Supervised Classification

January 24, 2015

1 Supervised Learning on Higgs and Bidding Datasets

1.1 Introduction

The goal of this analysis is to use Supervised methods of Machine Learning to detect Higgs boson particle from the noise of various particle collisions created in the Atlas experiment where protons of extra-high energy are brought head-on.

1.2 Data

1.2.1 Higgs Dataset

On July,4 2012 physicists of the Large Hadron Collider announced the discovery of the long-saught Higgs boson particle. Experiment was taking at CERN by ATLAS group where billions of head-on colisions were recorded in the hope that elusive particle will eventually show itself. The method of observing a Higgs particle is through it's decay into another two tau particles. The challenge lies in the fact that these decays are small signal in the large background noise, which makes the problem very interesting for Machine Learning classification.

Dataset Description ATLAS provided dataset with 250000 events: mixture of signal and background. The dataset is characterised by 30 predictor variables (features) prefixed with either:

- PRI (for PRImitives) - "raw" quantities from the bunch collision as measured by the detector
- DER (for DERived) - quantities computed from the primitive features, which were selected by the physicists of ATLAS

Additionally this training dataset includes weight column for each event as well as label ("s" for signal and "b" for background)

Data Wrangling As part of pre-analysis of the data, I have plotted all 30 features to understand their predictive power to distinguish between signal and background.

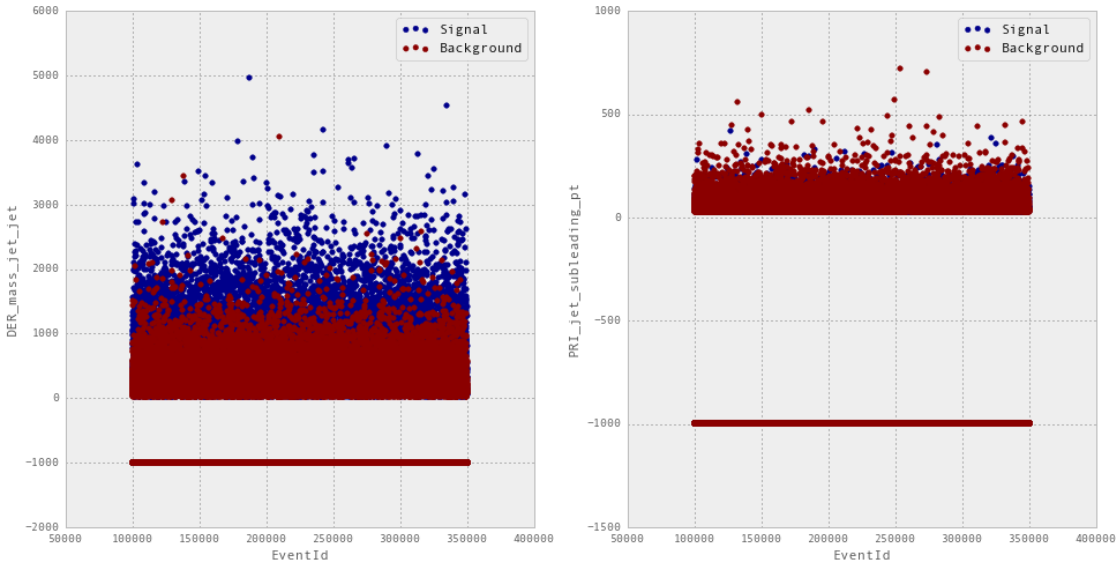
What I have found is:

- there is a lot of missing data in the both DER and PRI features (value = -999.0) which is considered just a noise and upon consulting ATLAS data description I have confirmed that this data values are outside of normal range;
- DER features are better at differentiating the signal as if the signal is being amplified in constrast to PRI features;
- weights columns is not uniformly distributed which means not all events are equially important; so there probabilities will need to be accounted for when calculated accuracy of the classifiers

Both these phenomena are demonstrated below on the example of two features without too much of the loss of generallity.

```
In [1]: %matplotlib inline
        from algo_evaluation.datasets import *

In [2]: df = describe_higgs_raw()
```



As a result of the visual analysis, I made following adjustments to the data prior to classification:

- drop data values (-999.0) as they do not contribute to the accuracy; sometimes such data is considered a missing values and is being replaced with mean, however in this case this might actually hurt the accuracy;
- select only DER features for classification since PRI are already indirectly used and the signal is not so easily separable;

Final data after cleanup and prunning:

```
In [2]: raw_data = load_higgs_train()
        features, weights, labels = raw_data
        print 'Size of the dataset:', features.shape[0]
        print 'Number of features:', features.shape[1]
        print 'Number of positives (signal):', labels.value_counts()['s']
        print 'Number of negatives (background):', labels.value_counts()['b']
```

```
Size of the dataset: 68114
Number of features: 13
Number of positives (signal): 31894
Number of negatives (background): 36220
```

1.2.2 Bidding Dataset

For secondary classification problem, I have selected advertising dataset from my work group research. Dataset represents data for one campaign for 1-4 days (based on the number of impressions shown by campaign per day)

In contrast to real-values features in the Higgs dataset, bidding features are mostly categorical, so in order to apply classification algoritms, I did pre-processing which transformed non-numerical features to numerical.

The last column in the dataset is prediction attribute - we want to predict which impressions led to conversion vs not. The class variable is what we are trying to predict and can be one of the following values

- non-converter : If this impression did not generate lead or conversion
- lead : If this impression made the user to visit lead pixel (for eg homepage, form page, etc)
- converter : If this impression led to conversion

```
In [3]: bid_data = load_bidding_train()
        bid_features, bid_weights, bid_labels = bid_data
        print 'Size of the dataset:', bid_features.shape[0]
        print 'Number of features:', bid_features.shape[1]
        print 'Number of converters:', bid_labels.value_counts()['converter']
        print 'Number of non-converters:', bid_labels.value_counts()['non-converter']
        print 'Number of leads:', bid_labels.value_counts()['lead']
```

```
Size of the dataset: 57970
Number of features: 15
Number of converters: 399
Number of non-converters: 54615
Number of leads: 2956
```

```
/Users/maestro/anaconda/lib/python2.7/site-packages/pandas/io/parsers.py:1130: DtypeWarning: Columns (7
  data = self._reader.read(nrows)
```

1.3 Higgs Classification

1.3.1 Decision Trees

The goal here is to create a model which predicts signal by learning simple decision rules inferred from derived features.

Splitting data Prior to running and tunning the classifier from sklearn library, I've split the dataset, leaving 1/3 out for evaluation purposes. This gave me the initial benchmark of 78% accuracy on the test set. Given the underlying difficulty of detecting higgs signal in general, the accuracy "out-of-the-box" was not bad, however I wanted to see how much more can be achieved with indirect prunning.

```
In [101]: dataset = split_dataset(features, weights, labels)
          for category in dataset:
              data = dataset[category]
              print '{}: {}'.format(category, {d: len(data[d]) for d in data})
```

```
test: {'labels': 22478, 'weights': 22478, 'features': 22478}
training: {'labels': 45636, 'weights': 45636, 'features': 45636}
```

Note: Same splitting applies to the rest of algorithm evaluation as well

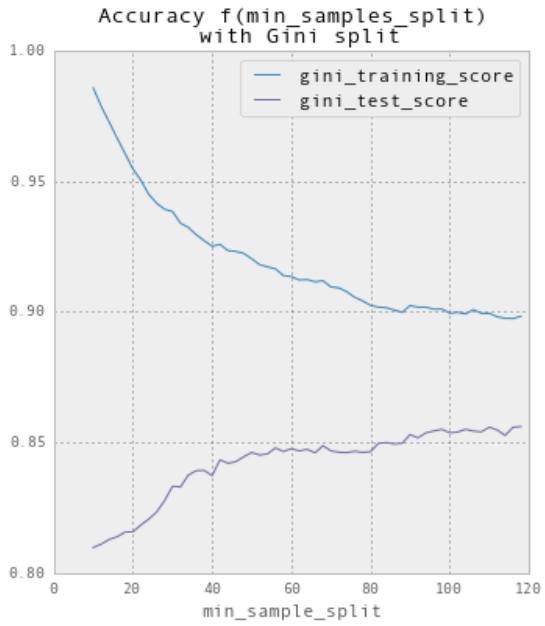
Prunning by tunning minimum number of samples required to split an internal node Below I plotted two different versions of the accuracy function of decision tree complexity expressed through min samples required to split - one for the Gini splitting criterion and another for entropy.

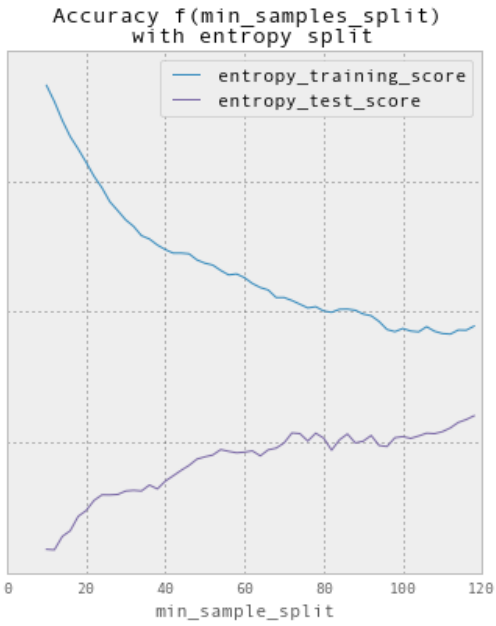
Note: Here and for all consequent graphs I have used rolling means to smooth the accuracy function to remove the sensitivity.

```
In [22]: from algo_evaluation.algos import decision_tree as dt
```

```
In []: df = dt.estimate_best_min_samples_split()
```

```
In [48]: dt.plot_accuracy_function(df, smoothing_factor=5)
```





Observation on min_sample_split:

Given that default setting for minimum number of samples required to split is 2, the classifier was clearly overfitting the data. By tunning the parameter, I was able to increase the test accuracy to above 85% while decreasing accuracy on training dataset. By visaul inspection, it can be inferred that optimal setting for minimum number of samples required to split is around 60.

Observation on splitting rule:

Generally, the decision-tree splitting criteria matters as entropy based criteria favors multinominal features? However, it does not appear to make a big difference on the higgs dataset. Both 'gini' and 'entropy' produce similar accuracy trends with hardly detectable slower ramp-up of the entropy for smaller values of min_samples_split.

Additional tunning of the classfier, such as maximum depth of the tree or minimum number of samples required to be at a leaf node did not contribute to the accuracy of the predictions, so they were left to default.

Below are the final scores achieved by Decision Tree classifier:

```
In [23]: dt_training_accuracy, dt_test_accuracy = dt.run_decision_tree(raw_data, min_samples_split=60)
         print 'Accuracy on training data:', dt_training_accuracy
         print 'Accuracy on test data:', dt_test_accuracy

Accuracy on training data: 0.912119515125
Accuracy on test data: 0.859211327755
```

1.3.2 Neural Networks

Since sklearn does not implement Neural network, in this analysis I am using pybrain library. Same goal as in the decision trees evaluation: classify Higgs boson from the background.

Fully connected Neural Network is constructed with the following specifications:

- input layer - 13 sigmoid neurons
- hidden layer - 19 sigmoid neurons
- output layer - 1 softmax neuron (since output should be binarized)

- training algorithm - backpropagation

```
In [207]: from algo_evaluation.algos import neural_network as nn
```

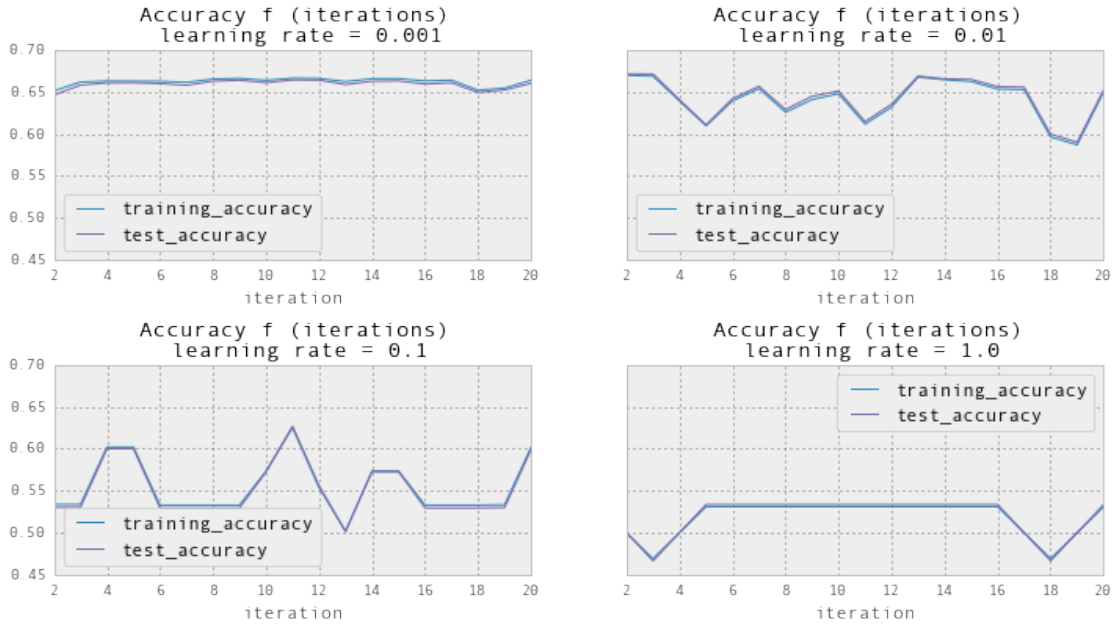
```
In [203]: nn_df = nn.estimate_training_iterations(n_iterations=20)
```

I have run the experiment for up to 500 iterations (only 20 iterations shows here due to time constraints) and unfortunately the network is not learning very well.

Neural Net on Higgs is the most stable with learning rate 0.001 as accuracy curve is not jumping very much. Demonstrated below are learning curves for 4 settings of learning rates: 0.001, 0.01, 0.1 and 1.0.

Due to poor results, it is concluded that Neural Networks is not the best algorithm for detecting Higgs.

```
In [210]: nn_plot = nn.plot_accuracy_function(nn_df, smooth_factor=2)
```




```
In [211]: epochs, nn_training_error, nn_test_error = nn.run_neural_net(raw_data)
          print 'Total epochs (iterations) trained', epochs
          print 'Accuracy on training data:', 1 - nn_training_error / 100
          print 'Accuracy on test data:', 1 - nn_test_error / 100
```

```
Total epochs (iterations) trained 5
Accuracy on training data: 0.53314196814
Accuracy on test data: 0.528940694933
```

1.3.3 AdaBoost

AdaBoost is an example of the ensemble classifier, where a collection of weak learners are combined to produce a meta estimator.

Sklearn python library is using DecisionTrees as the base estimator, so I should be able to get at least same accuracy as found during DecisionTrees evaluation.

Striving to increase the performance above my benchmark, I tunned two parameters:

- maximum number of estimators at which boosting is terminated

- learning rate

There is an obvious tradeoff between these two parameters and using grid search I was able find the most suitable combination for my Higgs dataset.

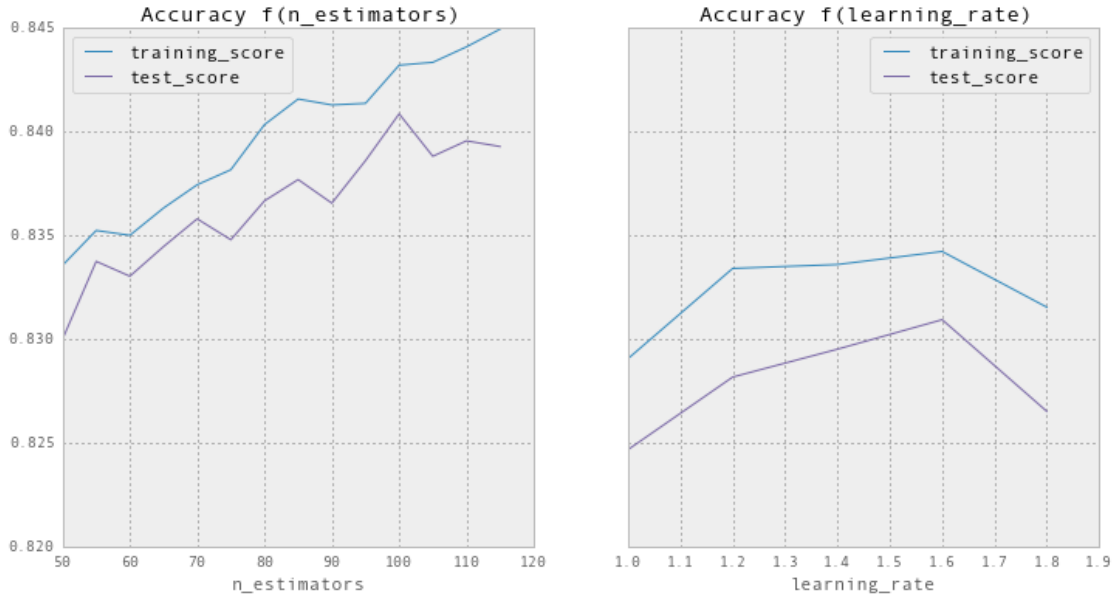
```
In [24]: from algo_evaluation.algos import adaboost as ab
```

Accuracy functions plotted below showed the expected behavior of the classifier.

- increasing number of estimator is positively correlated with the accuracy; the optimal number n_estimators ˜ 100 did not however improve the accuracy of what I was already getting with Decision Trees
- interestingly, very small values of learning rate were negatively affecting the accuracy which means classifier was overfitting; the graph below showed there is an optimal range of learning rates beyond which accuracy tanks drastically

```
In [15]: estimator_df = ab.estimate_best_n_estimators()
         learning_rate_df = ab.estimate_best_learning_rate()
```

```
In [16]: ab.plot_accuracy_functions(estimator_df, learning_rate_df, smoothing_factor=5)
```

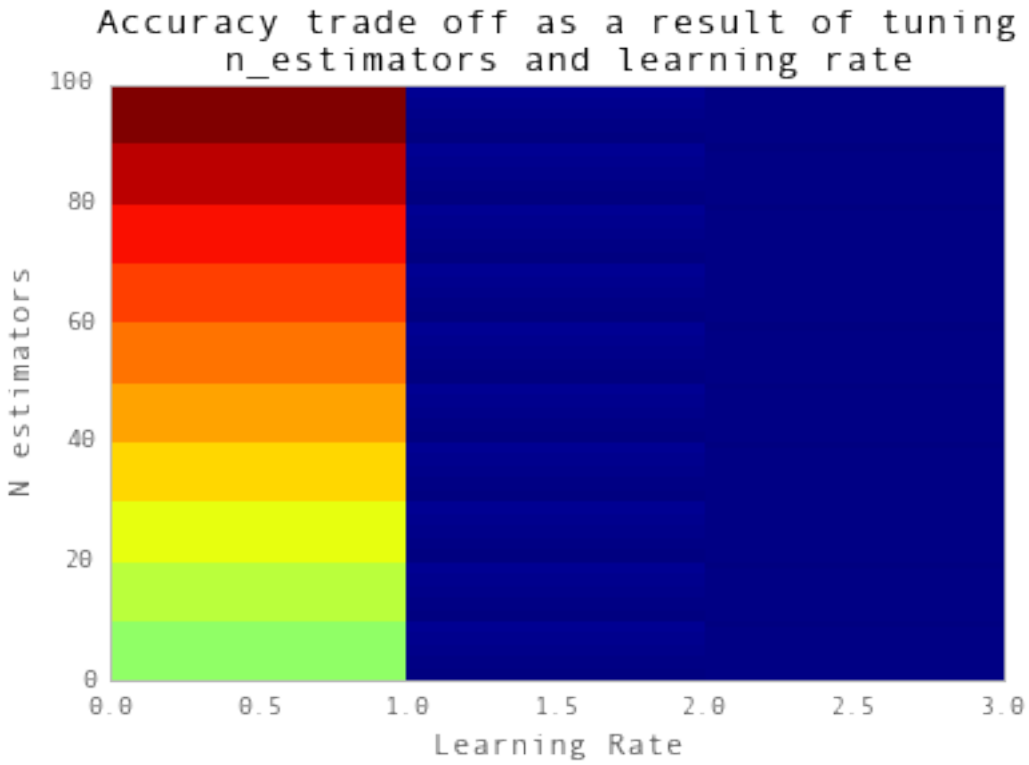


It is easier to observe the accuracy tradeoff with heatmap visualization.
From looking at the plot, following conclusions could be made:

- anything with learning rate higher than 1.0 has low accuracy as indicated by color blue
- as number of estimators is growing, accuracy increases as long as we are in acceptable range of the learning rate

```
In [50]: trades_df = ab.tradeoff_estimators_learning_rate(raw_data)
```

```
In [105]: heatmap = ab.plot_tradeoff(trades_df)
```



Below are the final scores achieved by AdaBoost classifier:

```
In [25]: boost_training_accuracy, boost_test_accuracy = ab.run_AdaBoost(raw_data,
                                                                        n_estimators=85,
                                                                        learning_rate=1.0)
         print 'Accuracy on training data:', boost_training_accuracy
         print 'Accuracy on test data:', boost_test_accuracy
```

```
Accuracy on training data: 0.840054465166
Accuracy on test data: 0.843939505107
```

1.3.4 Support Vector Machines

Support Vector Machines is very effective in high-dimensional spaces and given that I have selected 13 features so far for the Higgs dataset, SVM is expected to work very well. From my pre-analysis of the data, linear separability was out of question, so it is important to choose a good non-linear kernel.

Default kernel is `rbf` and upon training on the dataset it classifier every single example correctly (more on this phenomenon in the conclusion section).

```
In [182]: from algo_evaluation.algos import svm
```

```
In [144]: svm_training_accuracy, svm_test_accuracy = svm.run_svm(raw_data,
                                                                 regularization_term=1.0,
                                                                 gamma=0.0)
          print 'Accuracy on training data:', svm_training_accuracy
          print 'Accuracy on test data:', svm_test_accuracy
```

```
Accuracy on training data: 1.0
Accuracy on test data: 0.995929554356
```

Even though the accuracy is perfect, I performed multiple experiments tunning:

- regularization parameters 'C': [1, 10, 100, 1000]
- gamma 'gamma': [1e-3, 1e-4]
- kernel: rbf

Using grid search with python library, none of the parameters combinations gave better results, so there isn't anything interesting to plot here.

In addition to SVM parameters, I also changed the size of the dataset (each iteration size = 10X of the previous) and suprisingly enought even on the 1/10 of the original dataset size, SVM still performed as good as on full dataset.

1.3.5 K-Nearest Neighbours

K-Nearest Neighbours is an example of the instance based classification where instead of the learning the predictive function, all examples are stored and considered and when the new event is encountered, as set of similar events is used for classification.

Given this intuition behind the classifier, KNN is the best suited for classifying Higgs particle. To detect signal from background, we do not need to look at the whole data, but rather similar decaying events.

The main question to answer here, how many of such events should we look for.

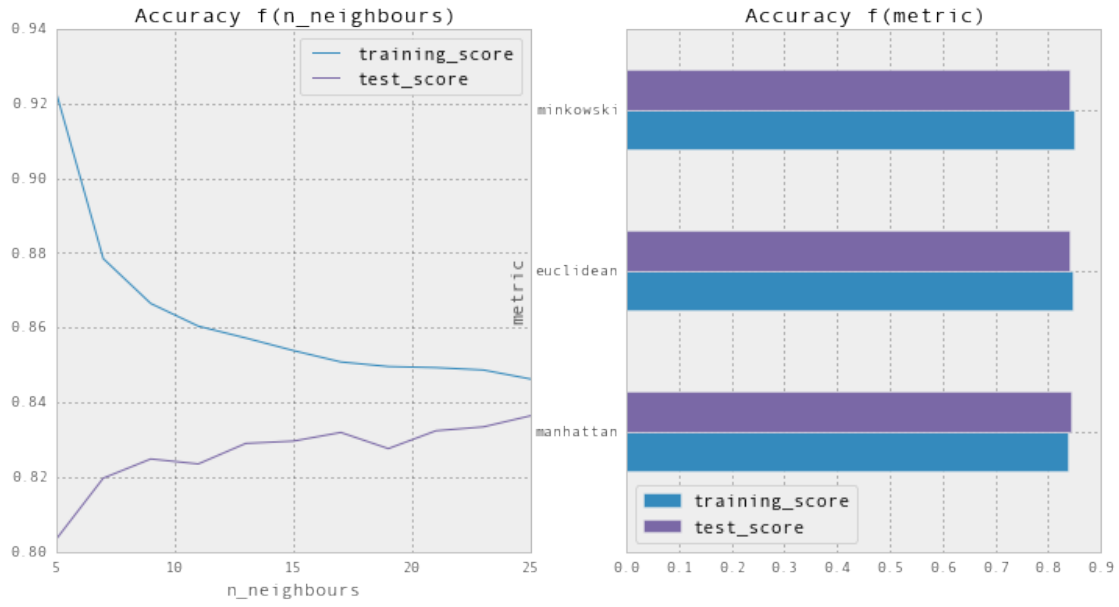
The accuracy curve below given as a function of the number of neigbours suggests 10 to be an optimal number which generalizes the dataset pretty well.

```
In [177]: from algo_evaluation.algos import knn

In [154]: knn_df = knn.estimate_best_n_neighbours()

In [169]: p_df = knn.estimate_best_power()

In [176]: knn_plot = knn.plot_accuracy_function(knn_df, p_df, smoothing_factor=3)
```



Additionally, KNN in theory classifies examples very different depending on the distance metric used.

However in the current dataset, metric did not alter accuracy very much and so default can be used. Metrics attempted in the evaluation:

- euclidian
- manhattan
- minkowski

Final estimation was using euclidian distance metric and produced following results:

```
In [27]: knn_training_accuracy, knn_test_accuracy = knn.run_knn(raw_data, n_neighbours=10)
         print 'Accuracy on training data:', knn_training_accuracy
         print 'Accuracy on test data:', knn_test_accuracy
```

```
Accuracy on training data: 0.888995233367
Accuracy on test data: 0.858143125028
```

1.4 Performance Comparison

After classifying Higgs particle with presented algorithms, it is very interesting to compare they accuracy scores on both training and test data.

- Accuracy across three algorithms - Decision Tree, AdaBoost, KNN - is comparable and around 85%.
- SVM algorithm exceeded expecations right "out of the box". Having accuracy of 0.99 on test data is extremely high which makes me conjecture that perhaps ATLAS simulated events for training using SVM (reading additional literature on particle detection from CERN boosts this hypothesis).
- Neural Network on the other side produced very low accuracy (53%) regardless of tuning and number of iterations. Another drawback of this algorithm is that it took very long time to train.

Aggreggate of all scores and their comparison is shown in the table below:

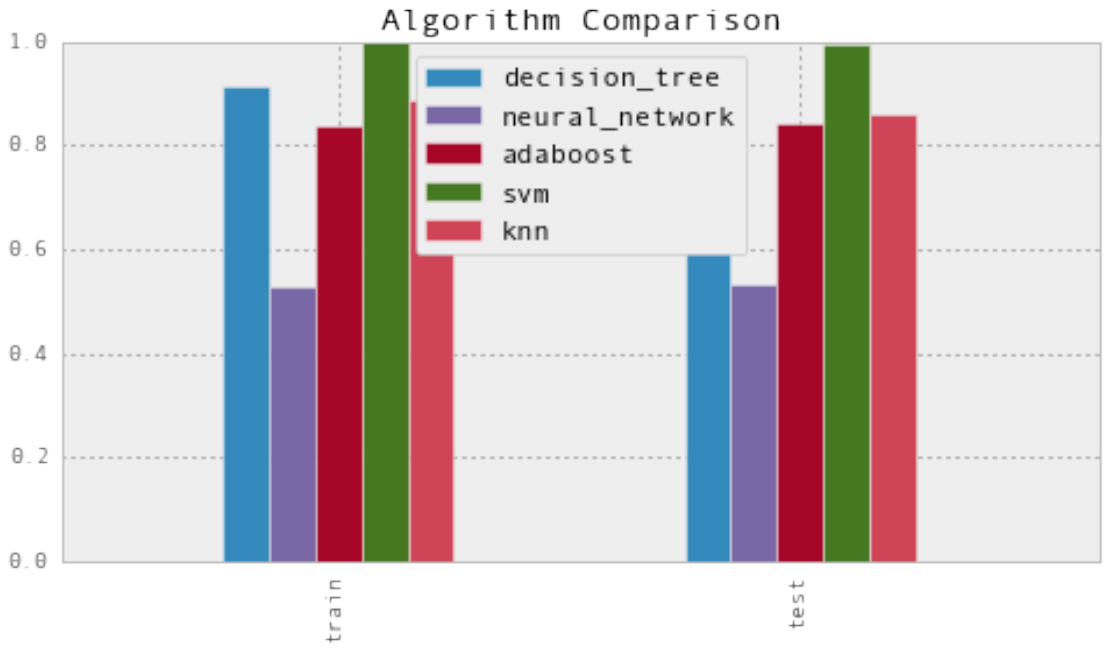
```
In [117]: training_scores = [dt_training_accuracy,
                             1 - nn_training_error/100,
                             boost_training_accuracy,
                             svm_training_accuracy,
                             knn_training_accuracy]
          test_scores = [dt_test_accuracy,
                         1 - nn_test_error/100,
                         boost_test_accuracy,
                         svm_test_accuracy,
                         knn_test_accuracy]
          algorithms = ['decision_tree', 'neural_network', 'adaboost', 'svm', 'knn']
          scores = pd.DataFrame.from_records([training_scores, test_scores],
                                             columns=algorithms,
                                             index=['train', 'test'])
          scores
```

Out[117]:

	decision_tree	neural_network	adaboost	svm	knn
train	0.912120	0.530030	0.840054	1.000000	0.888995
test	0.859211	0.535258	0.843940	0.996152	0.858143

Plot accuracy scores on both training and test data across all algorithms.

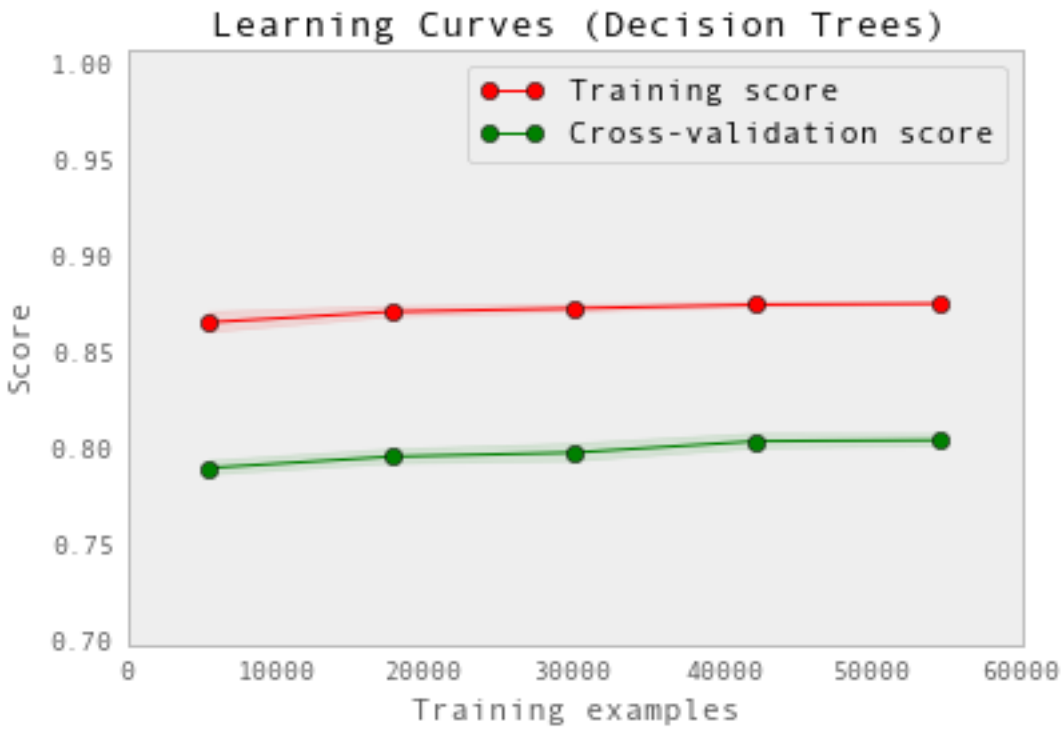
```
In [212]: comparison = scores.plot(kind='bar', figsize=(8, 4), title='Algorithm Comparison')
```

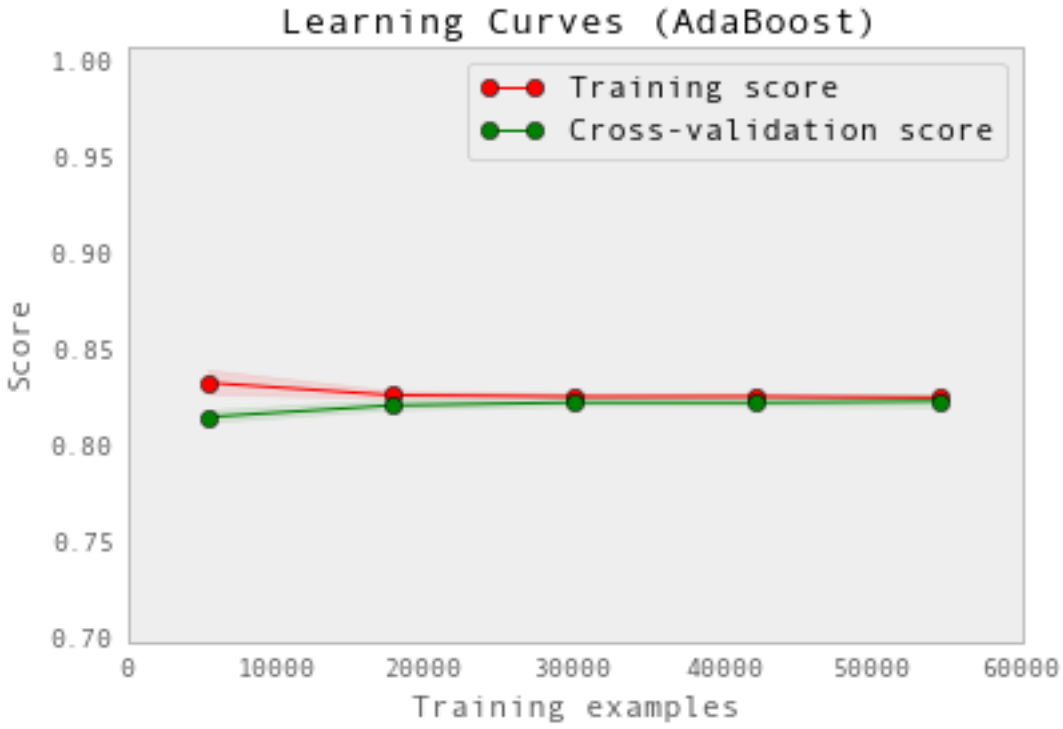


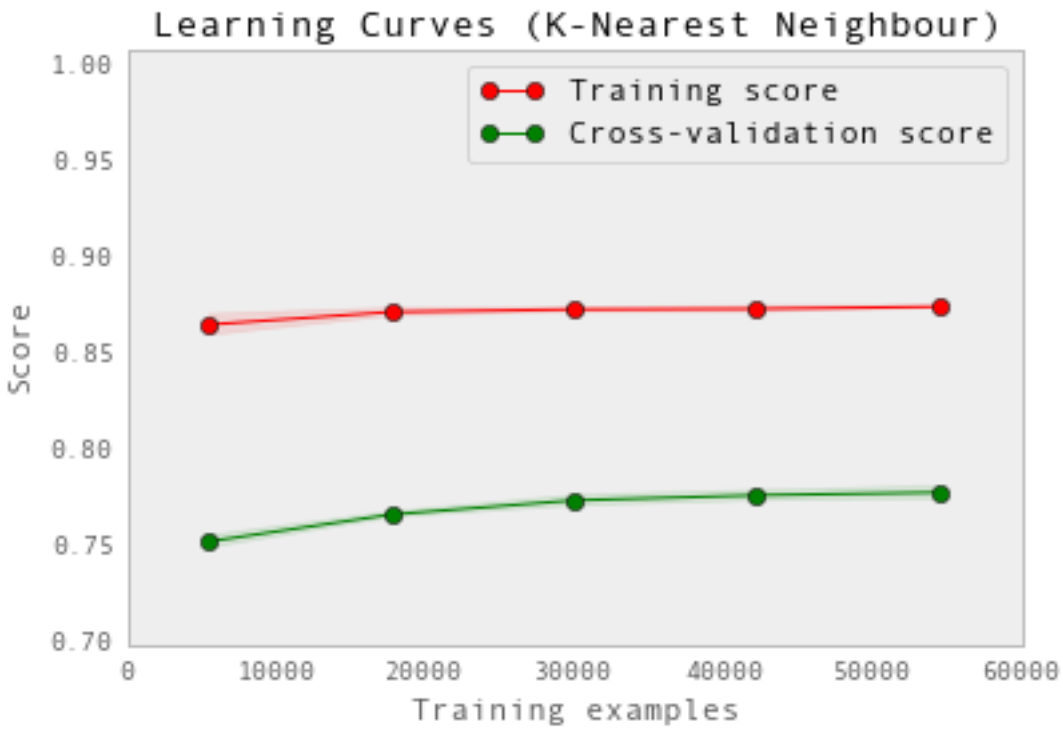
Higgs Detection Learning Curves So far the analysis of the accuracy was demonstrated as a function of algorithm parameters. It is also interesting to look at the learning curves as a function of the sample size (number of examples on the dataset)

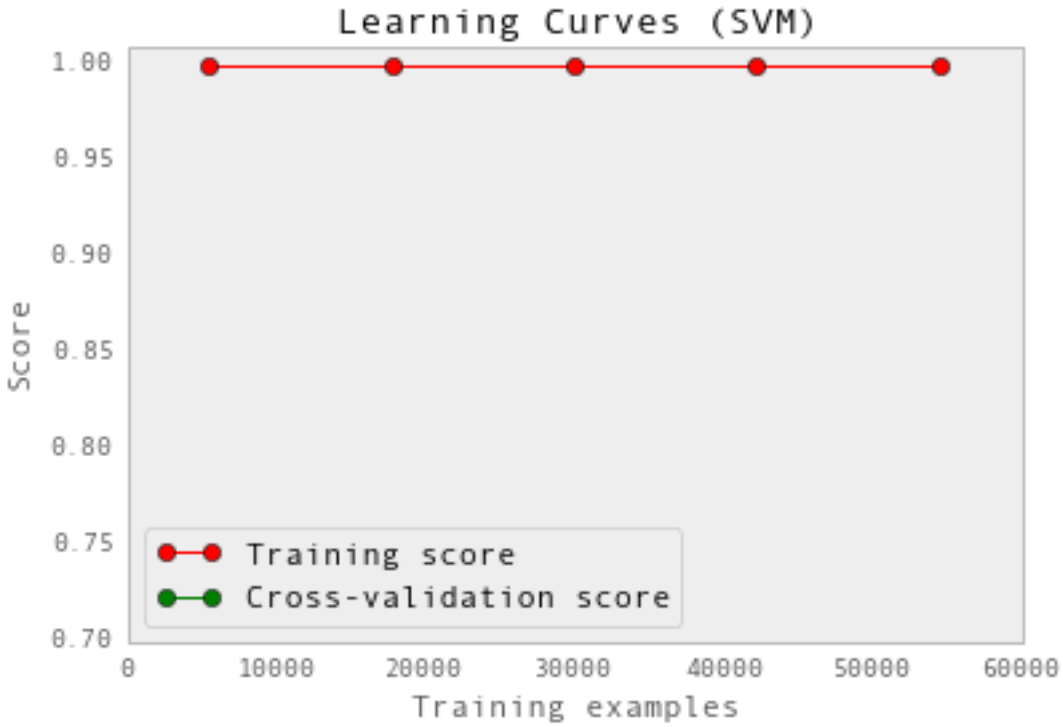
```
In [4]: from algo_evaluation.plot_learning_curves import plot_learning_curves
```

```
In [10]: higgs_learning_curve = plot_learning_curves(raw_data)
```


Learning Curves (Decision Trees)
Training score
Cross-validation score
Score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
10000
20000
30000
40000
50000
60000
Training examples


Learning Curves (AdaBoost)
Training score
Cross-validation score
Score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
10000
20000
30000
40000
50000
60000
Training examples


Learning Curves (K-Nearest Neighbour)
Training score
Cross-validation score
Score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
10000
20000
30000
40000
50000
60000
Training examples


Learning Curves (SVM)
Training score
Cross-validation score
Score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
10000
20000
30000
40000
50000
60000
Training examples

Sample of the Higgs dataset is large enough (60k examples) so that accuracy learning curve stabilized at 20k which seems to be enough acr

```
In []: bid_learning_curve = plot_learning_curves(bid_data)
```

Generally, Higgs particle detection is a very difficult machine learning task, but what the above experimentation showed me that even difficult problems like that could be tackled and provide the sufficient accuracy.

Having more domain knowledge could be helpful in feaure aggregation and fine-tunning of the algorithms (especially the ensemble ones) to achieve a better score.

Additionally I would have liked to have more processing power to see if Boosting is capable of doing more than just 85%.

1.5 Acknowledgement

Following python libraries were used for the evaluation of algorithms:

- scikit-learn (SVM, AdaBoost, KNN, Decision Tree, Accuracy and Error evaluation)
- pybrain (Neural Network)
- pandas (data analysis)
- numpy (data wrangling)
- matplotlib (plotting)

1.6 References

[1] Higgs Boson Machine Learning Challenge: https://www.kaggle.com/c/higgs-boson

[2] Learning to discover: the Higgs boson machine learning challenge: http://higgsml.lal.in2p3.fr/files/2014/04/documentation_v1.8.pdf

[3] Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC: http://arxiv.org/abs/1207.7214

[4] Support Vector Machines in Analysis of Top Quark Production: http://arxiv.org/abs/hep-ex/0205069

[5] Stephen Marsland. Machine Learning: An Algorithmic Perspective. CRC Press, 2009

[6] Scikit Learn Documentation, Online Available, at http://scikitlearn.org/stable/documentation.html

[7] Pybrain Documentation, Online Available, at http://pybrain.org/docs/index.html