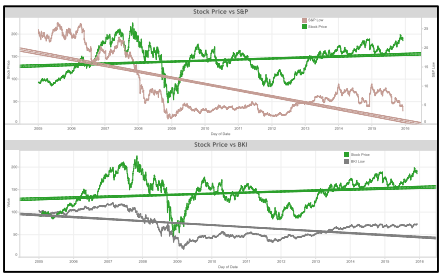


Predictive Analysis of Stock Price Using Random Forest Algorithm

Prepared By

Mahesh Suravajjala

ID: 0003646255

for

[FA15-BL-INFO-I590-34717]

Big Data Applications and Analytics

GIT: https://github.com/maheshs009/Project

1 SUMMARY

Primary purpose of this project is to train a Random Forest model to predict bank stock prices. Historical information of various market factors or features are used to train the Random Forest model. This project highlights the most important macroeconomic factors or features that would potential drive any stock price. A total of 25 different features/factors are considered, and last 10 years' data has been used to train and test the Random Forest model.

Most of the data used is real market data (both training and results), in order to avoid any conflicts, we will refer to the Bank that we are predicting stock price of as Bank A.

Key Highlights:

Random Forest Algorithm implemented and optimized using Python

Total number of features included: 25

Total number of data points: 2751 (01 Jan, 2005 to 03 Dec, 2015)

Model score (R^2) : 0.88 – Predicted stock price values are 88% correlated to actual stock prices.

2 HOW TO RUN

GIT: https://github.com/maheshs009/Project

Toolkit used: sklearn on the VM

Dataset:


feature_data.csv


results2.csv

Python Code:


RandomForest.py


RandomForest_Optimizer.py

Future Systems:

Login: msuravaj

All data is available on the VM machine - /home/ubuntu/Project/Data

IPython Files are available here - /home/ubuntu/IPythonFiles

Two files:

1. RandomForest_Optimizer.ipynb (contains steps to optimize the model)
2. RandomForest.ipynb

3 INTRODUCTION

Machine learning and other predictive analytics will start playing an increasing role in the financial world. While technical analysis (method of studying markets to predict future prices), and algorithmic trading have been around for a while, there are lot of other areas of finance where predictive analytics could play a major role. Collateral movements are one of the post-trade functions where billions of dollars are moved around on a daily basis. Collateral movements are purely based on daily market movements, and financial firms will need to make sure that they maintain enough liquidity to meet these collateral calls on a daily basis. Collateral is posted to the counterparties as a security based on the market movement. In order to post collateral, firms will need to maintain liquid cash/treasuries on a daily basis, which could otherwise be used for trading. Predicting collateral requirements is a key to financial firms to maintain a tight liquidity so that neither too high nor too low is maintained on a daily basis. While there is no straight forward of calculating the collateral needs of the firm directly, however, credit rating of the firm plays a key role. Credit Ratings have a huge impact on the stock price of a firm, so various fundamental market factors are considered to predict the stock price, which in turn, should help to predict collateral movements.

Identifying the parameters that impact the price movement of a financial firm is one of the primary outputs of this exercise. As a part of this project, Python based random forest algorithm that can predict the market movements of a financial firm (like Goldman Sachs or JP Morgan or Morgan Stanley) has been developed. Last 10 years' data has been availed to implement a supervised learning technique that can predict simple price movements and thereby other financial aspects of the firm. In a way, this could also help see if Random Forest techniques are applicable to financial data, or if overfitting might turn out to be a problem.

4 RANDOM FOREST INTRODUCTION

Random forest algorithm, as defined in wiki, is an ensemble learning method for classification, regression that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode or median of the classes. Random Forest is both a classification and regression algorithm developed by Leo Breiman and Adele Cutler. This technique uses a number of decision tree models to predict precisely by reducing the bias and variance of the estimates. This model is an ensemble since it aggregates many models together to produce a single final prediction. Algorithm of this model, on high level, is:

1. Create large number of decisions trees based on different subsets of variables
2. Bootstrap observations for each tree
3. Randomly select and use only part of the variables for each tree

4. Estimate the performance for each tree using the samples excluded
5. An average for regression estimates is derived as final prediction

5 Data Sources

General market indicators have been selected to act as the features/variables to train this model. Variables selected are:

1. S&P 500 – daily low, high, close prices
2. VIX – Volatility Index. This indicates how much volatile the markets are. – daily low, high, close prices
3. Banking Index – Index of all banking stocks – daily low, high, close prices
4. LIBOR Rates – 1-week, 1-month, 3-month, 6-month, 12-month rates
5. OIS – Overnight Interbank Spreads
6. Day of the week
7. Month
8. Historical stock price of a bank (referred to as Bank A)

Data has been sourced from multiple platforms. LIBOR rates have been sourced from St. Louis Fed research website (here). S&P, Bank A stock price are retrieved from Yahoo finance, and VIX has been retrieved from Quandl.com

All the data has been sourced from 01/03/2005 to 12/03/2015.

6 Data Curation – Data Loading/Cleaning

Data Loading:

There are total of 25 features loaded for the period of 10 years – Jan 1 2005 – till date. Please note that features and variables are interchangeable used all through this documentation. Features are usually divided into Dimensions (Categorical) and Measures (Numerical).

Training data, and results are loaded using pandas csv read module as shown below.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import roc_auc_score
import pandas as pd
training_data = pd.read_csv("../Project/Data/feature_data.csv")[:2500]
training_results = pd.read_csv("../Project/Data/results.csv")[:2500]
test_data = pd.read_csv("../Project/Data/feature_data.csv")[2501:]
test_results = pd.read_csv("../Project/Data/results.csv")[2501:]
training_results = training_results['Adj Close'].values
test_results = test_results['Adj Close'].values
```

Total number of rows in the data set is 2751. This dataset is further broken down into training_data and training_results, & test_data and test_results data sets. Training datasets will be used to train the model ([:2500] – first 2500 rows) and next 251 rows will be used to test the model ([2501:]).

And the last two lines in the above screenshot are needed to convert the result set from vector to 1-D array to avoid below warning:

```
-c:3: DataConversionWarning: A column-vector y was passed when a 1d array
was expected. Please change the shape of y to (n_samples,), for example us
ing ravel().
```

Numerical Data Curation:

Now that the data is loaded, we now check if all the data fields are populated. Check for any blanks, by running the describe () on training_data set.

```
In [107]: training_data.describe()
```

Out[107]:

	S&P Open	S&P High	S&P Low	S&P Close	S&P Volume	S&P Adj Close	LIBOR 1-Month	LIBOR 1-week	LIBOR 3-Month	LIBOR 6-month	LIBOR 12-Month
count	2750.000000	2750.000000	2750.000000	2750.000000	2.750000e+03	2750.000000	2701.000000	2701.000000	2701.000000	2701.000000	2701.00000
mean	9.634462	9.780182	9.477800	9.626204	3.548130e+07	9.293144	1.648726	1.598690	1.791196	1.955242	2.184853
std	7.580838	7.643744	7.519635	7.586268	3.491513e+07	6.972551	2.009020	2.005475	1.995154	1.936476	1.802230
min	1.590000	1.680000	1.350000	1.370000	2.224800e+06	1.370000	0.147750	0.116350	0.222850	0.319400	0.533500
25%	4.240000	4.320000	4.120000	4.210000	1.492432e+07	4.210000	0.199200	0.163710	0.277100	0.439900	0.765380
50%	5.790000	5.855000	5.680000	5.770000	2.670870e+07	5.770000	0.260000	0.248560	0.457600	0.727400	1.067000
75%	16.952499	17.157500	16.765000	16.962499	4.399730e+07	16.846355	3.220000	3.147500	3.379380	3.530000	3.780000
max	27.020000	27.200001	26.559999	26.900000	5.905267e+08	24.108302	5.823750	6.006250	5.725000	5.640000	5.766250

As you could see above, there are a total of 2750 values in each column. However, there a few missing values in LIBOR rates as shown above (total 2701 values present). In Random forest, we usually replace the missing values with the median of the feature. In Random Forest, decision trees are split on a certain value, and if it is less than a value it goes left and if greater than a value if goes right. So, if a median is picked in place of a missing value, it is not going be biased one way or the other with respect to that feature.

8 rows × 22 columns

```
In [ ]: #fill in blanks in age field with mean of the set
feature_variables = ['S&P Open','S&P High','S&P Low','S&P Close','S&P Volume','S&P Adj Close','LIBOR 1-Month','LIBOR 1-week','LIBOR 3-Month

for variable in feature_variables:
    training_data[variable].fillna(training_data[variable].mean(), inplace=True)
```

```
In [ ]: training_data.describe()
```

As you could see below, all the missing values are replaced with mean so the total count across all the features is 2750.

```
In [118]: training_data.describe()
```

Out[118]:

	S&P Open	S&P High	S&P Low	S&P Close	S&P Volume	S&P Adj Close	LIBOR 1-Month	LIBOR 1-week	LIBOR 3-Month	LIBOR 6-month	LIBOR 12-Month
count	2750.000000	2750.000000	2750.000000	2750.000000	2.750000e+03	2750.000000	2750.000000	2750.000000	2750.000000	2750.000000	2750.00000
mean	9.634462	9.780182	9.477800	9.626204	3.548130e+07	9.293144	1.648726	1.598690	1.791196	1.955242	2.184853
std	7.580838	7.643744	7.519635	7.586268	3.491513e+07	6.972551	1.991034	1.987521	1.977293	1.919140	1.786095
min	1.590000	1.680000	1.350000	1.370000	2.224800e+06	1.370000	0.147750	0.116350	0.222850	0.319400	0.533500
25%	4.240000	4.320000	4.120000	4.210000	1.492432e+07	4.210000	0.199300	0.165488	0.278210	0.442190	0.766500
50%	5.790000	5.855000	5.680000	5.770000	2.670870e+07	5.770000	0.260630	0.250250	0.465650	0.730400	1.069500
75%	16.952499	17.157500	16.765000	16.962499	4.399730e+07	16.846355	3.175000	3.133437	3.297500	3.490000	3.740473
max	27.020000	27.200001	26.559999	26.900000	5.905267e+08	24.108302	5.823750	6.006250	5.725000	5.640000	5.766250

8 rows × 22 columns

We need to make sure both the training_data and training_results have the same number of entries.

```
In [145]: print training_data.shape
          print training_results.shape

          (2750, 24)
          (2750, 1)
```

Categorical Data Curation:

As discussed above, every data set has two types of variables, dimensions (categorical) and measures (numeric). While numeric can be used directly for decision trees, categorical values will need to be dealt with carefully. In this dataset, there are 3 categorical variables as shown below:

```
In [222]: def describe_categorical(training_data):
              """
              Just like .describe but works for categorical
              """
              from IPython.display import display, HTML
              display(HTML(training_data[training_data.columns[training_data.dtypes == "object"]].describe().to_html()))
```

```
In [223]: describe_categorical(training_data)
```

	Date	Month	Weekday
count	2750	2750	2750
unique	2750	12	5
top	4/15/2013	Aug	Wednesday
freq	1	243	566

While Weekday might have an impact on the way stock price of a bank can move, the actual date might have little or no impact at all. Technical analysis of stocks does highlight the fact that stock price is impacted by Quarterly results, so month of the year might have some impact. So, Weekday, and Month will be left, while Date will be dropped.

```
In [196]: training_data.drop(["Date"], axis=1, inplace=True)
```

So, we will convert these categorical values – Month and Weekday into Boolean values. This way, these features can be used in decision trees. Dummies module in pandas support creation of dummy features in place of existing features. So, we create dummy variables using WeekDay – {Monday to Friday} & Month – {Jan to Dec} respectively.

```
In [246]: #convert all ccategorical variables mentioned below to booleans of 0 and 1
categorical_variables = ['Month', 'Weekday']

for variable in categorical_variables:
    training_data[variable].fillna("Missing", inplace="True")
    dummies = pd.get_dummies(training_data[variable], prefix=variable)
    training_data = pd.concat([training_data, dummies], axis=1)
    training_data.drop([variable], axis=1, inplace=True)
```

To confirm, we printall the features one more time:

Month:

```
In [248]: #Look at all the columns in the dataset
def printall(training_data, max_rows=100):
    from IPython.display import display, HTML
    display(HTML(training_data.to_html(max_rows=max_rows)))

printall(training_data)
```

h	BKI Low	BKI Close	BKI Volume	Close	Month_Apr	Month_Aug	Month_Dec	Month_Feb	Month_Jan	Month_Jul	Month_Jun	Month_Mar	Month_Ma
002	103.330002	103.529999	0	103.529999	0	0	0	0	1	0	0	0	0
998	102.250000	102.519997	0	102.519997	0	0	0	0	1	0	0	0	0
996	102.239998	102.250000	0	102.250000	0	0	0	0	1	0	0	0	0
997	102.360001	102.680000	0	102.680000	0	0	0	0	1	0	0	0	0
003	102.070000	102.089996	0	102.089996	0	0	0	0	1	0	0	0	0
999	101.739998	102.139999	0	102.139999	0	0	0	0	1	0	0	0	0
004	101.430000	101.720001	0	101.720001	0	0	0	0	1	0	0	0	0
000	100.620003	101.120003	0	101.120003	0	0	0	0	1	0	0	0	0
000	99.830002	99.940002	0	99.940002	0	0	0	0	1	0	0	0	0
003	99.730003	100.120003	0	100.120003	0	0	0	0	1	0	0	0	0

Weekday:

n	Month_Mar	Month_May	Month_Nov	Month_Oct	Month_Sep	Weekday_Friday	Weekday_Monday	Weekday_Thursday	Weekday_Tuesday	Weekday_Wednesday
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	0	0	0	0
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	0	0	0	0

To confirm the dimensions

```
In [253]: print training_data.shape
          print training_results.shape

          (2750, 39)
          (2750,)
```

7 RANDOM FOREST – FEATURE IMPORTANCE & CORRELATION

Features & Importance:

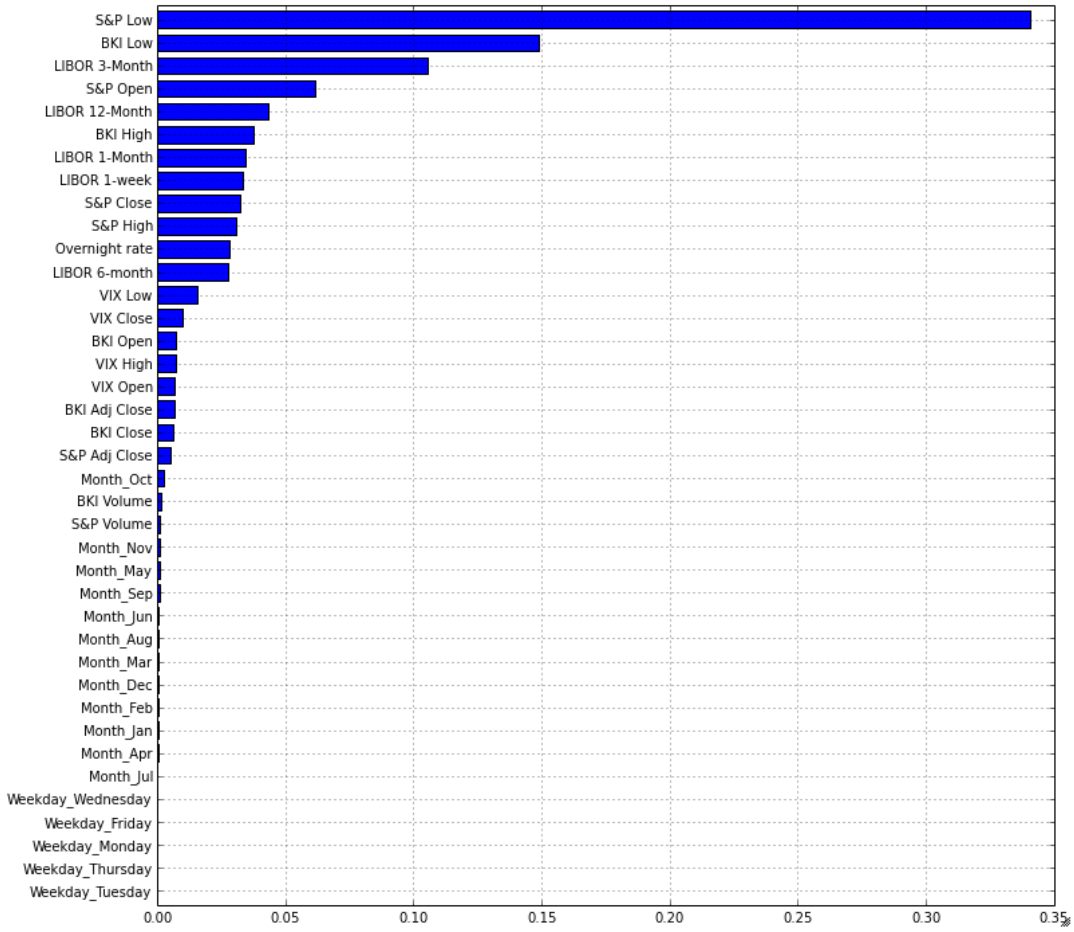
Now that the data is curated, and organized, Random Forest model can be run to understand the feature importance. Running the model with default settings

```
In [303]: model = RandomForestRegressor(1000, oob_score=True, n_jobs=-1, random_state=42)
          model.fit(training_data, training_results)
          #print "C-stat: " , roc_auc_score(y, model.oob_prediction_)

Out[303]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features='auto', max_leaf_nodes=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=1000, n_jobs=-1, oob_score=True, random_state=42,
                      verbose=0, warm_start=False)
```

Model.feature_importance_ explains the importance of all the features in the training data set as shown below.

```
In [305]: #Variable Importance Measure
          model.feature_importances_

Out[305]: array([  6.11011096e-02,   3.22464813e-02,   3.42301203e-01,
                   3.34594742e-02,   1.10883864e-03,   5.42740014e-03,
                   2.90411530e-02,   3.23512013e-02,   9.95679105e-02,
                   2.43733169e-02,   4.03106517e-02,   2.50916336e-02,
                   7.15187480e-03,   7.43404506e-03,   2.38056736e-02,
                   1.30372266e-02,   8.82239246e-03,   3.89360982e-02,
                   1.49913896e-01,   7.33754412e-03,   1.01278809e-03,
                   7.00845212e-03,   2.95835537e-04,   8.30054146e-04,
                   5.16145504e-04,   4.35199957e-04,   4.15734898e-04,
                   1.73096904e-04,   6.18907465e-04,   4.98763441e-04,
                   8.55608227e-04,   8.66397433e-04,   2.60264604e-03,
                   7.77283961e-04,   5.43731492e-05,   5.84213573e-05,
                   4.86660108e-05,   5.15858356e-05,   6.09152188e-05])
```

To represent the same in graphical format:



As defined by the graph, S&P Low, Banking Index Low, LIBOR 3-month rate, S&P Open, and LIBOR 12-month are some off the most important features that are informative, while the rest are not.

If traditional analytics were to be used, below is how correlation between the two 2 features (S&P and BKI) highlighted by above graph will stand:

R2 = 0.0066 (Between Stock Price and S&P Low)

Stock Price = 0.364092*S&P Low + 138.68

R2 = 0.13339 (Between Stock Price and BKI)

Stock Price = 0.471347*BKI Low + 109.646

Developed using Tableau

Traditional correlation factor R2, with 0.006 & 0.133, does not show that stock price has a great correlation with S&P Low or BKI Low price, but Random Forest paints a different picture. **This could be deemed as one of the primary advantage of using an ensemble model like Random Forest.**

Number of trees (n_estimators):

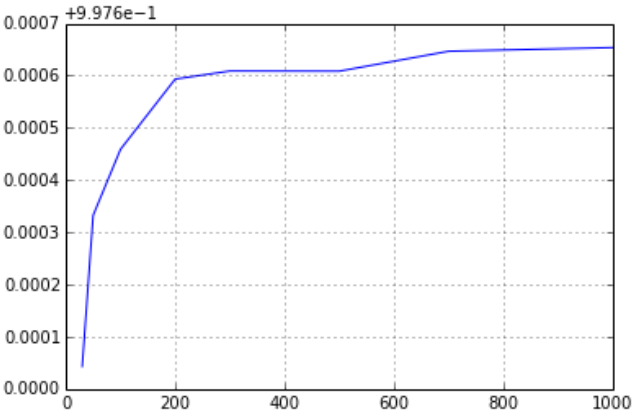
```
In [ ]: model = RandomForestRegressor(n_estimator=1000, oob_score=True, n_jobs=-1, random_state=42)
        model.fit(training_data, training_results)
```

n_estimator option defines the number of decision trees created. This has an impact on the accuracy of the model. A simple customized grid search algorithm can be run to find the optimal size of the decision trees needed for our model. As a part of this algorithm, RandomForestRegressor is run through different options for n_estimators like [30, 50, 100, 300, 500, 700, 1000, 1200] and out of bag (oob) and model scores are calculated.

```
results=[]
n_estimator_options = [30, 50, 100, 200, 300, 500, 700, 1000]

for trees in n_estimator_options:
    model = RandomForestRegressor(trees, oob_score=True, n_jobs=-1, random_state=42)
    model.fit(training_data, training_results)
    print trees, "trees"
    score = model.score(training_data, training_results)
    print "Model Score: ", score
    print "oob_score is: ", model.oob_score_
    results.append(score)
    print ""

pd.Series(results, n_estimator_options).plot();
```

```
30 trees
Model Score:  0.997643079863
oob_score is:  0.981989008688

50 trees
Model Score:  0.997932515081
oob_score is:  0.98461895245

100 trees
Model Score:  0.998058905193
oob_score is:  0.985533822104

200 trees
Model Score:  0.998193458398
oob_score is:  0.986688875717

300 trees
Model Score:  0.998208897075
oob_score is:  0.986681506202

500 trees
Model Score:  0.998208593531
oob_score is:  0.98689858929

700 trees
Model Score:  0.998246774113
oob_score is:  0.987119368415

1000 trees
Model Score:  0.99825378679
oob_score is:  0.98712059152
```

If these numbers are plotted, we can see the below. As shown in the plot, it becomes –estimators stable between 700-1000 trees, and thereafter it kind of flattens out. From this, it can be deducted that n_estimators need to be between 700-1000.



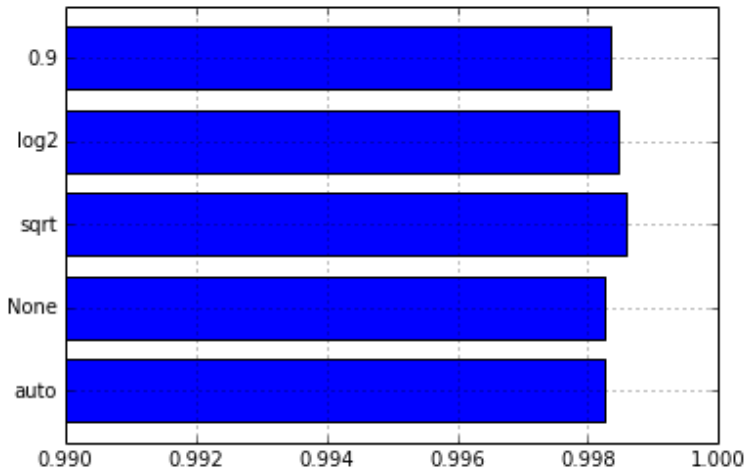
Max_features:

Max_features are the number of features Random Forest considers at each split point of the decision tree. Given that this exercise is to predict the value of a stock price (and not a Boolean), it is recommended that we use all the variables at each split point. However, a quick check could be run to understand the

behavior. This algorithm will be run with max_features varying between [auto, none, sqrt, log2, 0.9, 0.2]. Default option is “auto”.

```
In [5]: #max_features
        results = []
        max_features_options = ["auto", None, "sqrt", "log2", 0.9]

        for max_features in max_features_options:
            model = RandomForestRegressor(n_estimators=800, oob_score=True, n_jobs=-1, random_state=42, max_features=max_features)
            model.fit(training_data, training_results)
            print max_features, "option"
            score = model.score(training_data, training_results)
            print "Model Score: ", score
            print "oob_score is: ", model.oob_score_
            results.append(score)
            print""

        pd.Series(results, max_features_options).plot(kind="barh", xlim=(.99,1.00));

        auto option
        Model Score:  0.998262267546
        oob_score is:  0.987148022643

        None option
        Model Score:  0.998262267546
        oob_score is:  0.987148022643

        sqrt option
        Model Score:  0.998577813736
        oob_score is:  0.989420416342

        log2 option
        Model Score:  0.998471143042
        oob_score is:  0.988625906577

        0.9 option
        Model Score:  0.998356524772
        oob_score is:  0.987866764607
```

Representing these through plot shows that sqrt of total number of features will probably be enough to run this algorithm. While sqrt is usually an option used for classification problem, regression problems are usually defaulted to auto.



Min_sample_leaf:

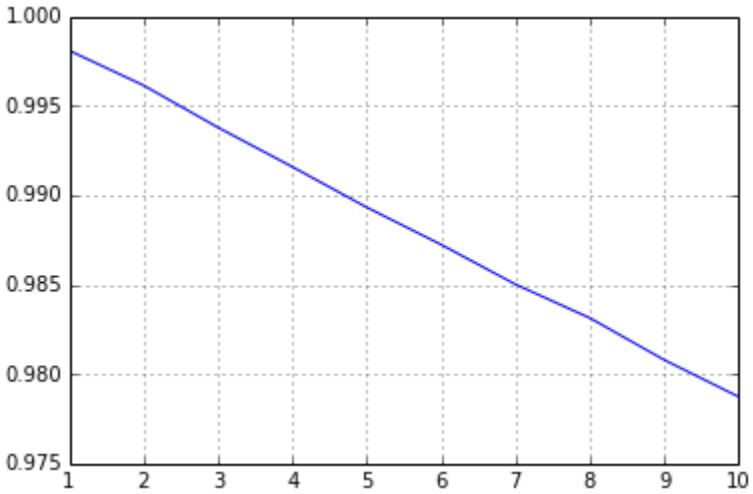
This is minimum samples per leaf. Decision tree is split further and further until there is only 1 observation left. However, models can be trained to stop splitting further by defining min_sample_leaf. Here, we will use the options [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to understand the accuracy of the model.

In [7]:
```
#min_samples_leaf
results = []
min_samples_leaf_options = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for min_samples in min_samples_leaf_options:
    model = RandomForestRegressor(n_estimators=800, oob_score=True,
                                  n_jobs=-1, random_state=42, max_features="sqrt",
                                  min_samples_leaf=min_samples)
    model.fit(training_data, training_results)
    print min_samples, "min samples"
    score = model.score(training_data, training_results)
    print "Model Score: ", score
    print "oob_score is: ", model.oob_score_
    results.append(score)
    print""

pd.Series(results, min_samples_leaf_options).plot();
```

Results:

```
1 min samples
Model Score:  0.998577813736
oob_score is:  0.989420416342

2 min samples
Model Score:  0.997132621367
oob_score is:  0.987669477335

3 min samples
Model Score:  0.995377544105
oob_score is:  0.985998656924

4 min samples
Model Score:  0.993438252621
oob_score is:  0.983841047807

5 min samples
Model Score:  0.991624579739
oob_score is:  0.982050555059

6 min samples
Model Score:  0.989515402955
oob_score is:  0.979527203426

7 min samples
Model Score:  0.987638896014
oob_score is:  0.977638466372

8 min samples
Model Score:  0.98574492997
oob_score is:  0.9755916442

9 min samples
Model Score:  0.983881667098
oob_score is:  0.973847646276

10 min samples
Model Score:  0.981933378211
```

Plotting the output shows that accuracy of the model decreases as the min_sample_leaf goes down. So, the model needs to have a min_sample_leaf value of 1.



8 THROUGHPUT

Throughput of the model is decided by the number of cores used to run the model. N_jobs field lets you define how many cores of the existing cores on the server can be used to run this model.

A value of n_jobs=-1 means that all the available cores will be used for this model. And noted below, the best run took **2.89 sec.**

```
In [18]:
%%timeit

model = RandomForestRegressor(n_estimators=800, oob_score=True,
                              n_jobs=-1, random_state=42, max_features="sqrt",
                              min_samples_leaf=1)
model.fit(training_data, training_results)

1 loops, best of 3: 2.89 s per loop
```

A value of n_jobs=1 means that 1 core out of the available cores will be used for running this model. As noted below, the best run took **7.05 sec**

```
In [20]:
%%timeit

model = RandomForestRegressor(n_estimators=800, oob_score=True,
                              n_jobs=1, random_state=42, max_features="sqrt",
                              min_samples_leaf=1)
model.fit(training_data, training_results)

1 loops, best of 3: 7.05 s per loop
```

Thanks to future systems setup, I was able to increase my VM to (m1.xlarge flavor), which has 8 vcpus and 16g ram. So the model run took 2.8 sec to run on future systems. If the same model were to run on my dual core laptop it takes 4-5 sec per run.

9 RANDOM FOREST

Now that all the important attributes have been tested and optimized, Random forest model can be run using these optimal values.

```
In [8]:
model = RandomForestRegressor(n_estimators=800, oob_score=True,
                              n_jobs=-1, random_state=42, max_features="sqrt",
                              min_samples_leaf=1)
model.fit(training_data, training_results)
model.score(training_data, training_results)

Out[8]: 0.99857781373633836
```

Now that the model is run, it can be tested.

Prepare Test Data:

One of the pre-requisites for running Random Forest is to have same number of features in the test data, as in the training data. And the test data will need to be prepared, as training data was done. Below code snippet does the same.

```
In [13]: #Prepare test data
feature_variables = ['S&P Open','S&P High','S&P Low','S&P Close','S&P Volume','S&P Adj Close','LIBOR 1-Month','LIBOR 1-week','LIBOR 3-Month

for variable in feature_variables:
    test_data[variable].fillna(test_data[variable].mean(), inplace=True)

test_data.drop(["Date"], axis=1, inplace=True)

categorical_variables = ['Month', 'Weekday']

for variable in categorical_variables:
    test_data[variable].fillna("Missing", inplace="True")
    dummies = pd.get_dummies(test_data[variable], prefix=variable)
    test_data = pd.concat([test_data, dummies], axis=1)
    test_data.drop([variable], axis=1, inplace=True)

print test_data.shape
print test_results.shape

(249, 39)
(249,)
```

Now the test data and results can be run through the model to figure the model score.

```
In [14]: print test_data.shape
         print test_results.shape
         model.score(test_data, test_results)

         (249, 39)
         (249,)
Out[14]: 0.88089776228522865
```

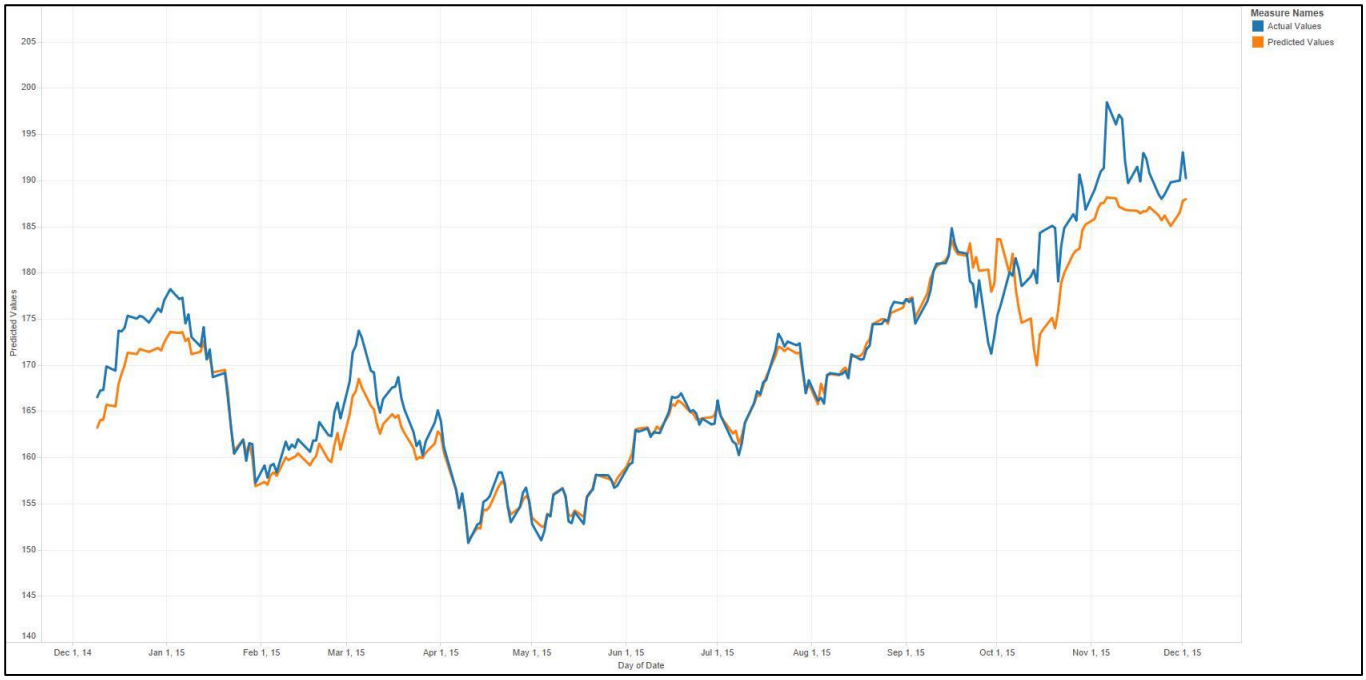
Model takes the test_data and generates the predicted stock price of the values, and then compares it with the test results (actual value) and generates a score. The score is nothing but the coefficient of determination R^2 of the prediction. R^2 of 0.88 suggests that the model predicts the values close to 88% of the actual values.

Predicted stock values can be stored in the file as shown below.

```
In [15]: savetxt('../Project/Data/predict.csv', model.predict(test_data), delimiter=',', fmt='%f')
```

10 RESULTS

A quick graph of the actual vs predicted values shows that the values are mostly correlated. Actual value of the stock price has usually been outsmarting the predicted value.



Developed using Tableau

1. Actual value (blue line) is more than the predicted value (orange line). This is expected. There are a lot more features/attributes that impact the stock price of a bank.
2. There is a lot more volatility in the actual value, compared to predicted value. Stock prices react to many of the qualitative and quantitative factors apart from the features included in this project.
3. The model can be definitely fine-tuned further. There are other features like quarterly results, and bank fundamentals that can make the prediction a lot more robust.

11 Next Steps

If this model were to be turned into a complete application, below are next steps:

1. Write an API to gather the feature data real time. Most of the feature/attributes used in this model are available through feeds real-time from various systems like Bloomberg, Reuters quandl etc.
2. Collect the data in NoSQL server like Mongo Db. This way the features/attributes can be added/removed real-time. So a traditional relational db does not work
3. More features need to be added, as the current set of features are very generic and not specific to Banking sector. Different stock prices can be predicted based on different features available. Algorithm can be fine-tuned to pick different features for different sectors.
4. One of the major uplift needed for this model will be to include “Sentiment Analysis”. Stock prices are vulnerable to market sentiments, as much as they are to the attributes used for this model. For ex, actual stock price in the graph above is always higher and volatile than the predicted value because stock price reacts to different market events.

5. If the model were to be made robust, “sentiment analysis” will be a major feature. This will include market events like rise in coupon payments, major firms filing for bankruptcy, change in political system or unpegging a major international currency (swiss franc was unpegged earlier this year from USD).
6. Over-fitting is one of the common problems with ensembling techniques like Random Forest. Test this model for over-fitting problem.
7. Use Hadoop infrastructure to collect and store the date.
8. Work in a team!!! This could have been a fully functional web application if there were more hands.

Finally, a big thank you to support group. You guys have been amazing and quick on feet!!!

REFERENCES

1. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html#sklearn.ensemble.RandomForestRegressor.score
2. https://www.youtube.com/watch?v=0GrciaGYzV0
3. https://www.kaggle.com/c/titanic/details/getting-started-with-random-forests
4. http://www.bios.unc.edu/~dzeng/BIOS740/randomforest.pdf
5. https://www.youtube.com/watch?v=MRi99ax50sw
6. http://www.analyticsvidhya.com/blog/2015/09/random-forest-algorithm-multiple-challenges/