictions = [float(round(x)) for x in probabilities]
                #print ("Predictions: ", predictions)
                accuracy = numpy.mean(predictions == Y)
                print("Prediction Accuracy: %.2f%%" % (accuracy*100))

                if (accuracy >= 0.3):
```

Finally, once we are satisfied with the performance of our fit model, we can use it to make predictions on new data.
This is as easy as calling the predict() function on the model with an array of new input patterns.
In our case -a binary classification problem- the predictions is an array of probabilities for the first class that can be converted to a 1 or 0 by rounding.



Feedback and Accuracy Measuring

True Labels

[0, 0, 1, 0, 0]

Prediction 1

[0, 0, 0, 0, 0]

Prediction 1

[0, 0, 1, 0, 1]

Accuracy

80%

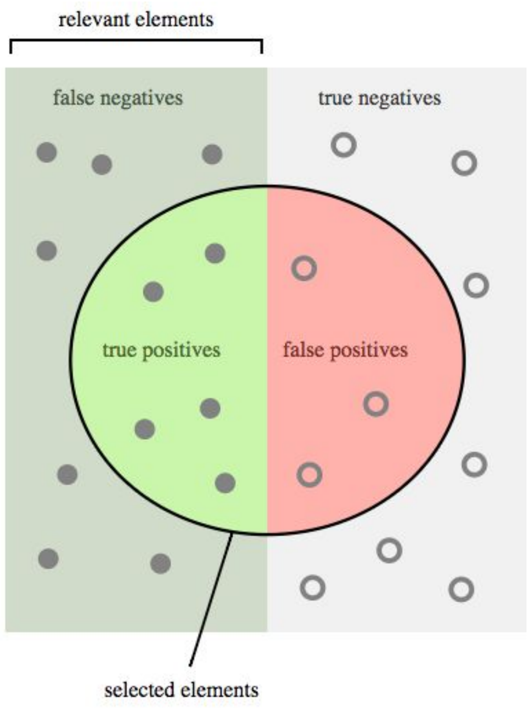
80%

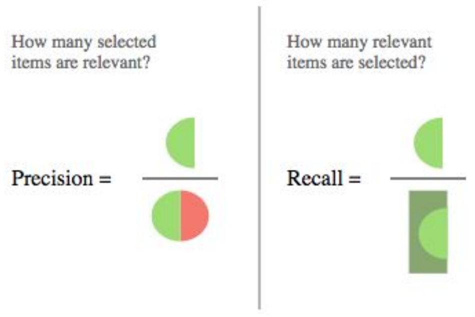
		label	no label
0	predict label	true positive	false positive
100,000	predict no label	false negative 10	true negative 99,990

accuracy = tp + tn / (all) = 99,990 / 100,000 = **99,99%**

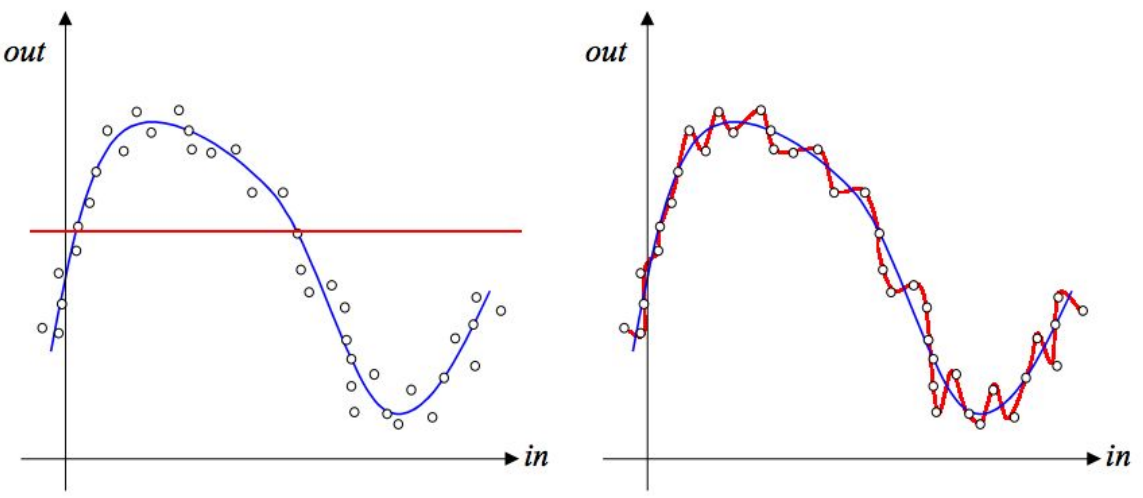
for rare truths, accuracy is not a good measure

Now, let us turn the attention to precision, recall and F-score. Basically, precision is the average probability of relevant retrieval and recall is the average probability of complete retrieval.




How many selected items are relevant?
Precision =
How many relevant items are selected?
Recall =

So, the more we tested our dataset, the more we became curious how to improve the precision and recall. In some cases we got high precision but low recall and vice versa. We have searched and read some relevant papers, and finally found out that overfitting or underfitting could occur and influence our F-score. Please, see the figure below.



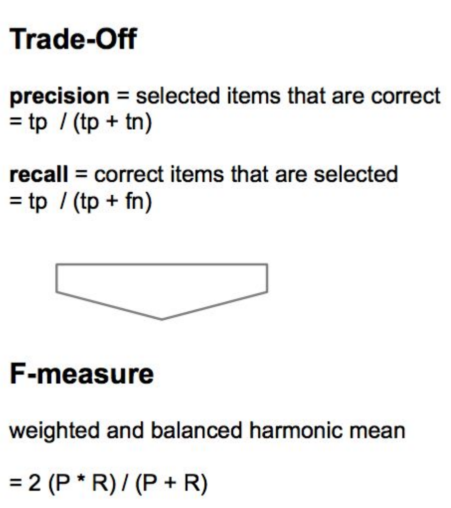
Ignore the data ⇒
Big approximation errors (high bias)
No variation between data sets (no variance)

Get every data point ⇒
No approximation errors (zero bias)
Variation between data sets (high variance)

So, the following actions could be done:

- Getting more training examples (fixes high variance)
- Trying smaller sets of features (fixes high variance)
- Increasing decay coefficient lambda (Fixes high variance)
- Adding features (Fixes high bias)
- Adding polynomial features (Fixes high bias)
- Decreasing decay coefficient lambda (Fixes high bias)

Bias is the difference between the average network output and the regression function. This can be viewed as the approximation error. Variance is the divergence of the approximating function over all the training sets. It represents the sensitivity of the results on the particular choice of data D.



Trade-Off

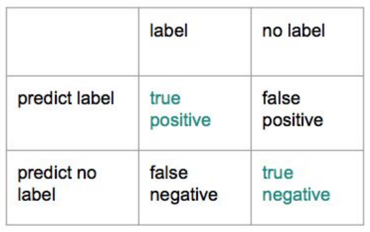
precision = selected items that are correct
= tp / (tp + tn)

recall = correct items that are selected
= tp / (tp + fn)

F-measure

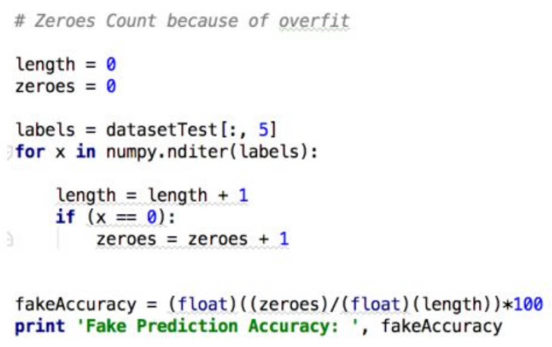
weighted and balanced harmonic mean

= 2 (P * R) / (P + R)



	label	no label
predict label	true positive	false positive
predict no label	false negative	true negative

Feedback and Accuracy Measuring



```
# Zeroes Count because of overfit

length = 0
zeroes = 0

labels = datasetTest[:, 5]
for x in numpy.nditer(labels):

    length = length + 1
    if (x == 0):
        zeroes = zeroes + 1


fakeAccuracy = (float)((zeroes)/(float)(length))*100
print 'Fake Prediction Accuracy: ', fakeAccuracy
```

Critical Note

Advantages of Neural Network based anomaly detection

The first advantage of using neural network would be the ability to generalize from past behavior to detect novel attacks. The accuracy of classification by NN benefits from its classifier algorithm. The classifier algorithm determines the best solution by trying to minimize the number of incorrectly classified cases during the training process. A neural network might be trained to recognize known suspicious behaviors with a high degree of accuracy. NN learning can solve problems with the noisy and complicated training data. NN learning is robust to errors in the training dataset. NN can also detect time series attacks with the help of its strong classification ability. Time series of the sequential attacks can be treated as one NN's input node for analysis. An ideal application in anomaly detection will be to gather sufficient normal and abnormal audit data for a user or a program and then apply a classification algorithm to learn a classifier that can label or predict new unseen data as belonging to the normal class or the abnormal class. With an input layer, a hidden layer and an output layer a neural network can be constructed any arbitrarily complex function. Anomaly detection based on NN can detect novel and modified attacks, decrease the rates of false positives and even detect hiding intrusion by NN's intrinsic black-box feature. The another reason for using NN is that the black box of NN can approximate the internal states of commercial software even if the source code is unavailable.

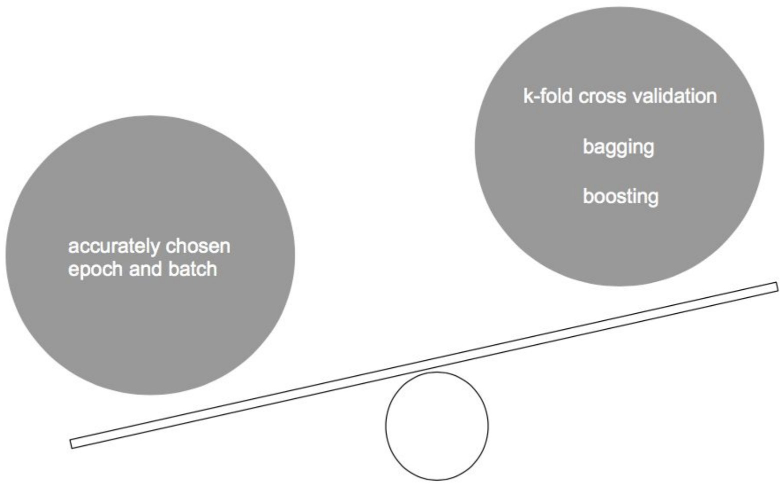
Disadvantages of Neural Network based anomaly detection

The ability of NN to detect anomalous data depends upon the accurate training of the anomaly dataset. The training data and the training methods are critical. The training process needs large amount of data to avoid overfitting. Overfitting is especially dangerous because it can easily lead to predictions that are far beyond the range of the training data. Overfitting can also produce wild predictions in multilayer perceptrons even with noise-free data. Data collection process is also difficult.
The second disadvantage of applying networks to intrusion detection is the black box nature of the neural network. Neural networks have been viewed as a black box that cannot explain how they actually model data. Neural networks adapt the analysis of data in response to the training conducted on the network. The connection weights and transfer functions of the various network nodes are usually frozen after the network has achieved an acceptable level of success in the identification of events. While the network analysis is achieving a sufficient probability of success, the basis for this level of accuracy is not often known.

Conclusion

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, due to our experience in this assignment, extra care is needed to fit a neural network and small changes can lead to different results (number of neurons, epoch iterations, threshold etc.).
There a bunch of ways and techniques to improve the neural network based model. All in all, we came up with some basic ideas that can help to build accurate neural network for anomaly detection (i.e. two-classified dataset):

- Data pre-processing is very important. It needs to be checked if there are any missing values. Also train set could be splitted into three subsets: train set (80% of data with no anomalies), cross-validation set (10% of data with 50% of all anomalies) and test set(10% of data with 50% of all anomalies). We have already given the idea and main purpose of cross-validation. So, before making the prediction of real test set, we can evaluate our model by applying the test set picked from the train set.
- Measuring the Bias and the Variance can help while dealing with over- and under-fitting the data.
- Another good option that we have chosen is using a different architecture on your neural network, a different algorithm or modified features.
- Ensemble learning discussed earlier in the report could be applied (bagging/boosting)



Bibliography

1. Principles of Neural Networks, Daniel graupe, 3rd Edition, (University of Illinois, Chicago, USA)
2. Fundamentals of Neural Networks: AI Course slides

3. Fundamentals of Neural Networks : Architectures, Algorithms and Applications. L, Fausett, 1994
4. An Introduction to Neural Networks (2nd Ed). Morton, IM, 1995
5. Understanding and Using K-Fold Cross-Validation for Neural Networks, James McCaffrey https://visualstudiomagazine.com/articles/2013/10/01/understanding-and-using-kfold.aspx
6. Bagging and Boosting, Amit Srinet, Dave Snyder
7. Neural network ensembles http://www.alglib.net/dataanalysis/mlpensembles.php
8. Boosting Neural Networks, Holger Schwenk, Yoshua Bengio, 2000, Massachusetts Institute of Technology
9. Keras documentation https://keras.io/models/model/
10. Critical study of neural networks in detecting intrusions, Rachid Beghdad, 2008
11. Improve Neural Network Generalization and Avoid Overfitting https://www.mathworks.com/help/nnet/ug/improve-neural-network-generalization-and-avoid-overfitting.html
12. Lambda learning rule for feedforward neural networks, J.M. Zurada, Neural Networks, 1993., IEEE
13. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, N. Srivastava, G. Hinton, Journal of Machine Learning Research, 2014
14. Neural Network Tuning and Overfitting Avoidance http://homepages.gold.ac.uk/nikolaev/311over.htm
15. An Attractor Neural Network Model of Recall and Recognition, E.Ruppin, Y.Yeshurun, 2007