DATA 618 Spring 2017 MiniProject 2

James Topor

April 11, 2017

This code makes use of three separate machine learning algorithms for purposes of attempting to predict the upward or downward movement of an individual financial market security. The three algorithms used are as follows:

```
1. Random Forest Classification
2. Support Vector Classification
3. Gaussian Naive Bayes Classification
```

These three classification algorithms are used as part of an "ensemble" stock trading methodology, wherein the predictive output of each algorithm has a direct impact on the structure of the next stock trade to be executed. The algorithms themselves are fitted through the use of recent "minutely" stock/security data metrics, such as price, volume, high, and low.

The methodology used within the code proceeds as follows:

```
build_models() module:
    1. To fit the classification models, the most recent 300 minutes of price,
    volume, high, and low data are retrieved.

    2. Binary 0/1 vectors indicating the directional changes in prices, volumes,
    highs and lows are derived from the 300-minute data set.

    3. A 15-item (items = minutes) length sliding "window" is applied to
    the binary 0/1 vectors calculated in
    step 2 above to extract small chronological subsets to be used for model fitting.
    This process yields two data structures: one containing data relative to the
    independent variables and one containing only the dependent variable to be
    predicted. In this code, we use the changes in price, volume, high, and low
    as the independent variables while the change in price is the dependent variable.

    4. Each of the three classification models is fitted using the binary 0/1
    subsets derived in step 3. Once fitted, the rounded output of each classifier
    will be indicative of whether or not the respective classification algorithm
    sees the price of the security either rising (rounded output == 1) or falling
    (rounded output == 0).

trade() module:
    1. Retrieve the most recent 15 minutes of price, volume, high, and low data

    2. Create binary 0/1 vectors indicating the directional changes of the
    15-minute snippet of data retrieved in step 1 above for the price, volume,
    high, and low variables.

    3. Create a single feature vector comprised of the binary 0/1 vectors created
    in step 2 above.

    4. Submit the feature vector to each of the three classification algorithms.
```

```
5. Sum the rounded outputs of the three classification algorithms. This sum
is referred to as "votes" in the code.

6. The number of "votes" is then used to assign a value to a 'weight'
variable. The 'weight' variable is used to adjust the amount of the security
to be bought or sold relative to the predictions of the three classification
algorithms. These weights are user-configurable, but are set as follows
herein:
    if votes == 3 (all 3 classifiers predict a price increase)
        weight = 1.00
    if votes == 2 (2 of the 3 classifiers predict a price increase)
        weight = 0.75
    if votes == 1 (1 of the 3 classifiers predicts a price increase)
        weight = 0
    if votes == 0 (None of the 3 classifiers predicts a price increase)
        weight == 0

7. The total amount of cash held in the portfolio is then determined

8. The algorithm then allows a maximum of 80% of that cash to be used
for the purchase of shares of the security. The 'weight' variable scales
the available cash amount before an order is placed. So, for example, if
votes == 2, a total of (cash * 0.80 * 0.75) can be purchased. The exact
number of shares to be purchased is determined by dividing the scaled available
cash amount by the current price of the security.

9. Place an order for the calculated amount of shares. Please note that for the
code as currently configured:
    - if votes == 1, no stock will be bought or sold
    - if votes == 0, all currently held stock will be sold
```

The build_models() and trade() modules are invoked according to a pre-defined schedule as shown within the initialize() and before_trading_start() modules. build_models() is invoked prior to the start of trading each day, and trade() is first invoked during the first minute of trading. Subsequently, both modules are invoked hourly up until 30 minutes before the close of trading.

Additional comments:

```
1. The trade() module has built-in flags that support the shorting of a security if
so desired by the user. For example, shorting might be appropriate if the price
of a security is predicted to decline. To enable shorting, set
context.shorting_enabled = True and set the weight variable to a negative value
between (-1, 0) within the trade() module where the "elif votes == 0:" clause
is evaluated.

2. The user is free to change the values assigned to the weight variable if
so desired. For example, the code as implemented here ensures that no trade
is executed if only one of the three classifiers predicts a price increase.
To change that behavior, simply increase the weight value within the trade()
module where the "elif votes == 1:" clause is evaluated.
```

Backtesting this code as-is with $100,000 in initial capital for the period 1/4/2010 - 4/7/2017 on Apple's

stock (AAPL) produces a total return = 347.9%, alpha = 0.09, sharpe = 1.29. The algorithm actually outperformed Apple's stock for long streches of time as indicated in the attached graphic.