Machine Learning Nano Degree

Capstone Project

Johnny Ward

Stock Price Predictor

Introduction

In this project I apply machine learning to the problem of predicting changes in the stock price of companies listed on significant stock exchanges, such as the NASDAQ, Dow Jones, and FTSE.

The problem is interesting because an ability to accurately predict the movement of stock prices would enable one to make profitable trades. The problem is difficult because the movement of prices is not deterministic. However much is known about the present state of the market, the company in question, and the past performance of the market and company stock, no model will be able to predict exactly what will happen in the coming weeks, or even days. Stock prices are determined by a wide spectrum of factors: at one extreme are matters internal and specific to the company, such as the appointment of a particular executive; at the other extreme, macroscopic events whose effects go well beyond the price of stocks can have significant effects: the Japanese Tsunami, for example, or a war. Factors such as consumer demand for particular products, political events, and seasonal cycles lie in between.

It is clearly not feasible to model all factors having an effect on stock prices. Instead, what is attempted is a technical analysis, which does not directly seek to model anything other than the changes over time to the stock price of a particular company, and the exchange on which it is traded.

Problem Statement

This report describes a model, developed using Python and machine learning libraries, which is useful in predicting the movement of stock prices.

The model:

- Retrieve information about a particular stock, and the exchange on which it is listed, from the Yahoo Finance API, including the adjusted closing price for a given period;

- Predict the price of that stock on a given date in the future, or at least indicate whether the stock is likely to have risen or fallen by that date.

- On predictions looking 7 days ahead, be accurate to around 5% of true value, in 75% of cases.

Datasets and Inputs

The model uses the Yahoo Finance API to retrieve data about the past performance of stocks. The API provides data for a variety of exchanges, including the NASDAQ, Dow Jones, S&P500, FTSE100 and FTSE250. The data is freely available and in the public domain.

The data is downloaded at runtime. List of ticker symbols are downloaded and stored locally. During development, the model used data spanning a period which ended not later than three months before the present date, so that test data was available on which performance could be measured. In the deployment phase, the model retrieves data for a period ending on the last trading day, in order that predictions as to future movements can be as accurate as possible.

The data used is the adjusted closing price of a given stock, and of the wider market (i.e. the relevant exchange). The adjusted closing price takes account of divisions and other deliberate changes to stock prices, which, without appropriate treatment, could significantly hamper the learning task, by suggesting that the changes represented a true change to the value of the stock. From these metrics, other measures and features (such as rolling averages, momentum etc) are calculated, for varying periods. The precise combination of features, and the periods for which past data is useful, has been subject to analysis and tuning in the process of development, as described below.

Solution Statement

The model is developed in Python, and uses the following libraries:

Numpy (1.11.2)
Pandas (0.19.1)
Scikit-Learn (0.18.1)
Yahoo Finance (1.4.0)
Keras (1.2.0)
Tensorflow (0.12.1)

The model is not compatible with earlier versions of Pandas.

Overview of the Model

The model works slightly differently in development and deployment, largely because the former requires a window of time to be available for testing, which falls after the data on which the model is trained. The latter works on up-to-date information. The functionality and workings of the models are described in turn.

Development Phase

The model is contained within the Price_Predictor / Predictor classes (contained within the stock_predictor.py / lstm_classifier.py etc modules). A Price_Predicor instance takes the name of a stock, in the format of the ticker symbol by which it is represented on the exchange (e.g. "RBS.L" for Royal Bank of Scotland, on the FTSE100). The model then selects a random date (within the last six months), in order to ensure variation in training data, and queries YF for the name of the exchange on which it is traded. The model then fetches price data for both the stock and the market composite, for the two years preceding the date randomly selected. This data is stored in a Pandas DataFrame instance ("DF"), with the name of the stock, the dates, and the adjusted close prices on each trading day in the period for the stock and the market, representing columns.

The next step is to pre-process the data and derive additional features. Unfortunately, YF does not provide historic data for all the fields available for current queries. The method prepare_data() carries out this step, by a series of operations on the columns of the DF. The derived features include alpha, beta, momentum, market-relative price change, and r2. The logic for calculating these parameters was largely derived from "Calculating Stock Beta, Volatility, and More" by Gouthaman Balaraman (http://gouthamanbalaraman.com/ blog/calculating-stock-beta.html).

The model then extracts the target variable, which is the adjusted closing price seven days later, for each date for which data is available (base date and seven days later). The data is then scaled using the Sci-kit learn MinMaxScaler, which ….

Given that stocks are not traded every day, there are inevitable gaps in the DF. These gaps could either be filled – with the values from adjacent rows or the median for the column – or dropped.

The model then uses a regression algorithm to model the relationship between the price and derived features on the one hand, and the time-delayed price on the other. Alternatively, it can use a binary classifier, to specify whether the stock in question is likely to rise, or to fall (or remain stable), by the date specified.

The model was tested on the testing data, by taking dates from that data, calculating and separating out the time-delayed price, calculating the derived features as at that date, scaling the data, and using the regression algorithm to predict the price at the desired interval. The predicted price is then compared with the true price. Accuracy is calculated by $|p - a| / a$ where p = predicted price and a = actual price.

Data Exploration

The project does not employ a single, finite dataset. Instead, during training, for a given stock, data relating to the market on which that stock is traded is accessed, downloaded and processed. One of the interesting features of the problem is that there is a vast amount of data available, but there is a balance to be struck between using more data on the one hand, and avoiding stale, irrelevant data on the other. For example, to increase the data available for a given stock, we can look further into the past. But the further back one goes, the less likely it is that the data will be useful for predicting future movement. Accordingly, the amount of training data is a parameter to be explored and set during development.

A call to the YF API (Share.get_historical(start_date, end_date) yields a range of results, all of which relate to the price of the stock, save for one, which provides the volume traded on the day in question.

The data from the 'get_historical' call is refined to 'adjusted close' and 'volume', and then supplemented with a range of metrics calculated from these data.

In order to calculate some of these metrics, data about the overall market (i.e. the exchange on which the target stock is traded) will need to be accessed and processed. For example, for 'VOD.L' (Vodafone), the FTSE 100 index (YF: ^FTSE) will be required.

In the case of any stock, the (useful) data available will be limited to the days on which the relevant market was open, and the stock was traded. Indeed, the YF API does not recover information for non-trading days.

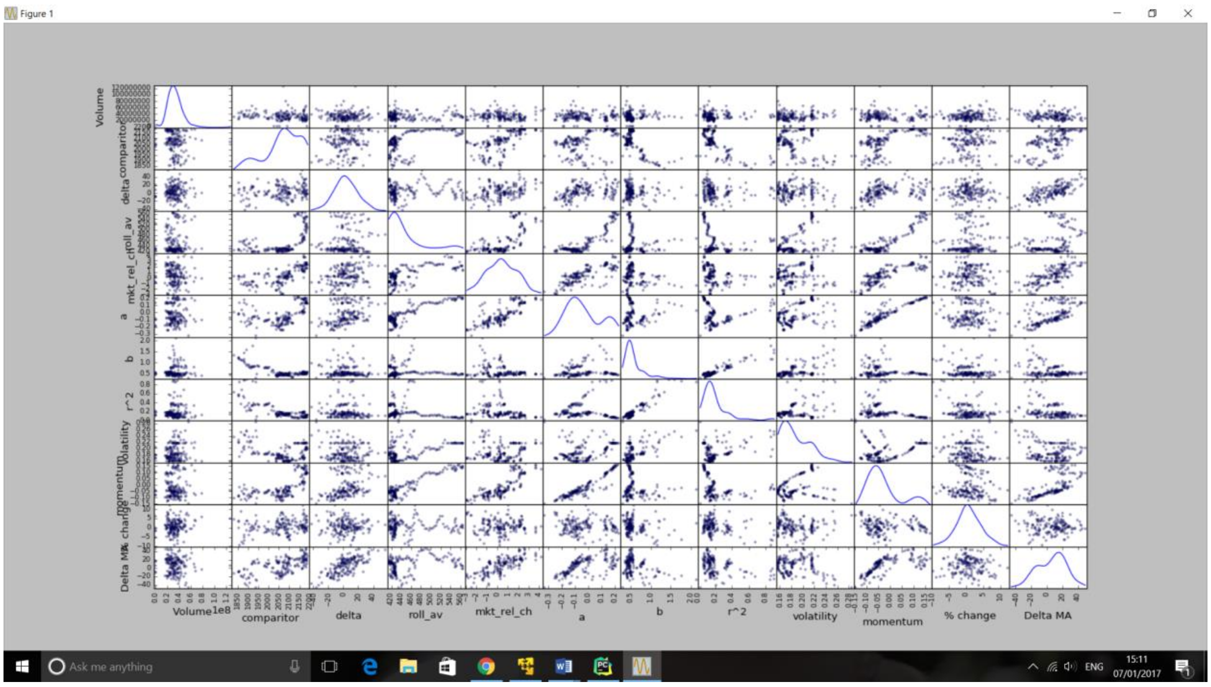
In calculating metrics, such as market-relative price change, or 30-day rolling average, the requisite data spans periods including non-trading days. It is therefore necessary to decide how to treat missing data. This may change, depending on the data in question. One solution would be to ignore non-trading days, so that a 30-day rolling average was actually the average over 30 trading days. Alternatively, such days could be taken into account, such that the actual number of days' data used would vary depending on the season. This may have some impact on the ability to model future behaviour.

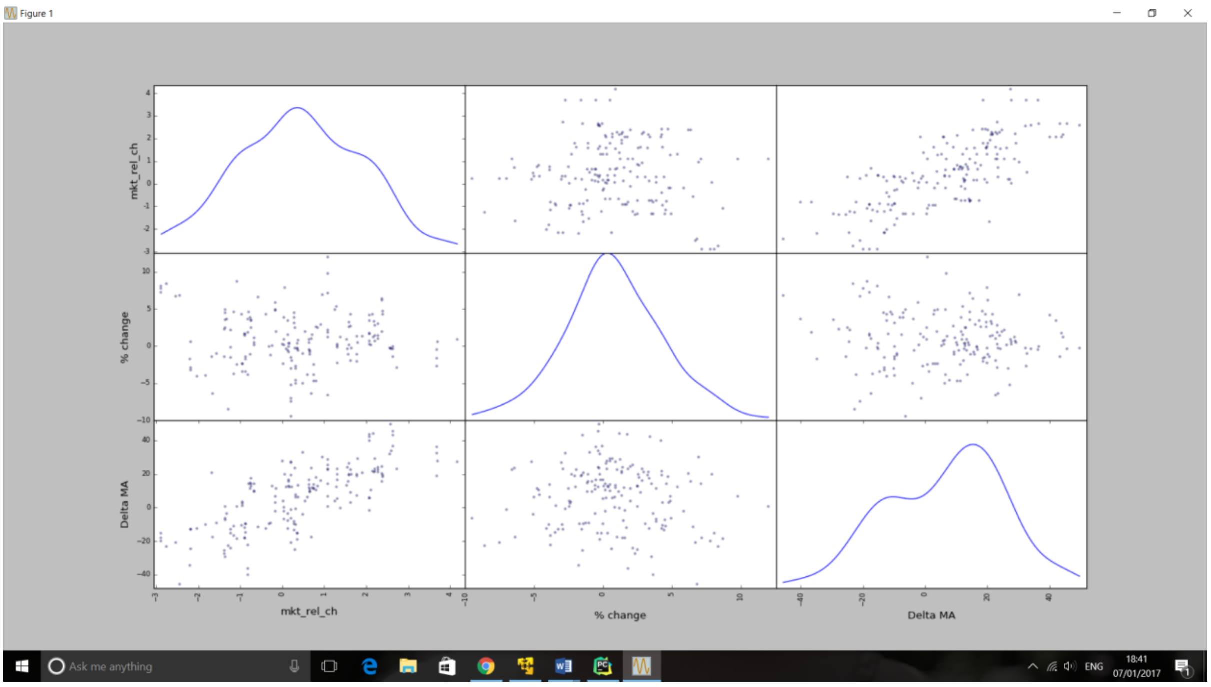
Another solution would be to pad the data with results from surrounding days – either before, after, or by some calculation (such as the median for the relevant metric). Again, this would have the potential to skew results because the perceived volatility of the stock would vary as a function of the number of non-trading days in the training period.

On reflection, it appears that the best solution would be to ignore non-trading days, so that when rolling averages, momentum and similar metrics are calculated, they are always calculated on the same number of trading days. This will obviously have the effect of extending the period over which the metric is calculated in holiday periods, but I find this to be preferable to padding the data, particularly where the overall number of data points is relatively small.

There may well be other missing data points, as a result of transcription error rather than the close of markets. In such cases it may seem more sensible to simply fill the gap with the previous, next, or composite of previous and next days' data. However, one reason for the absence of data may be the suspension of trading on a particular stock, which would obviously be a significant event and one that should not be glossed over with replacement data. The chosen solution is therefore to remove all data points where the 'adjusted close' and/or 'volume' metrics are not recoverable from the YF API.

The following figures illustrate the relationships between the various variables. As can be seen, none of the variables appear to be strongly correlated with the change in price over the following seven days. The most promising correlation seems to be between market-relative price change, and deviation from the moving average of the stock's price. Most of the variables appear to follow a normal distribution, although there is some skewing. I also plotted a scatter matrix of the log-data, and larger plots of certain features.





The weak correlations exhibited in the scatter matrices mean that the prospects of learning a reliable model from this data are not good. However, since we have ready access to a wealth of time-series data, it would be worth applying a model that can learn from data presented in a sequential manner, to see if future behaviour can be predicted from time-series patterns in the past.

Benchmarking

Given that stock prices do not, generally speaking, increase or decrease by orders of magnitude in the space of a week, it is reasonable to assume that the rolling average of the stock's adjusted closing price, taken over the previous (say) 14 days, would provide a reasonable basis for estimating the closing price of the stock 7 days later. In order for a learning algorithm to be of any assistance, it will need to perform better than a model which predicts stock movement solely by reference to the rolling average of the price of the stock in question.

The Learning Algorithms

As alluded to above, two broad approaches were taken to modelling the data. Firstly, a range of regression algorithms and simple neural networks were applied to a data set containing 'snapshots' of given features on consecutive days in a defined period. Secondly, a Long-Short Term Memory ("LSTM") recurrent neural network was applied to a limited subset of the data, in windows of time-steps. These approaches are described in turn below.

Multiple-Feature Models

Once the above features have been derived, the dataframe is refined so as to drop a number of features. The remaining dataframe consists of the following columns: date, adjusted close, alpha, beta, r-squared, momentum, volume, market-relative price change, and price seven days later.

In order to further prepare the data for learning, the target feature – price seven days later – is removed and stored separately. The remaining data is then scaled using the Scikit-Learn MinMixScaler. The purpose of the scaling is to ensure that features which are likely to have large values (such as volume) are not treated as more significant in the learning process than those which are likely to be small (such as alpha), merely because of their relative size. The scaler assigns new values to each data point, in the range 0 – 1, in proportion to the relative magnitude of that data point in comparison to the other data in the training set. The MinMaxScaler uses the formula

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

(see http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html).

The data is now in the form of a scaled dataframe containing the features mentioned above, and a pandas series object representing the target variable, the price seven days after the day to which the other data relates. The algorithm is constructed in such a way that the features, scaling, and learning algorithm can easily be adjusted, for the purposes of experimentation and/or adaptation. By way of example, it is trivial to swap a column in the 'X' dataframe – replacing momentum with monthly range of prices, say – or to add or remove particular features. Similarly, the scaling can easily be removed, or swapped for an alternative formula or implementation. The range of dates on which the model is trained can easily be adjusted, and the learning algorithm can be swapped by importing and applying a different module from Scikit-Learn, or elsewhere.

The following learning algorithms were tested on the data:

- Support vector machine (sklearn.svm.SVR)
- Random forest (sklearn.ensemble.RandomForestRegressor)
- Gradient boosting regressor (sklearn.ensemble.GradientBoostingRegressor)
- Multi-layer perceptron (of my own design: github.com/lemontrachet/MLP_in_Python3)
- Ensemble of weak-learning MLPs (my own: as above)

The best results were achieved with the Gradient Boosting regressor ('GBR'). The GBR is an ensemble model, which means that it trains a specified number of models on random subsets of the overall data available (bagging). Rather than seeking to achieve a high degree of accuracy (recall) on the training data with a single model, it aims to maximise performance on the unseen test set by training a large number of models to a moderate extent (hence the name 'weak learners'). The predictions from these sub-models are then aggregated and combined so as to predict values for unseen data. The strength of ensemble models is that they tend not to overfit the training data, because none of the sub-models is allowed to achieve a strong fit on its subset of the data, and because the sub-models are generally not trained on all of the available training data.

Scikit-Learn also provides tools for tuning and optimising learning algorithms, such as GridSearchCV, which, given a list of hyperparameters, systematically trains and tests the selected learning algorithm on training and validation sets extracted from the data, with all combinations of the hyperparameters, to discover the optimum. In this case, because there is no single dataset, but instead a range of datasets for each one of thousands of different stocks, which have different sizes depending on the date range and feature extraction which is applied (which can themselves be considered parameters of the learning model), and because the training and testing process takes about 60 seconds (Dell XPS 15 Windows 10 64-bit 8-core 2.6GHz 16GB RAM) I decided to write a bespoke testing algorithm to take advantage of all of the available cores. Using the ProcessPoolExecutor from the concurrent futures module, I defined a range of hyperparameters to tune the model, and then ran eight models, with different combinations of hyperparameters, in parallel, for a range of different stocks, and dates. Sample output from these test runs is set out below.

0.6666666666666666, GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None, learning_rate=0.1, loss='huber', max_depth=4, max_features=3, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=50, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=True)),	0.7246376811594203, GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None, learning_rate=0.1, loss='huber', max_depth=5, max_features=3, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=50, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=True)),
0.6376811594202898, GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None, learning_rate=0.1, loss='huber', max_depth=5, max_features=5, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=50, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=True)), warm_start=True)),	0.6376811594202898, GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None, learning_rate=0.1, loss='huber', max_depth=4, max_features=3, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=False)), warm_start=False)),

0.6521739130434783, GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None, learning_rate=0.1, loss='huber', max_depth=5, max_features=3, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=True	

Ultimately, I found the following combination to produce the best results on the stocks that I used for tuning (VOD.L, GOOG, AAPL):

```
loss='huber'
n_estimators=250
max_depth=8
max_features=3
warm_start=True
learning_rate= 0.15
subsample=0.95
alpha=0.9
max_leaf_nodes=10
min_samples_leaf=5
```

Minimal-Feature, Time-Series Models

I applied a variety of LSTM-based recurrent neural networks to the data. In order to prepare the data for time-series learning, I modified the program described above so as to produce a dataframe containing the adjusted closing prices for a given stock over a given period, together with the rolling average price, and the ratio of the price to the overall price of the market. The data is then batched into sequences of length *n* (a range of lengths was tried), such that for each date, the batch contains the prices for that date, or one of the other data features, and the equivalent data for the *n* preceding days. A series of data is derived, representing the adjusted closing price of the stock a given number of trading days after the end-date of the associated batch (for example, for day D, the time-series batch would consist of closing prices or other data for D, D-1, D-2, D-3, D-4, D-5 and D-6, and the associated (target) data would be the closing price for D + n, where n is the number of trading days between D and the date for which a prediction is sought. For the purpose of testing, I used n = 6, which is roughly equivalent to 7 days, taking account of non-trading days, and depending on the time of year.

The data is then split into training and test sets, largely as described above. As the data is time-series data, it is not shuffled. It is split 75:25 such that smaller, later tranche follows directly from the larger, earlier tranche. None of the test data is seen by the network during training. The data is then scaled using the MinMaxScaler described above.

The next stage was to design the recurrent neural network. This topic was not covered in the Machine Learning Nano-Degree lectures at the time that I took them, and so I referred to the Keras documentation (https://keras.io/), and tutorials at http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/ and http://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

LSTMs are particularly good at modelling time-series information, or indeed any sequences where the likely next item or items depend to some extent on the previous items in the sequence, such as in language or genetics. One of the central advantages of LSTMs over standard RNNs is their ability to model relationships between remote items in sequences, where several time steps separate the two. They get around the 'vanishing gradient' problem by incorporating a series of gates (input, output and forget), and storing an internal state which is only exposed to the subsequent layers of the network if the weights associated with the relevant gate are sufficiently strong. The weights relating to all of the gates are exposed to the input to the LSTM unit, and through iterations of learning the unit can be adapted so as to model long-range dependencies.

The network uses the Keras module and TensorFlow backend. I experimented with a range of LSTM units and layers, and regular, connected (Keras 'Dense') layers, activations, dropouts, and other hyperparameters. The final configuration of the model was as follows:

```
model = Sequential()
model.add(LSTM(25, input_dim=1))
model.add(Dropout(0.2))
model.add(Dense(30, activation='tanh'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error', optimizer='rmsprop')
model.fit(X_train, y_train, nb_epoch=50, batch_size=1, verbose=1)
```

After training the network on the training data, it is used to make predictions on the training and testing data. The training and testing predictions and then compared with the target data (which, in the case of the training target data, the model has already seen), and the results are collated. I used the root mean squared error (from sklearn.metrics) and abs(prediction – actual) / actual to calculate the error.

Testing

The following are the results of a series of predictions for the 'GOOG' stock on various dates, using the Gradient Boosting Regressor, and the LSTM network. 'Within 5%' refers to the proportion of the predictions which achieve an accuracy of 5% of the true value, or better:

GBR: within 5%: 0.647, RMSE: 36.631 (68 predictions)
LSTM: within 5%: 0.85, RMSE: 18.977 (89 predictions)

Similarly, for 'GLEN.L':
GBR: within 5%: 0.014, RMSE: 83.06 (70 predictions)
LSTM: within 5%: 0.620, RMSE: 14.99 (92 predictions)

Finally, for 'HSBA.L':
GBR: within 5%: 0.03, RMSE: 77.52 (71 predictions)
LSTM: within 5%: 0.815, RMSE: 18.758 (92 predictions)

The following is a random selection of 49 stocks listed on the London FTSE:
BARC.L', 'BATS.L', 'BDEV.L', 'BKG.L', 'BLND.L', 'BLT.L', 'BNZL.L', 'BP.L', 'BRBY.L', 'GKN.L', 'GLEN.L', 'GSK.L', 'HIK.L', 'HL.L', 'HMSO.L', 'HSBA.L', 'IAG.L', 'III.L', 'IMB.L', 'INF.L', 'INTU.L', 'ITRK.L', 'MNDI.L', 'MRW.L', 'NG.L', 'NXT.L', 'OML.L', 'PFG.L', 'PPB.L', 'PRU.L', 'PSN.L', 'PSON.L', 'RB.L', 'RBS.L', 'SAB.L', 'SBRY.L', 'SDR.L', 'SGE.L', 'SHP.L', 'SKY.L', 'SL.L', 'SN.L', 'SSE.L', 'STAN.L', 'STJ.L', 'SVT.L', 'TPK.L', 'TSCO.L', 'TUI.L'

Both models were trained on data for each of the above stocks. In the case of the LSTM model, only the adjusted closing price (batched into 6-day series) was used to train the model and make predictions. Predictions were also derived based solely on the 7-day rolling average of the stock price (i.e. the predicted price is simply the rolling average price taken 7 days before the predicted date). The results appear below.

	Mean Error		**RMSE**		**Within 5% of True**		**Baseline (7-day rolling average within 5% of True)**
Stock	**GBR**	**LSTM**	**GBR**	**LSTM**	**GBR**	**LSTM**	
BARC.L	0.21	0.02	51.6	6.93	0.10	0.90	0.910
BATS.L	0.04	0.03	254.8	113.3	0.63	0.79	0.934
BDEV.L	0.04	0.03	20.1	19.7	0.57	0.78	0.746
BKG.L	0.05	0.04	151.1	115.4	0.53	0.64	0.803
BLND.L	0.04	0.03	26.1	33.1	0.80	0.72	0.885
BLT.L	0.21	0.16	281.1	261.9	0.00	0.52	0.590
BNZL.L	0.05	0.02	113.6	49.5	0.56	0.97	0.975
BP.L	0.12	0.03	61.4	14.7	0.03	0.77	0.844
BRBY.L	0.10	0.08	151.9	143.2	0.08	0.20	0.893
GKN.L	0.03	0.06	13.8	21.6	0.69	0.32	0.902
GLEN.L	0.27	0.05	78.4	16.639	0.01	0.46	0.700

GSK.L	0.05	0.03	81.2	56.3	0.80	0.75	0.910
HIK.L	0.14	0.04	271.50	105.1	0.13	0.67	0.770
HL.L	0.06	0.06	76.0	86.6	0.38	0.61	0.730
HMSO.L	0.03	0.03	17.7	21.8	0.74	0.80	0.861
HSBA.L	0.13	0.03	89.4	18.8	0.03	0.81	0.918
IAG.L	0.10	0.04	49.8	25.6	0.21	0.71	0.811
III.L	0.10	0.09	0.66	0.47	0.28	0.36	0.472
IMB.L	0.05	0.03	206.9	133.2	0.59	0.68	0.893
INF.L	0.04	0.03	32.6	17.0	0.63	0.82	0.960
INTU.L	0.06	0.04	18.8	16.0	0.29	0.62	0.861
ITRK.L	0.05	0.02	189.3	65.7	0.70	0.93	0.934
MNDI.L	0.02	0.03	40.4	59.2	1.00	0.78	0.762
MRW.L	0.08	0.04	19.4	8.07	0.14	0.66	0.780
NG.L	0.06	0.02	68.7	23.2	0.56	0.89	0.951
NXT.L	0.05	0.02	291.1	188.4	0.39	0.92	0.984
OML.L	0.03	0.08	6.39	18.0	0.87	0.28	0.877
PFG.L	0.03	0.03	117.0	103.5	0.84	0.80	0.885
PPB.L	0.04	0.07	383.6	684.2	0.79	0.47	0.795
PRU.L	0.11	0.02	212.7	40.9	0.20	0.90	0.926
PSN.L	0.03	0.06	69.6	106.5	0.76	0.50	0.754
PSON.L	0.04	0.07	37.9	92.1	0.50	0.27	0.885
RB.L	0.02	0.02	149.8	120.2	0.93	0.99	0.983
RBS.L	0.05	0.03	13.0	13.7	0.60	0.79	0.852
SAB.L	0.02	0.02	125.6	96.3	1.00	0.90	0.896
SBRY.L	0.04	0.02	11.3	6.82	0.76	0.92	0.820
SDR.L	0.07	0.02	232.2	82.4	0.24	0.93	0.910
SGE.L	0.06	0.02	44.0	17.7	0.73	0.76	0.852
SHP.L	0.06	0.03	335.6	229.3	0.40	0.75	0.787
SKY.L	0.07	0.03	70.7	34.15	0.31	0.89	0.910
SL.L	0.04	0.06	18.3	56.5	0.53	0.65	0.795
SN.L	0.04	0.02	57.4	24.5	0.69	0.99	0.975
SSE.L	0.02	0.03	41.8	52.7	0.92	0.82	0.770
STAN.L	0.07	0.14	48.7	141.1	0.40	0.00	0.811
STJ.L	0.03	0.03	46.6	35.0	0.63	0.74	0.803
SVT.L	0.05	0.03	136.7	78.2	0.39	0.61	0.926
TPK.L	0.11	0.02	166.1	55.6	0.11	0.95	0.951
TSCO.L	0.13	0.04	26.7	12.3	0.00	0.61	0.754
TUI.L	0.04	0.03	51.5	36.0	0.72	0.83	0.754

As can be seen, the GBR model did not perform well and, although the accuracy of the LSTM-based model's predictions is generally good, it does not offer any advantage over making predictions based only on the rolling average of the price of the given stock. Indeed, the rolling average is in the majority of cases the best guide to the likely price in 7 days' time. This comes as little surprise, given the apparent lack of correlation between the metrics and the target feature, discussed above.

For this reason, I considered an alternative approach, based on the premise that it is more useful to know whether a stock price is likely to rise or fall than to know, within 5% or so, the likely price of the stock. On this basis, I tweaked the LSTM model so as to provide classification, rather than regression, output, with '1' signifying a rise in the price and '0' signifying a fall, and combined the available features into a single model.

Again, I tweaked the model until it produced satisfactory results. It was notable that unless the network is constructed with a substantial number of units (both LSTM and standard perceptron (Keras 'Dense') units), it was incapable of producing varying outputs, and would simply return all '1's or all '0's. Because of hardware limitations, and the time taken to train multi-layer LSTM networks, I was unable to test as many feature / parameter / network combinations as I would have liked, and I expect that further performance gains could be made from further experimentation and refinement.

The final configuration adopted was as follows:

```
model = Sequential()
   model.add(LSTM(
      75,
      activation='sigmoid',
      input_shape=(X_train.shape[1:]),
      dropout_W=0.15,
      return_sequences=False))
   model.add(Dropout(0.2))
   model.add(Dense(250, activation='sigmoid'))
   model.add(Dense(1, activation='sigmoid'))
   model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
   model.fit(X_train, y_train, nb_epoch=100, batch_size=1, verbose=0)
```

The results produced by the LSTM classifier model on the stock data set out above are reproduced below.

Stock	LSTM Accuracy	Baseline Accuracy
BARC.L	0.777	0.439
BATS.L	0.682	0.548
BDEV.L	0.586	0.478

BKG.L	0.554	0.497
BLND.L	0.841	0.605
BLT.L	0.707	0.471
BNZL.L	0.573	0.561
BP.L	0.796	0.618
BRBY.L	0.701	0.465
GKN.L	0.777	0.592
GLEN.L	0.764	0.490
GSK.L	0.828	0.529
HIK.L	0.726	0.529
HL.L	0.771	0.503
HMSO.L	0.701	0.592
HSBA.L	0.682	0.459
IAG.L	0.777	0.548
III.L	0.569	0.507
IMB.L	0.541	0.554
INF.L	0.484	0.548
INTU.L	0.809	0.522
ITRK.L	0.682	0.529
MNDI.L	0.554	0.611
MRW.L	0.841	0.484
NG.L	0.667	0.490
NXT.L	0.758	0.389
OML.L	0.701	0.510
PFG.L	0.567	0.490
PPB.L	0.790	0.618
PRU.L	0.662	0.465
PSN.L	0.713	0.567
PSON.L	0.701	0.637
RB.L	0.548	0.548
RBS.L	0.618	0.459
SAB.L	0.519	0.538
SBRY.L	0.828	0.592
SDR.L	0.605	0.414
SGE.L	0.800	0.624
SHP.L	0.828	0.503
SKY.L	0.905	0.618
SL.L	0.590	0.401
SN.L	0.771	0.586
SSE.L	0.822	0.586
STAN.L	0.611	0.490

STJ.L	0.554	0.541
SVT.L	0.764	0.580
TPK.L	0.720	0.535
TSCO.L	0.822	0.503
TUI.L	0.737	0.492
Mean	0.700	0.528

In contrast to the results for the regression models, the LSTM-based classification model consistently out-performed the baseline, rolling average model. Further, whereas the rolling average accurately predicted the direction of movement of the price in about half of cases (0.528/1), and is therefore no better than simply guessing, the LSTM-based model did so in 0.700/1 cases. Accordingly, assuming that sufficient funds were invested, in a range of the stocks the subject of the tests above, the model could be expected to deliver consistent returns, even taking account of brokerage fees, which are unlikely to exceed approximately 20 GBP per trade, if sufficient volumes are involved.

Deployment Phase

I decided to deploy the algorithm as part of a Twitter-bot, named Giles (@stockwizgiles) ('SWG'). SWG runs on a Raspberry Pi. The code is on github.com at https://github.com/lemontrachet/brokerbot. It consists of a Twitter-bot module together with a stock-broker module. The twitter module consists of a listener / responder class, which is a sub-class from the TwythonStreamer class in the Twython module, together with an infinite loop which runs various functions, deploying instances of the Twython class where necessary. The broker module (stock_broker.py) contains a class (Broker) which is able to handle multiple accounts. It keeps a ledger in csv format containing the data for all accounts that the class holds. It has methods such as checking and adding to the balance of the account(s), retrieving predictions for stocks, and managing portfolios by buying and selling, either based on predictions or simple rules (e.g. buy where a stock has fallen below its 3-month rolling average and starts to climb again).

The Broker object is able to take advantage of the stock-predicting algorithm. This is run (stock_predictor.py) on an AWS EC2 Ubuntu instance. It is called from survey_market.py, which also runs on the EC2. The latter randomly selects stocks from the FTSE100 index (60 at a time) several times per day, and then runs stock_predictor.py across several threads, for maximum efficiency. In fact, since the EC2 instance I have is single-core, only the I/O is achieved in parallel. The results are saved in CSV format, and retrieved by the Broker object (by SSH) when it needs access to the predictions. By having the algorithm run several times per day, SWG is able to access up-to-date predictions without undue delay, for the majority of the listed stocks.

If SWG receives a tweet which mentions a ticker symbol (whether on the FTSE100 or otherwise) it will respond with a 7-day forecast. It will first check whether survey_market.py has generated a prediction for the stock in question, that day. If not, it will (through a Broker instance) run the stock_predictor.py algorithm locally for that stock, and tweet back the prediction.

SWG also trades on its own account. The state of his account can be seen on the profile page (@stockwizgiles). In total, I have put in 40,000 (fictional) GBP. At the time of writing his portfolio is worth 42,843 GBP.

Evaluation

It is fair to say that neither of the regression models has produced useful results, which improve on forecasts based solely on the rolling average of the stock price. The major difficulty was the inability to access financial data on the Yahoo Finance API other than daily price and volume information, from which the other metrics were calculated. Other, more fine-grained data, is likely to be more useful in predicting prices.

That said, the classification algorithm does produce useful results, which improve significantly upon those based solely on rolling averages. If sufficient funds were available to invest, and such investments were made across a diverse range of stocks (on the FTSE 100 at least), then one could expect to make a net gain over a prolonged period. The model could be re-run after the initial investments, to assist in deciding whether to retain the purchased stocks, or to sell and use the funds to invest elsewhere. Although brokerage fees would detract from the gains, many stocks would attract dividend income if they were retained for a longer period.

Although none of the models achieves the desired accuracy of 5% from the true price, 75% of the time, it should be remembered that the movement of stock prices is intrinsically unpredictable, being subject to influences that cannot be anticipated by numerical analysis; and the current market conditions (and those for the period for which training data is used - especially in London / FTSE 100) are particularly unpredictable, with the effects of Brexit, the US election, and other macroscopic events.

Future Work

Since the survey_market module collates and saves more financial data than is available on the YF API, it will be possible to use this data in future to assist in making predictions. I am also interested in applying reinforcement learning, to the problem, perhaps in conjunction with the LSTM model. There is certainly scope for further refinement of the LSTM model: the multiple features used in the classifier could be applied to the regressor, to compare the results with the baseline. It would also be sensible to simulate purchases based on the model for a period, to assess the returns, taking account of brokerage fees and dividend income.