Proof of Concept for ML

Overall Idea - We want to create an algorithm that can run in Quantopian and generates simple buy and sell signals based on the learning from Google data in some highly liquid securities and index funds, AMZN, AAPL, GOOG, SPY etc.

Technical Details

Popular indicators can be simulated by a simple two layered neural network. Teaching the neural network to follow a single equity will build the needed indicators/features of this equity within the network.

Algorithms rely on these indicators, so it should be possible to

- extract needed indicators
- train the algorithm to take the correct indicators into account

Memory of the RNN could add even more features of that equity, because for sequence prediction/ stream generation they use LSTM networks rather successfully

There are also industry and volatility indicators (TRIN/LIBOR/etc) that require weighting of multiple securities at the same time. Cross-security learning is more resource-consuming and will be done in future stages, when we can aggregate the indicators created from the single equity NN.

POC 1 - Create a very simple algo that we control that creates a signal. Doesn't need to work perfectly.

Sequence Prediction - do we need to worry about this?

(failed concept)

Train RNN to follow market prices. Then make strategy based on predictions coming from the network.

Input parameters:

- Data for a single equity from google finance

Result: predicted chart.

Output is too noisy. But more importantly there is no way to tell “level of confidence” of particular parts of the predicted sequence.

Signal Prediction

Train RNN to predict some of the ideal buy/sell events.

Input parameters:

- Data for a single security from google finance
- Ideal strategy for this data

Result: buy/sell signals for a single security chart

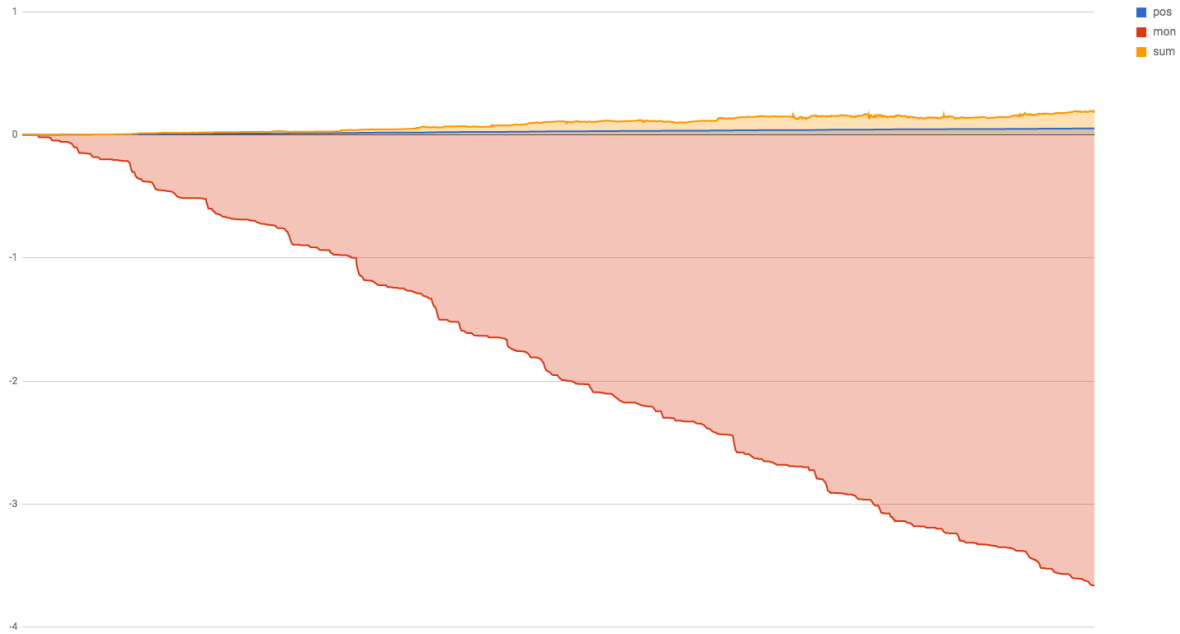
Model 9

Output is noisy, however there are some minor spikes for buy/sell happening from time to time

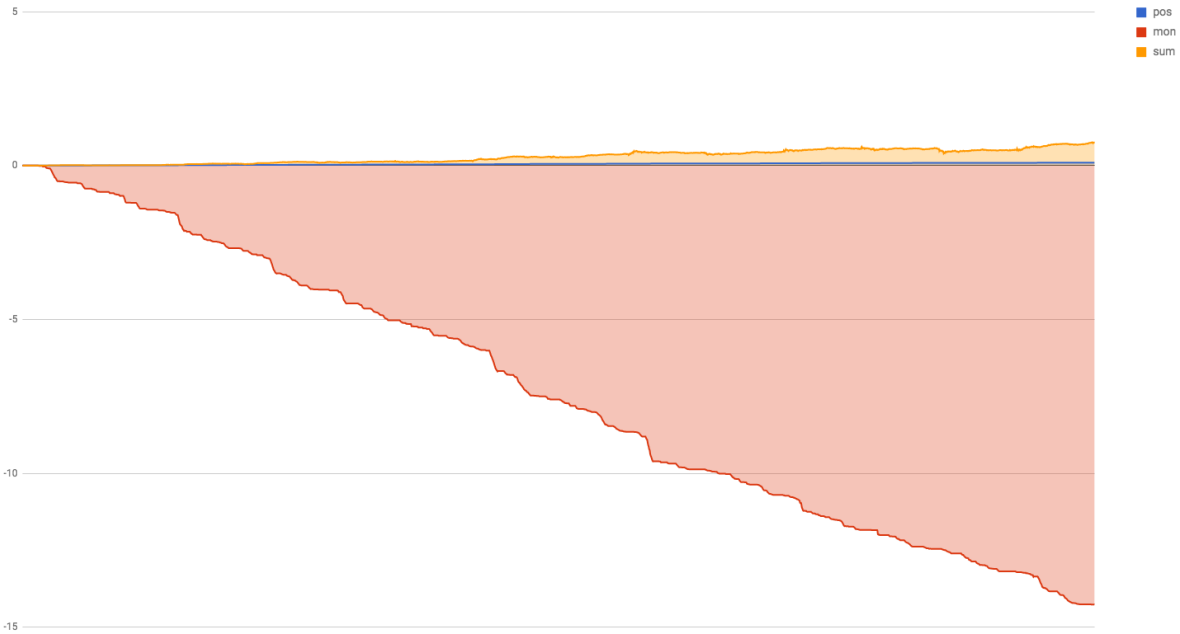
After running model on several datasets it looks like the trained model from the last iteration.

> See 9.model folder. Trained on MSFT quotes from google finance. 1000 days of minute candles ~390 records per day. Only close price of candles used for now.

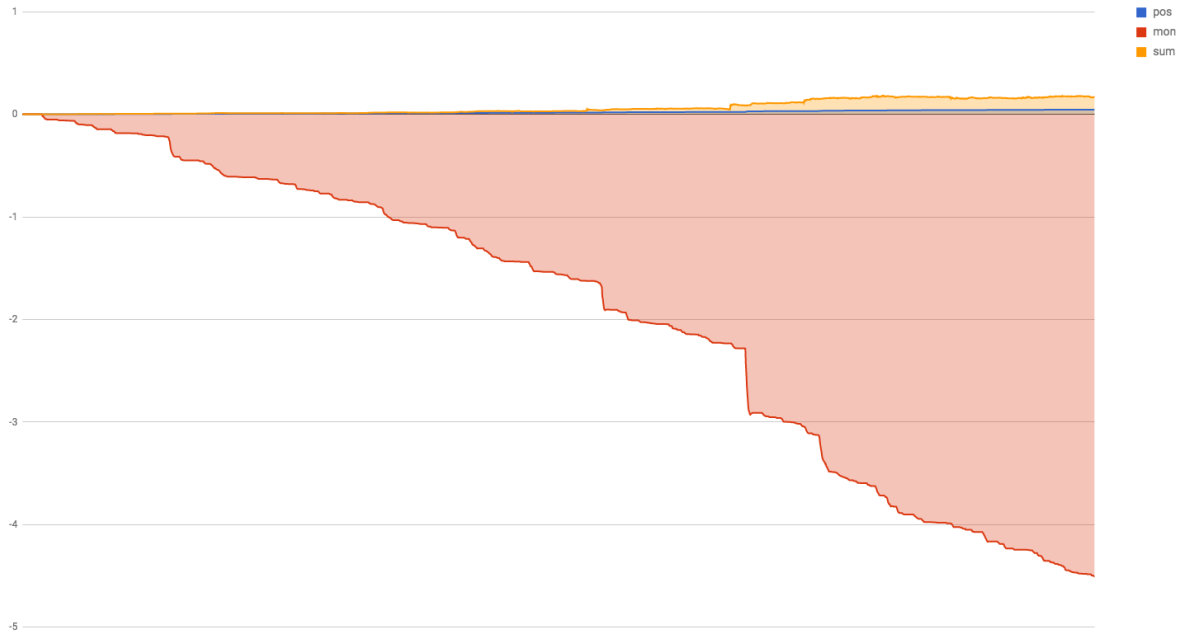
Model has a significant bias towards buying equities and infrequently sells, resulting in an imbalance between cash and owned equities



Same behaviour happened when algorithm was applied to Apple (1000 days of minutes candles)



And Time Warner (100 days of minute prices)



Model sequence has 80 past records as an input for every step (80 minutes history), 256 hidden neurons, 2 classes (buy/sell signals), sigmoid activation. Prediction frame of a training algorithm is 400

Though model shows some net growth, random buying will show some growth too, because all these instruments were rising during last 1000 days.

Model 10 and Control

Same configuration as model 9, but with additional class for checking for actual learning progress.

In order to check learning progress, I've added two outputs to the network: constant and a simple indicator looking like Williams %R

$$(h - p)/(h - l)$$

Where p is current price, h and l are respectively high and low on some time frame prior to p.

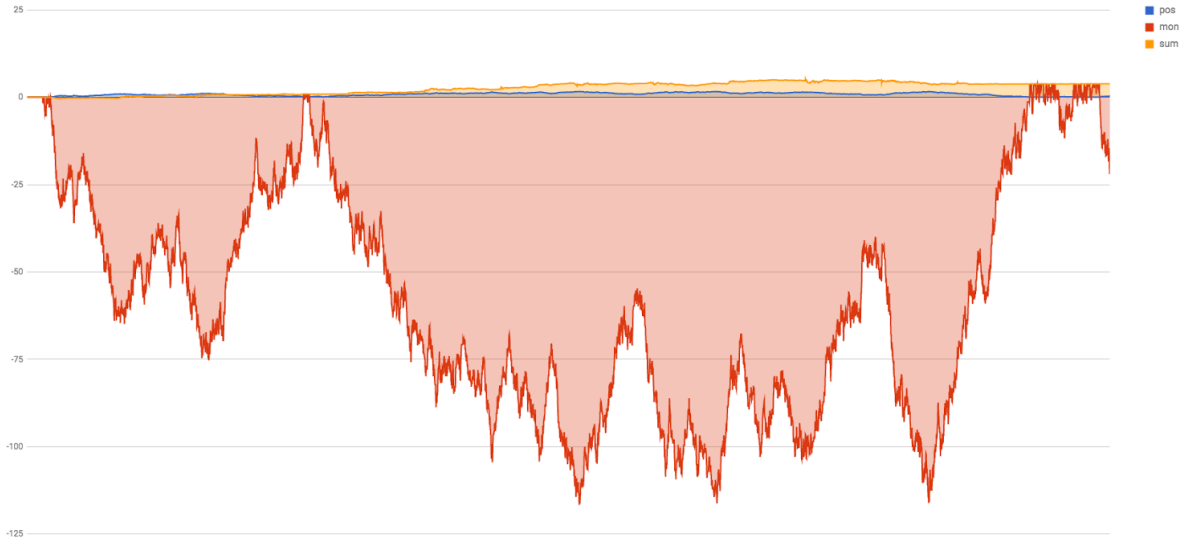
So learning progress could be checked by deviation from constant and this indicator.

Also we could check if neural network is applicable to charts other than the source one.

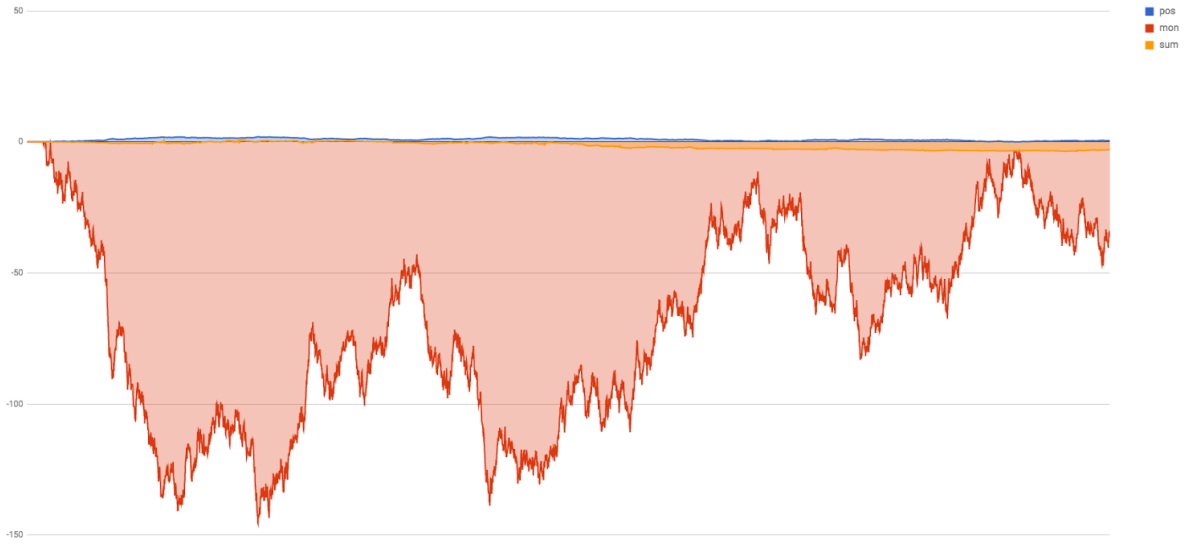
Besides that I've setup some random strategy to check if we really have some progress.

Funny thing is that the random strategy on the bullish market consistently wins by a small margin.

Here is a typical example of random strategy output on a generally rising market:



And consistently loses on bearish market:



The important difference of learned and random algos are that learned algo don't like selling acquired positions.

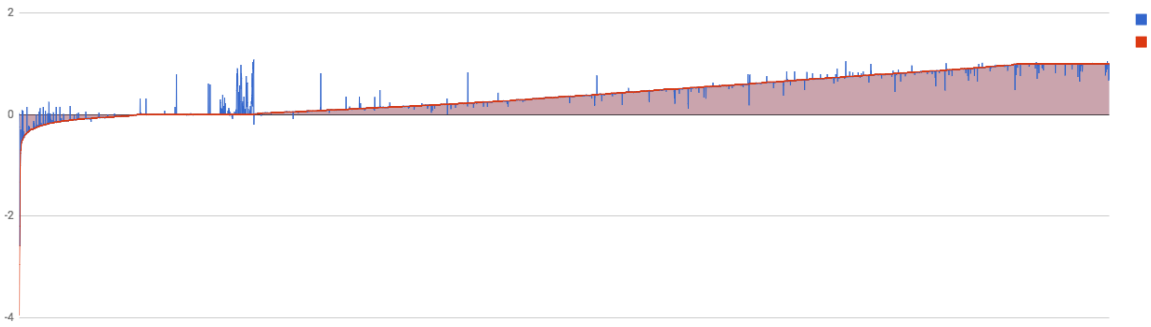
Learning times with control outputs for LSTM network became too high, so I had to switch to a simple perceptron

To reduce overfitting, most models were trained on several 30K iterations with batch size around 1000. Most of the time it was around 90K iterations. Since learning dataset is rather small (by machine learning standards) and test dataset is absolutely tiny there is no way to check if overfitting is becoming an issue, so it's better to left model somewhat undertrained.

Model 11

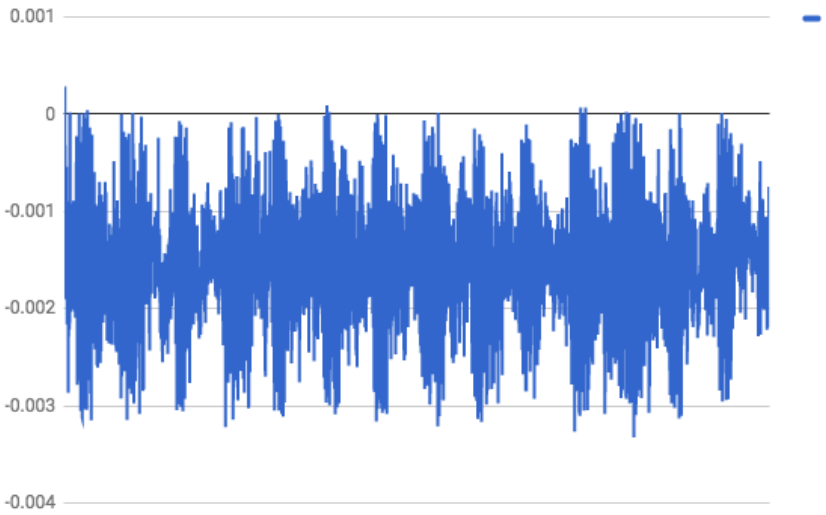
Once I've replaced LSTM with 2 layered perceptron with softmax activation and Adam optimizer learning happened much more quickly.

Control indicator deviations on sorted data set:



Blue is learned one, red is ideal value. Chart is sorted by ideal values

Constant control:



Since learning rate is 0.01, -0.0003 is quite reasonable error

Buy vs ideal strategy buy, sorted by ideal signal value:

(big red values correspond to the strong buy signals from the ideal strategy. Blue is the learned behaviour for these signals)

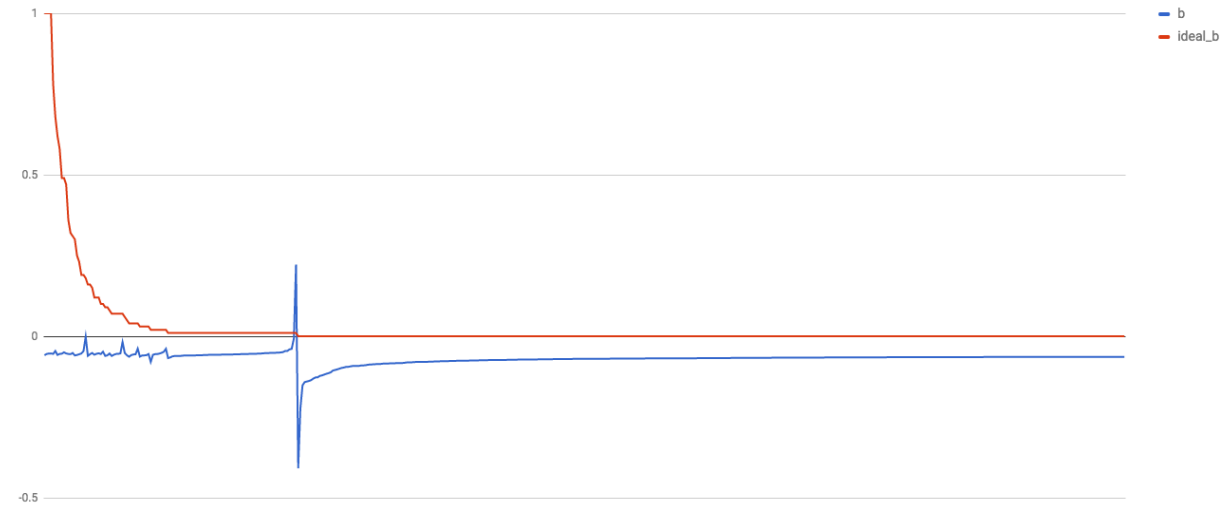
It's the left 10% of the chart. NN is rather good on not firing "buy" signal, so the interesting thing is about the left part. We can see here around 8 detected features for buy events. Also there is a little deviation below zero for the learned algorithm, but it is ignored by testing routine. Only spikes perform buys.

Sell distribution don't look that consistent about learned features:

There is a lot of chatter around zero for the rest of the chart.

Problems with selling were still there. Probably because there are no strong criterias to sell at the bullish market, so I have to find broader learning set.

The most successful part of testing this model is this:



Red is the distribution of the ideal strategy buy signals for Time Warner, sorted from strong to weakest signal.

Blue is the response of the network trained on the Microsoft data.

Chattering in the left side with tiny positive signals and lack of false signals on the most part of the chart where no buy signal is expected is signifying that network is having some very weak, but working indicators inside applicable to any stock.

Here is the analysis of the worst discrepancies of ideal strategy and one learned on Microsoft:

All false positives compared with all ideal positives (same scale as before).

Rest 80% of the chart both ideal and learned strategy doesn't produce any signals

So we need to negate false positives and emphasize universal positives.

This might take too much time with direct learning and optimization. There should be another negating classifier for non-universal features.

Model 12, 13

Trying different intervals (hourly, daily, etc), combining different securities in pairs, feeding benchmark (SPY) as a third input.

The most successful configuration for stock pair and a benchmark was

s(s[s(isec1Isec1+bsec1) +s(isec2Isec2+bsec2) + s(ispyIspy+bspy)]L1+b1)L2+b2)

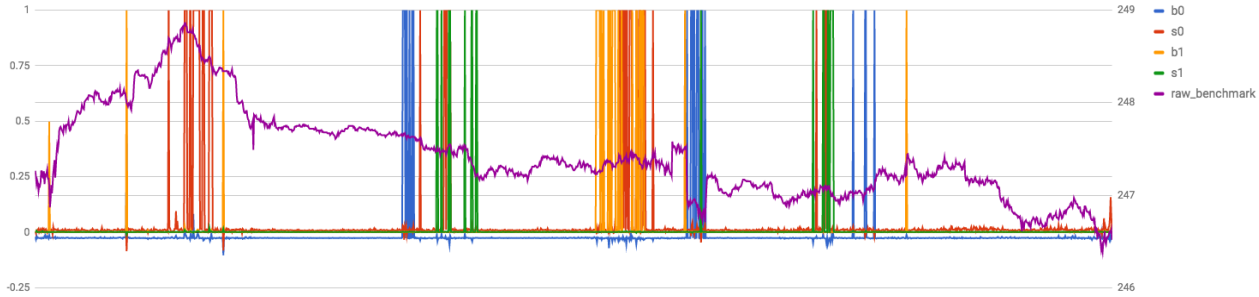
i: input vectors of last 80 closing prices

I: matrices 80x384 weights, so iI is a vector of 384 elements

b: vector of biases

s: softmax logistics function on the vector (each element is si = e-xije-xj)

L: layer matrices, first 384x768. Second 768x4.



Sample chart of learned signals for model 13 overlayed on SPY

Exporting (Model 14)

Model was retrained with “one hot” buy/sell signals. All ideal buy/sell signals values were considered of the same value.

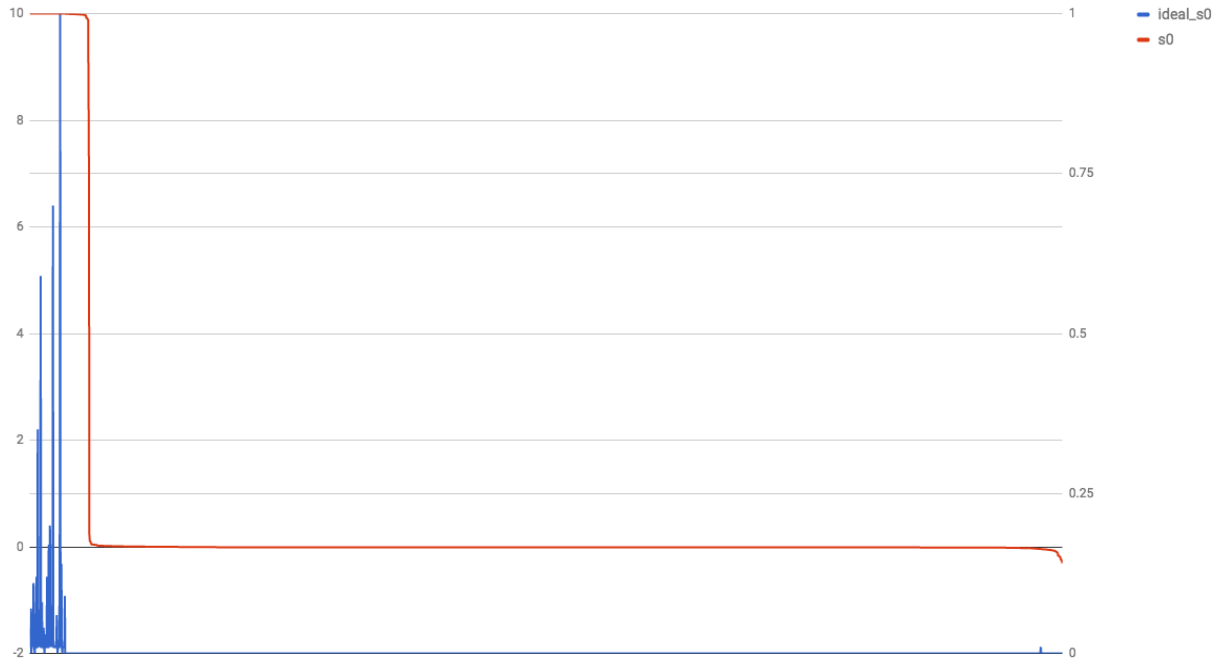
In order to get rid of the tensorflow dependency, I’ve made model export in form of Python code and json. See model_exporter.py for export, for evaluation evaluator.py.

Zipline Portfolio Evaluation (Model 15)

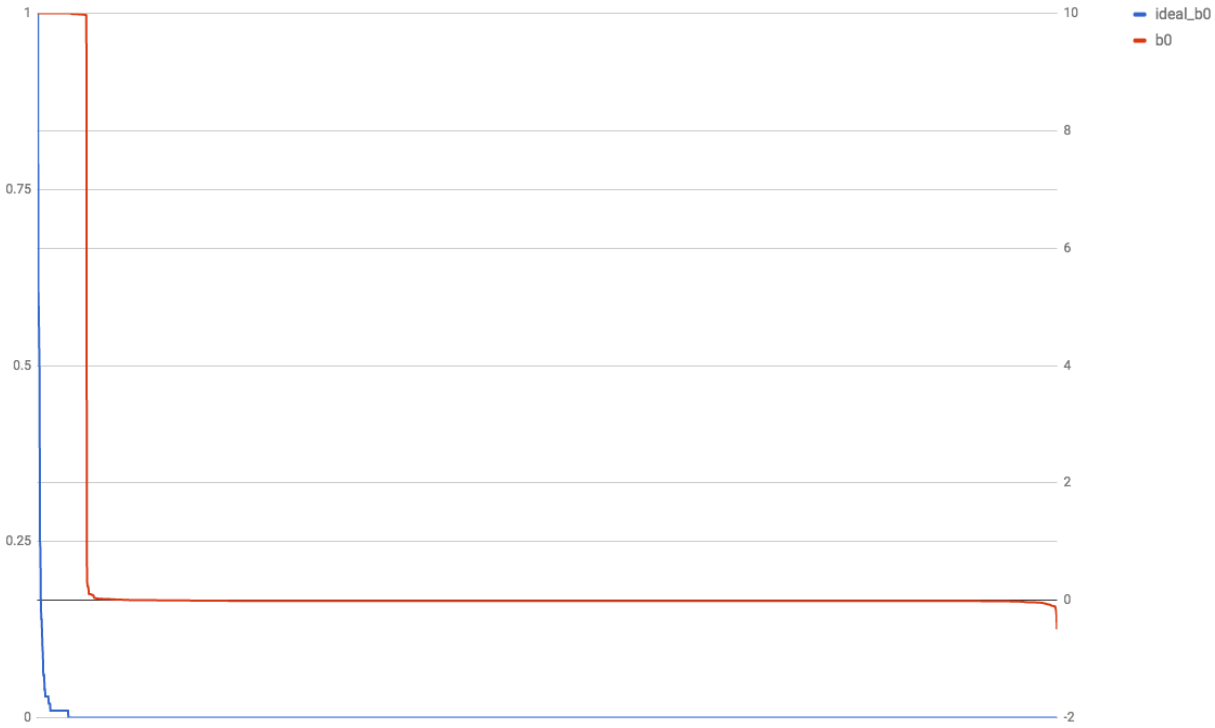
Offline zipline installation for some reason wasn’t able to load symbol(‘SPY’) or sid(8554). So model was retrained on GOOG as a third input

Sample result for portfolio evaluation using zipline using daily data:

Learned vs ideal signals (sorted by learned signal value):



Buy (learned vs ideal):



Ideal Strategy

In any case we need an ideal strategy to compare with what we get from the network.

For now there are two signals: buy and sell.

On the historical data we can have ideal strategy within some “prediction_frame”. On the lowest price of that frame there should be a buy signal, on the highest price there should be sell signal.

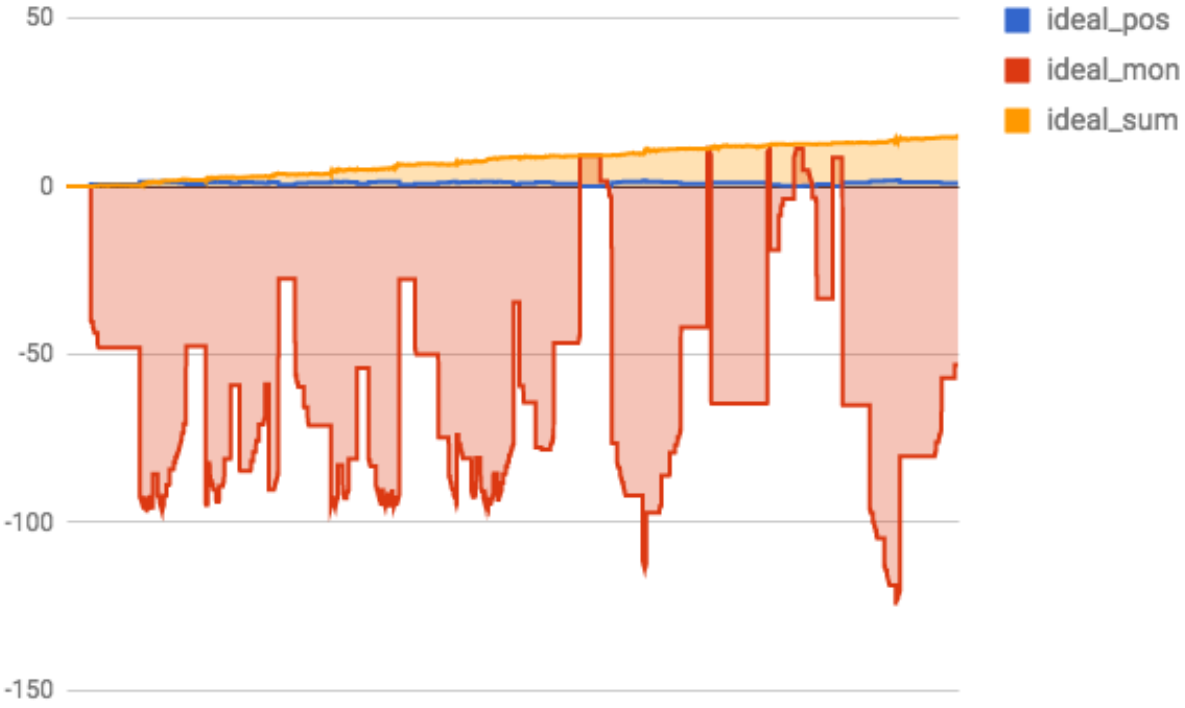
Code for ideal strategy for network to learn:

```
 for i in range(len(self.rawData) - prediction_frame):
    minIndex = maxIndex = i
    for j in range(i, i + prediction_frame):
        if self.rawData[j] < self.rawData[minIndex]:
            minIndex = j
        if self.rawData[j] > self.rawData[maxIndex]:
            maxIndex = j
    self.signals[minIndex][0] += 1.0
    self.signals[maxIndex][1] += 1.0

 min_sig = 0.0
 max_sig = 0.0
 for i in range(len(self.rawData)):
    min_sig = max(min_sig, self.signals[i][0])
    max_sig = max(max_sig, self.signals[i][1])

 for i in range(len(self.rawData)):
    self.signals[i][0] *= 1.0 / min_sig
    self.signals[i][1] *= 1.0 / max_sig
```

Here is an ideal strategy sample



- Blue is purchased equities.
- Red is cash.
- Yellow is overall sum of purchased equities (at the current price) and cash balance, assuming all orders will be fulfilled. Which is probably reasonable, because close prices are rarely touching high and low of the period
- Scale is abstract on all the charts. Every strategy could be scaled up or down.
- Prediction frame is 400 (slightly more than a day for minute records)

Model 16

Model 16 is an attempt to reduce network size to something exportable to quantopian.
Still in progress.

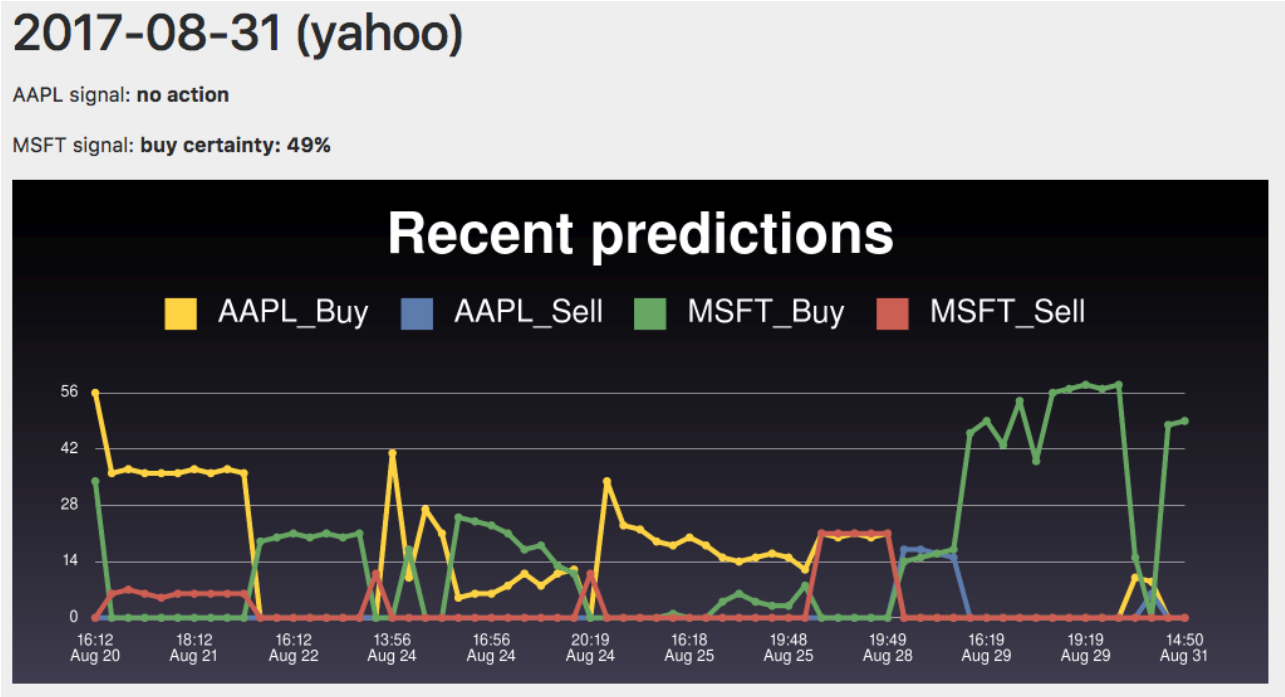
Standalone Server

Standalone server is providing daily signals based on several trained models

Server itself is based on Sinatra, model layers are exported to json files. Model evaluations are performed every hour on the data loaded from Yahoo finance.

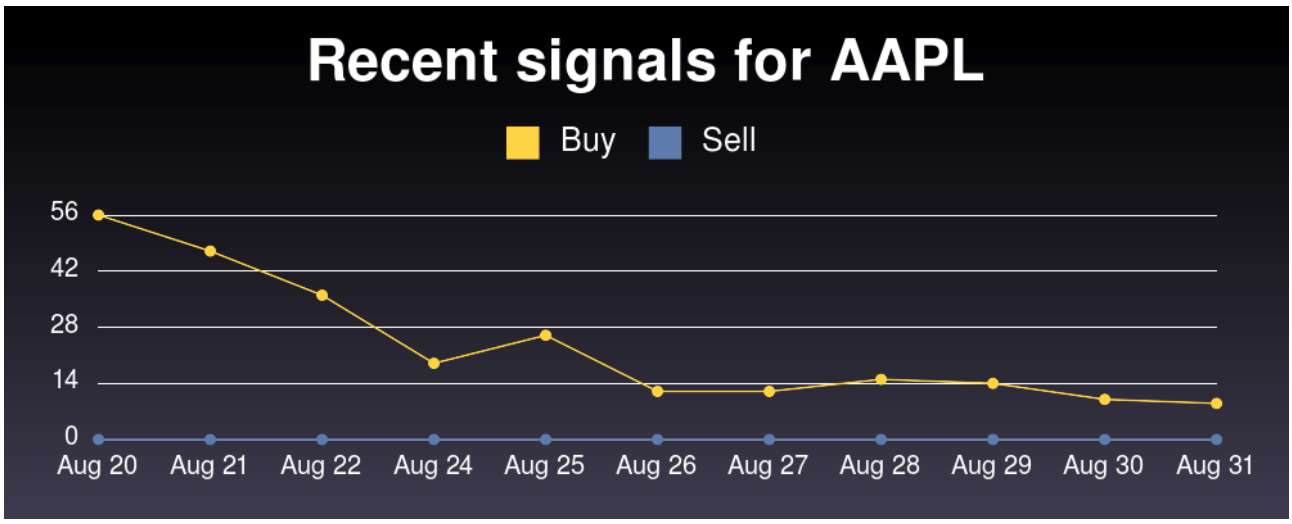
In addition to model 15 described above, there is model 17 trained on bigger data set, model 18 trained on different stocks and model 19 trained on close prices and volumes.

Since close price is changing during the day, model output is changing too:



Time is local server time. Yahoo data is delayed.

Predicted signals are averaged by date. For instance, model 15 predictions for Apple



Left axis is in percents of buy/sell certainty. Model output is filtered to remove chatter and output percents are just what stock indicator outputs are: suggestions.

These daily average charts show another interesting thing. Every day input is shifted one step, so each value goes through different route inside the network and yet predictions are going up and down rather smoothly.

Model Analysis

In order to separate better trained signals from false signals we need to analyze model behaviour. For instance, there could be a group of signals that are too hard to generalize. To optimize learning we need to get rid of such signals.

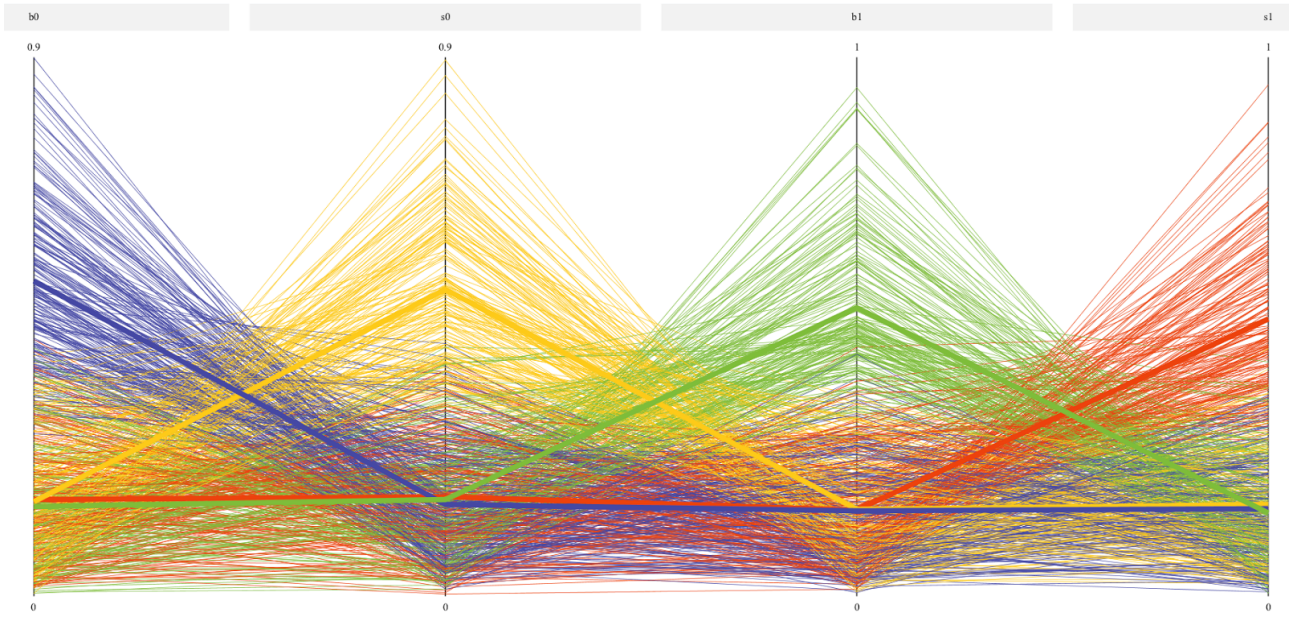
To classify signal learning qualities we can use some kind of clustering. The problem is that the input space is too big to enumerate. It's at least 3 to the power of seq_len*3, where seq_len is network input length for a stock. There are several possible workarounds:

1. Analyze network structure instead of inputs
2. Create random chart samples to cover some tiny percentage of input space

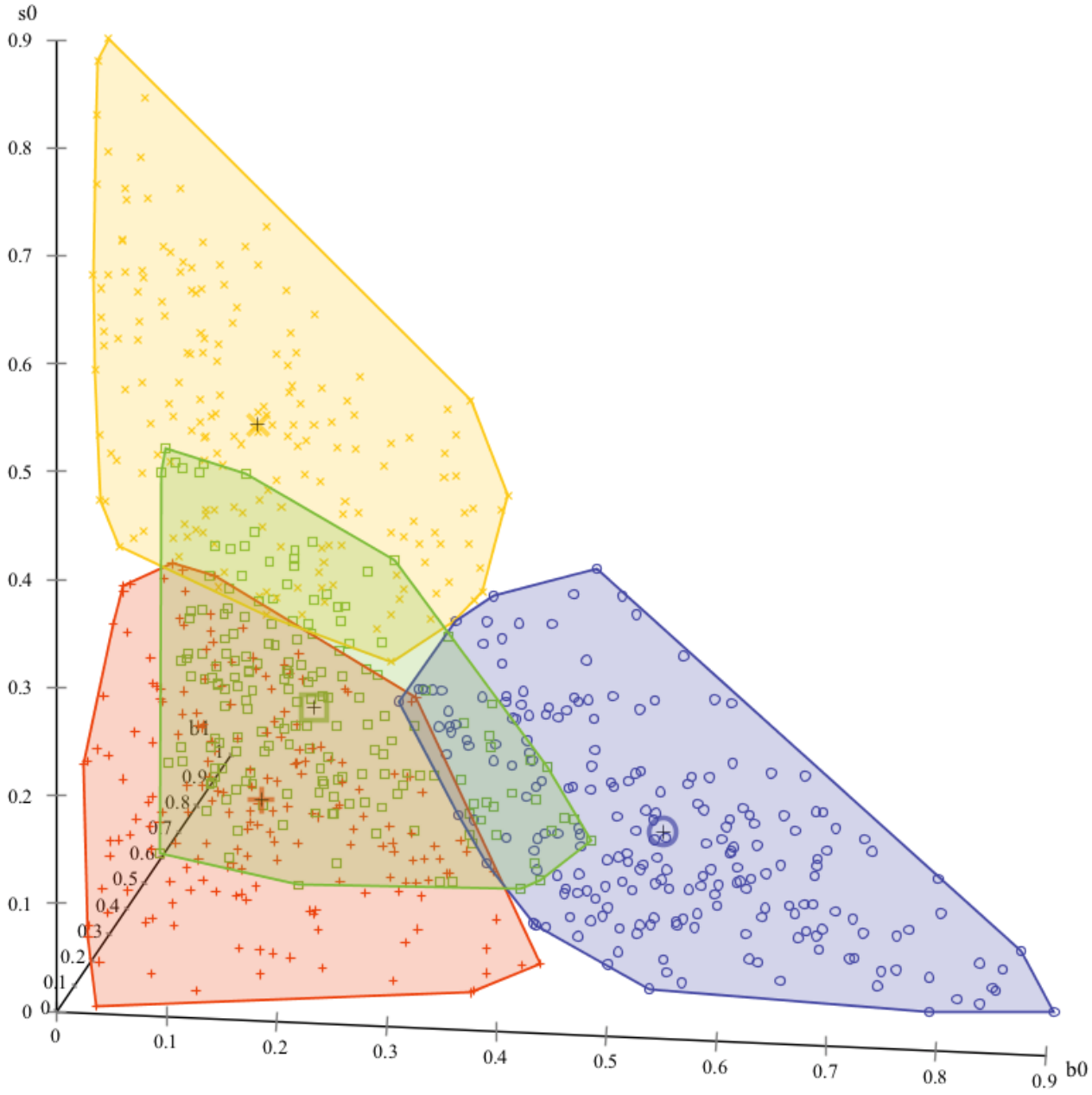
The problem with second option is that random data tend to stay near the averages and don't have trends that network could extract, so purely random data won't cover actually working parts.

Layers output clustering

Output layer clustering (using ELKI):



Buy/sell signals for 2 securities based on one-hot input of the second layer



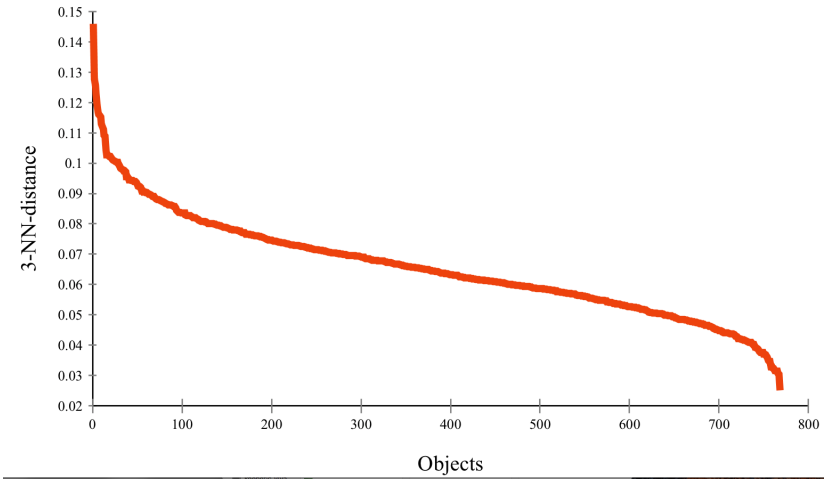
These clustering diagrams show that there is no direct correlation between stock signals.

Input clusterization with subspace methods (to reduce number of dimensions to something presentable) aren't giving any results yet.

Input intrinsic dimensionality is around 10:

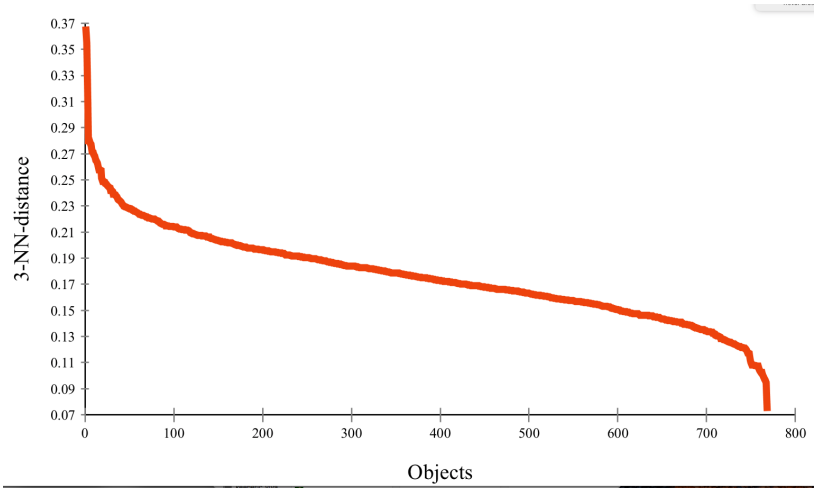
de.lmu.ifi.dbs.elki.algorithm.statistics.EstimateIntrinsicDimensionality.intrinsic-dimensionality: 9.9387467857533

Input distance distribution (for 3 nearest neighbours)



This chart is showing that most of neurons act similar to one-hot input.

Random distribution for same structure looks like this:

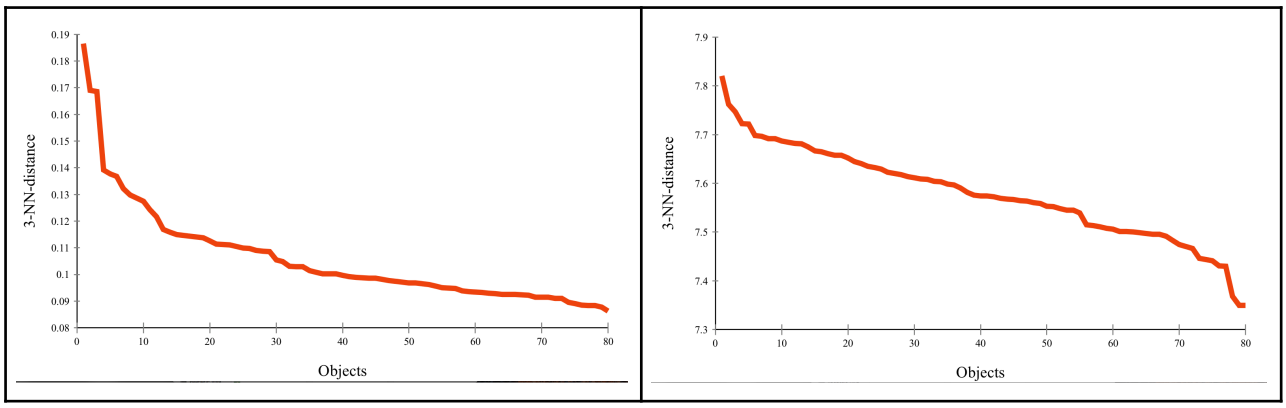


This means that we have a lot of rather tightly grouped neuron responses in the input layer. Around 2 times more tightly packed than a random distribution.

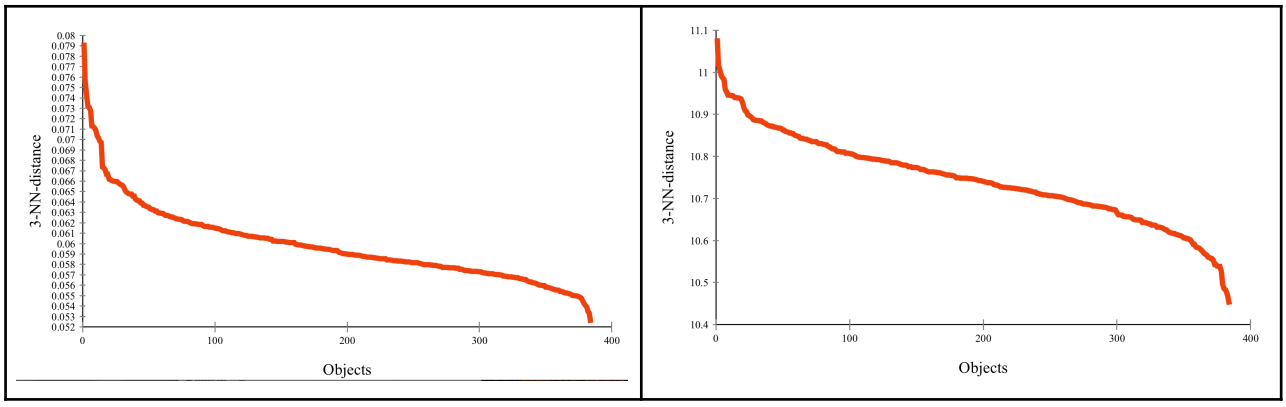
DBSCAN and other methods are showing only one cluster:

This is not a bad sign, because it means that input layer don't have real preferences on filtering input data and probably has more or less gaussian distribution.

Visually worse situation happens in hidden layers.



First hidden layer distance distribution (on the left) shows very tightly packed responses compared to the random sample distribution on the right. Similar situation is for the second hidden layer:



This might mean that means that network is trying to compensate for shifting input signals in all of it's layers.

So there is a room for improvement: we can replace input part of the network by a bunch of 1D convolution filters.

On the other hand, softmax logistic function could cause this visual grouping to some degree.

Framework

- Python + Tensorflow.
- Tested network kernels: BasicLSTMCell, perceptron with 2 hidden layers
- Tested activation functions for RNN: Linear (discarded), sigmoid
- Tested activation functions for perceptron: RELU, softmax
- Tested loss functions for RNN: Softmax cross entropy with reduced mean (discarded), absolute difference

- Tested loss functions for perceptron: absolute difference
- RNN optimizer: gradient descent optimizer
- Perceptron optimizer: Adam optimizer

To be tested/implemented:

- back propagation for an optimizer
- run learning in batches with similar signals (requires signal classification)
- different activation functions
- different kernels
- filter out ideal signal noise
- check which signals could be learned best (using some kind of classifier), reduce learning to only these signals
- increased input sequence size, increased/decreased number of hidden neurons
- add volume to the input data (model 19 on the standalone server)
- loosening ideal strategy by moving ideal buy/sell events a step forward or backwards or increasing tolerance for the loss function
- replacing learning inputs with a “one-hot” arrays instead of partial signals