Pattern Recognition for Mining Historical Prices
- In Progress -
Documentation

Ahmed-Karim BERRADA Nabil NOUAMAN
Karl BOU ABBOUD

January 23, 2017

Abstract

The goal of this paper is to present our findings on how to use machine learning techniques in the domain of pattern recognition in order to predict future movement of stocks. In our first report, we discussed the different possible algorithms that could easily adapt to this problem. We have implemented different methods to predict future prices. In this paper, we will review our methodology for each solution, and compare the performance in predicting future prices.

Contents

1 Introduction **2**
1.1 Problem . . . 2
1.2 Solutions . . . 2
1.3 Universe . . . 2
1.4 Framework . . . 2

2 Preparing the data **3**
2.1 Downloading stock data . . . 3
2.2 Adding the candlestick period . . . 3
2.3 Normalize the data . . . 3

3 Classification **3**
3.1 Our Approach . . . 3
3.2 Preparation . . . 4
3.3 Logistic Regression Classifier . . . 4
3.4 Random Forest Classifier . . . 4
3.5 Optimization . . . 5

4 Regression **6**
4.1 Linear Regression . . . 6

5 Appendices **7**
5.1 Python Code Organization and files . . . 7

1 Introduction

1.1 Problem

One of the biggest challenges of the finance sphere is to be able to predict precisely future stocks' fluctuations. In order to solve this problem of predicting future prices, we develop in this paper various machine learning approaches and compare their accuracy and speed in predicting the future price.

1.2 Solutions

In our first report, we have provided an overview of the principal machine learning approaches that could be used in our problem. Our conclusions can be summed up in the Table 1 below. In a first trial, we tried solving the price prediction problem with a classification algorithm. Because of its strong accuracy, we decided to use the *Random Forest* algorithm. We also worked with the *Logistic Regression* algorithm for classification, to compare the results obtained with the two techniques.

Algo	Type	Interpret	Accuracy	Train Speed	Small Data
KNN	Either	XXX	XX	XXX	No
Logistic Regression	Classification	XX	X	XXX	Yes
Random Forest	Either	XX	XXX	XX	No
Neural Network	Either	X	XXX	X	No
K-means	Either	X	X	XX	No
SVM	Classification	X	XX	XX	No

Table 1: Comparison of parameters configuration performance for the Regularized regression.

The goal of this project is to try out the machine learning algorithms among the ones above that seemed most relevant to solve the stock prediction problem.

1.3 Universe

The universe we work on to train and test our algorithms is the Tech sector in US. As our work is supposed to be used in other sectors and other databases, our code should be very flexible so the universe could be expanded later on.

1.4 Framework

Notations in the framework

- **n** = number of stocks
- **m** = number of bars used to describe a candlestick pattern
- **p** = size of in-sample training window
- **t** = number of days of the back-test

Data format All input are in matrix (or vector for only 1 stock) form. The matrices, the first dimension being each observation across time and the second dimension being each random variable, are in chronological order (the most recent observations on the last row).

Candlestick pattern description We use high/low/close prices for each day to represent a bar (we don't take into account the open price here). An m-period candlestick pattern will be described with $3m - 1$ values:

- The *close-to-close returns* over the m periods ($m - 1$ values)
- The *high* relative to the close on that day (m values)
- The *low* relative to the close on that day (m values)

Training algorithm

- **Input** (predictor): the m-period candlesticks of n stocks, as of day $t + 0$
- **Output** (observed outcome): the direction of day $t + 1$ price move (i.e. the sign and value of close-to-close return as of day $t + 1$). The algorithm will take p samples (i.e. the above-mentioned pair as of day t + 0, t - 1, ..., t – p + 1)

2 Preparing the data

2.1 Downloading stock data

To be able to run algorithms on some stock data, we first need to download them to prevent any firewall issue in accessing online data. For starting, we worked on two stocks to train and test our algorithms:

- The Yahoo stock : $YHOO$ from January 2010 to January 2017.
- The Microsoft Corporation stock : $MSFT$ from January 2010 to January 2017.

The function *get_ YF_ raw_ data* from *data_ extraction.py* extracts history prices using Yahoo Finance from a given start date until a stop date and filtrates by the main features.

2.2 Adding the candlestick period

One important part of predicting future prices is to determine a relevant number of past days to use as predictive features in the algorithm. The function *add_ feat* from *data_ extraction.py* adds the main features from the candlestick period chosen, and returns the data with the new columns.

2.3 Normalize the data

3 Classification

3.1 Our Approach

For the classification approach, we train the algorithm on the returns during a *training period* that works as a rolling window, and try to predict the return bucket of the day following the train period. We do this for every day and compute the Sharpe ratio and the accuracy, defined as the proportion of good classifications. The features are the high, low and returns of previous days. The buckets that the algorithm tries to predict are computed according to the previous distribution of the returns, during a bucket sampling period that we call *distribution period*.

3.2 Preparation

Setting Return buckets As we want to predict a multi-class variable, and not only a "positive/negative" binary value of the return, we need to divide the target variable into buckets.

Measure performance We decided to use the Sharpe Ratio to measure the performance of our classification algorithm.

3.3 Logistic Regression Classifier

Algorithm Principle

Example We computed a *Logistic Regression* classifier algorithm on the Yahoo stock *YHOO* from 2012-01-02 to 2016-02-05 with 30 days used for the features, 120 days for the training period and 120 days to compute the buckets.
The command line is :

- **python logit_regression.py YHOO 2012-01-02 2016-02-05 30 120 120**

where :

- **logit_regression.py** is the python file containing the execution module for the algorithm you want to test, in this case the *Logistic Regression* classifier.
- **YHOO** is the stock code from the Yahoo Finances database of the stock price you want to predict.
- **2012-01-02** is the start date.
- **2016-02-05** is the stop date.
- **30** is the candlestick period, which is the number of days used for the features.
- **120** (first one) is the number of days for the training period.
- **120** (second one) is the number of days for the distribution period to compute the bucket ranges to class the buckets in classes.

The result is shown in Figure 1.

```
('Accuracy (%) : ', 24.100719424460433)
('Sharpe Ratio bucket range number : 0', 0.94358110242706195)
('Sharpe Ratio bucket range number : 1', 0.17712403396171089)
('Sharpe Ratio bucket range number : 2', 0.82966229638968547)
('Sharpe Ratio bucket range number : 3', 0.28868260083023023)
('Sharpe Ratio bucket range number : 4', 0.15592932567774301)
('Sharpe Ratio bucket range number : 5', 1.4420493667439318)
```

Figure 1: Example of result obtained with the *Logistic Regression* classifier.

We can see that we obtain a global accuracy of 24%, as well as different Sharpe ratios for each bucket.

3.4 Random Forest Classifier

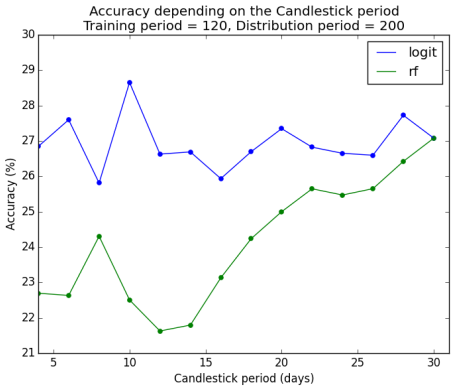
Similarly, we ran a test with the *Random Forest* classifier algorithm, for the same stock data. The accuracy we obtain is 29,5%, which is a little bit higher than with the *Logistic Regression*. It is not a good result for a predictive algorithm, but it is still better than a random choice between the 6 buckets (chances of success would be 17% in that case).

3.5 Optimization

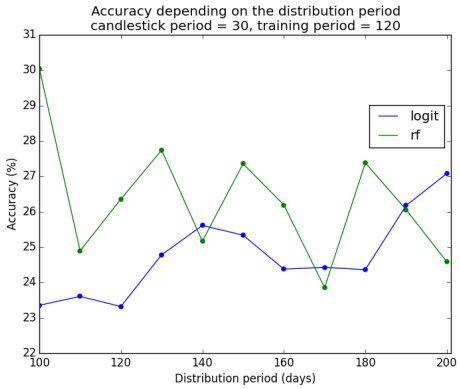
Now that our algorithms work and already give satisfying results. It can be interesting to try and optimize the parameters used in the algorithms. The three parameters we use in our models are :

- *The candlestick period* : the number of days that we consider as predictors (each day contain the high, low and the close return)
- *The training period* : the period to fit the model
- *The distribution period* : the number of days to model the distribution of the returns so we can define the quantiles/percentiles and classify returns into buckets.

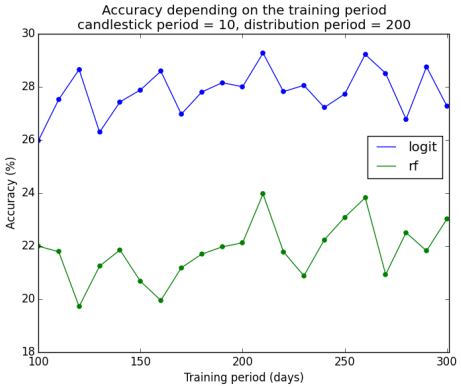
In other to find the best parameters, we start by fixing two and vary the third one to see how the accuracy evolves. We repeat the same process and using this time the value that gives the best accuracy. The code we used for this part is in the file *stats_ classifiers.py*.



(a) Candlestick period optimization.



(b) Distribution period optimization.



(c) Training period optimization.

Figure 2: Optimization of the parameters for classification : 2a for candlestick period, 2b for the distribution period and 2c for the training period.

The results show that we can't expect more than 30% in accuracy for the *Random Forest* and the *Logistic Regression* classifiers. It is a low value for a global accuracy, but it is still better than a random choice (one chance out of six : 17%).

4 Regression

4.1 Linear Regression

5 Appendices

5.1 Python Code Organization and files

- **README.md** : contains python package requirements list to run the algorithms, and examples of use cases of how to run this code from the command shell.
- **constantes.py** : contains global constant variables that are used throughout all the code :
  - *default_limit_classes* : bin ranges to class return values into categories ([-5,-2],[-2,-1]...).
  - *quartile_ranges* : quartile ranges for the return values from the distribution period to create categories of return classes.
  - *main_feat* : main features used for the prediction. Usually contains 'High', 'Low' and 'Close'.
  - *min_range* : minimum range for the return classes.
  - *max_range* : maximum range for the return classes.
  - *step_range* : step range for the return classes.
  - *yclass_label* : target variable's name to predict in classification algorithms.
  - *y_reg_label* : target variable's name to predict in regression algorithms.
- **data_extraction.py** : contains main functions to extract, download and process stock data from the YAHOO databases :
  - *get_raw_data* : extracts history prices from start date until stop date and filtrate by main features from the current folder.
  - *get_YF_raw_data* : extracts history prices from start date until stop date and filtrate by main features using Yahoo Finance.
  - *add_feat* : adds the main features from the candlestick period chosen, and returns the data with the new columns.
  - *add_bucket* : labels the returns into return classes computed with the percentiles of the distribution period given in parameter.
  - *get_ret_class* : labels the returns into return classes given in parameter, the class zero corresponding to the lowest return level.
- **logit_regression.py** : contains the logistic regression classifier :
  - *MAIN EXECUTION MODULE* : main module executed when the python file *logit_regression.py* is directly called form the command shell. Retrieves the 6 arguments *stock_name*, *start*, *stop*, *p_days*, *period*, *dist_period* entered by the user, and returns the logistic regression classifier's prediction accuracy and sharp ratio for each bucket range.
  - *LogisticRegression CLASS* : the logistic regression class object containing *train* and *predict* functions.
  - *softmax* : normalizes values to avoid overflow.
  - *test_lrn* : builds the multiclass Logit Regression Classifier.
  - *calc_pred_data* : calculates the predicted class for a certain observation and for a certain window period of training.
- **random_forest.py** : contains the random forest classifier :

  - *MAIN EXECUTION MODULE* : main module executed when the python file *random_forest.py* is directly called form the command shell. Retrieves the 6 arguments *stock_name*, *start*, *stop*, *p_days*, *period*, *dist_period* entered by the user, and returns the random forest classifier's prediction accuracy.
  - *fitting_forest* : returns the accuracy and predictions of the random forest classifier, given a stock data over a certain number of days, a training period and a number of estimators to be used in the random forest algorithm.

- **linear_regression.py** : contains the linear regression algorithm :
  - *MAIN EXECUTION MODULE* : main module executed when the python file *linear_regression.py* is directly called form the command shell. Retrieves the 5 arguments *stock_name*, *start*, *stop*, *p_days*, *period* entered by the user, and returns the linear regression's prediction Mean Squared Error and Score.
  - *fitting_linear* : returns the Mean Squared Error, Score and predictions of the linear regression algorithm, given a stock data over a certain number of days, a training period and a return threshold to be used in the linear regression algorithm.

- **stats_classifiers.py** : contains plotting functions to evaluate and optimize the accuracy of classifiers :
  - *MAIN EXECUTION MODULE* : main module executed when the python file *stats_classifiers.py* is directly called form the command shell. Retrieves the 5 arguments *stock_name*, *start*, *stop*, *p_days*, *period* entered by the user, and returns the plotting of random forest and logistic regression accuracy with two of the three parameters *candlestick period*, *training period* and *distribution period* which are fixed, and the last one that varies in a certain range of values.
  - *get_accuracy* : return the accuracy of an algorithm given in parameter with a given set of other parameters.

- **utils_lib.py** : contains functions for user input control :
  - *check_data_input* : raises exceptions if user input contains incorrect values.

- **utils_logit** : contains functions to compute accuracy and Sharpe Ratio for the *Logistic Regression* classifier :
  - *norm_input* : normalizes the inputs (training and test inputs) for different features (High, Low , Close).
  - *get_expeted_ret_range* : adds the expected return level in order to calculate the Sharpe Ratio. The expected return is approximated by the mean value of the return range.
  - *acc_pred* : calculates the accuracy of the predictions for the test period.
  - *sharpe_ratio* : calculate the Sharpe Ratio for the all period.
  - *get_ret_range* : returns a sorted array of return levels to help build the return classes.
  - *adjust_ret* : converts each return class k into an array with zeros and one in the kth index.
  - *get_sharpe_per_bckt* : returns Sharpe Ratio for each bucket of return range values.

- **utils_rf** : contains functions to preprocess stock data for the *Random Forest* classifier:
  - *subdivise_data* : returns a dictionary with the data subdivided into Train and Test.