CQF Final Project

2019

Market Prediction with Machine Learning

By

Deepak Kumar Giri

Table of Contents

1 Market Prediction with Machine Learning 3
2 Engineering Data Features 3
2.1 Feature Normalization 3
2.2 Lagged Returns 4
2.3 Simple moving average (SMA) 5
2.4 Exponential weighted moving average 7
2.5 RSI 8
2.6 MACD 11
2.7 Simple Moving average of Variance 12
2.8 EWMA on Variance 13
3 Model Evaluation 13
3.1 K-Fold Cross Validation 13
3.2 Train Test split of data set 14
3.3 Confusion Matrix 15
3.4 Precision recall 16
4 Predictions using Classifiers 17
4.1 Logistic Regression 17
4.1.1 Logistic Regression with L1 Penalty & different Regularization strength 22
4.2 Support Vector Machines 23
4.3 K-Nearest Neighbours (KNN) 26
4.4 Decision Tree and Ensemble Methods (AdaBoost) 31
4.5 ANN and Deep Learning 33
5 Python code 39
5.1 Common Classes and Methods 39
6 Conclusion 40
7 Software Tools and Libraries 40
8 References 40

1 Market Prediction with Machine Learning

Stock market prediction, which has the capacity to reap large profits if done wisely, has attracted much attention from academia and business. Due to the non-linear, volatile and complex nature of the stock market, it is quite difficult to predict. The question remains: To what extent can the past history of a common stock's price be used to make meaningful predictions concerning the future price of the stock?". In this project, the goal is to examine the important and potential factors/predictors that could drive the stock market and develop a set of models to predict the short-term stock movement and price.

However, usage of machine learning in stock market prediction requires much more than a good grasp of the concepts and techniques for supervised machine learning.

2 Engineering Data Features

This section is going to delve into the mechanics of feature engineering for the sorts of time series data that you may use as part of a stock price prediction AI modelling systems.

Feature engineering is a term of art for data science and machine learning which refers to pre-processing and transforming raw data into a form which is more easily used by machine learning algorithms. We'll cover the important types of feature extraction methods, then offer some useful python code recipes for transforming the raw source data into features which can be fed directly into a ML algorithm or ML pipeline.

2.1 Feature Normalization

Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. For machine learning, every dataset does not require normalization. It is required only when features have different ranges.

Most of the times, your dataset will contain features highly varying in magnitudes, units and range. But since, most of the machine learning algorithms use Eucledian distance between two data points in their computations, this is a problem.

If left alone, these algorithms only take in the magnitude of features neglecting the units. The results would vary greatly between different units, 5metre and 500cm. The features with high magnitudes will weigh in a lot more in the distance calculations than features with low magnitudes.

To supress this effect, we need to bring all features to the same level of magnitudes. This can be achieved by scaling.

The result of Normalization (or Z-score normalization) is that the features will be rescaled so that they'll have the properties of a standard normal distribution with

$\mu=0$ and $\sigma=1$

where μ is the mean (average) and σ is the standard deviation from the mean; standard scores (also called z scores) of the samples are calculated as follows:
$z=(x-\mu)/\sigma$

Here is the sample python code for the same:

```
def gaussian(xa):
    x = xa.copy()
    mean = x.mean()
    std = x.std()
    return (x - mean) / std, mean, std
```

2.2 Lagged Returns

The returns of a stock can be predicted by the below formulae

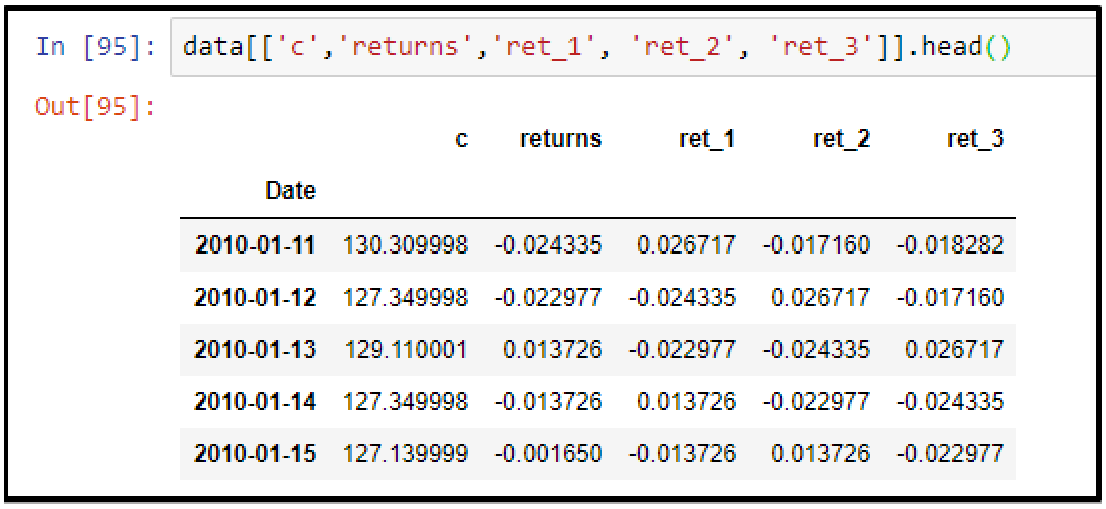
```
    # The log return
    data['returns'] = np.log(data['c'] / data['c'].shift(1))
```

We compute log returns by dividing close price with yesterday's close price, and the above code does this for all the historical close prices.

The lagged return with lag=1 is the returns computed 1 day ago, similarly lagged returns for lag=n is the return n day ago. Once we have returns computed for all the dates in the data frame then a lagged return can be computed by the below code.

```
    LAGS = 5
    for lag in range(1, LAGS+1):
        col = 'ret_%d' % lag
        data[col] = data['returns'].shift(lag)
        cols.append(col)
```

Note that the data is shifted by the amount of lag and then appended to with a new column name. Here is the sample data of lagged returns



```
In [95]: data[['c','returns','ret_1', 'ret_2', 'ret_3']].head()

Out[95]:
                     c    returns     ret_1     ret_2     ret_3
      Date
2010-01-11  130.309998  -0.024335  0.026717 -0.017160 -0.018282
2010-01-12  127.349998  -0.022977 -0.024335  0.026717 -0.017160
2010-01-13  129.110001   0.013726 -0.022977 -0.024335  0.026717
2010-01-14  127.349998  -0.013726  0.013726 -0.022977 -0.024335
2010-01-15  127.139999  -0.001650 -0.013726  0.013726 -0.022977
```

2.3 Simple moving average (SMA)

A simple moving average (SMA) is an arithmetic moving average calculated by adding recent closing prices and then dividing that by the number of time periods in the calculation average. A simple, or arithmetic, moving average that is calculated by adding the closing price of the security for a number of time periods and then dividing this total by that same number of periods. Short-term averages respond quickly to changes in the price of the underlying, while long-term averages are slow to react.

The Formula For SMA Is

$$\text{SMA} = \frac{A_1 + A_2 + \ldots + A_n}{n}$$

where:

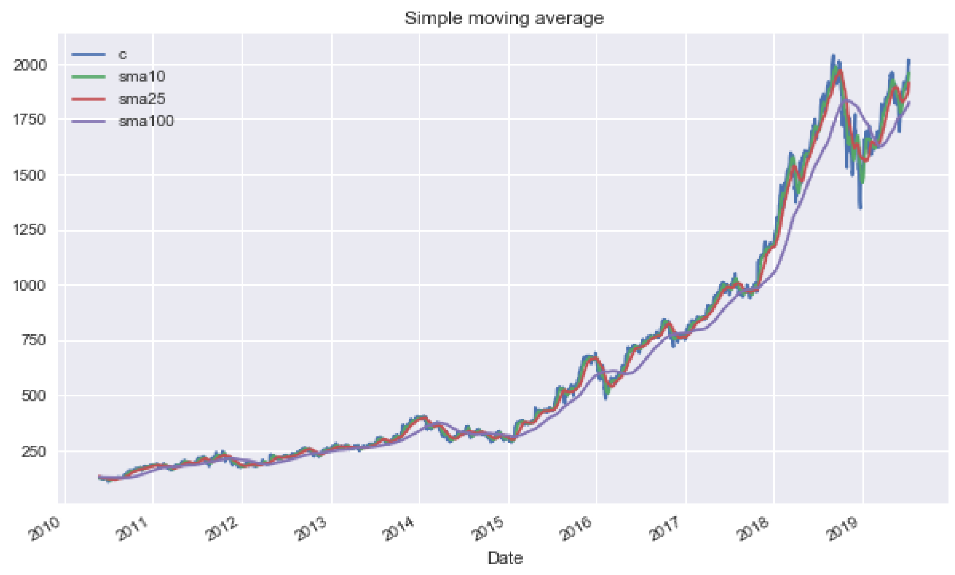
A_n = the price of an asset at period n

n = the number of total periods

Here is the plot of Simple moving average computed for a period of 10 days, 25 days and 100 days. Note that in our prediction algorithms we have relied on SMA of 10 days and 25 days. The plot of SMA 100 days in the figure shows us how the SMA data is lagging behind the actual trend.

```
    data['sma10'] = data['c'].rolling(10).mean() # Simple moving average
    data['sma25'] = data['c'].rolling(25).mean()
    data['sma100'] = data['c'].rolling(100).mean()
```

```
In [80]: data[['c','sma10', 'sma25', 'sma100']].plot(figsize=(10, 6), title="Simple moving average")
Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0x2138a6057b8>
```



```
In [81]: data[['c','sma10', 'sma25', 'sma100']].head()
Out[81]:
```

Date	c	sma10	sma25	sma100
2010-05-25	124.860001	126.306000	132.968401	129.4782
2010-05-26	123.209999	125.240000	132.039601	129.3651
2010-05-27	126.699997	124.763000	131.104001	129.2931
2010-05-28	125.459999	124.456000	130.377201	129.2008
2010-06-01	123.239998	123.888999	129.422401	129.1107

2.4 Exponential weighted moving average

An exponential moving average (EMA) is a type of moving average (MA) that places a greater weight and significance on the most recent data points. The exponential moving average is also referred to as the exponentially weighted moving average. An exponentially weighted moving average reacts more significantly to recent price changes than a simple moving average (SMA), which applies an equal weight to all observations in the period.
Traders often use several different EMA days, for instance, 20-day, 30-day, 90-day, and 200-day moving averages.

The Formula For EMA Is

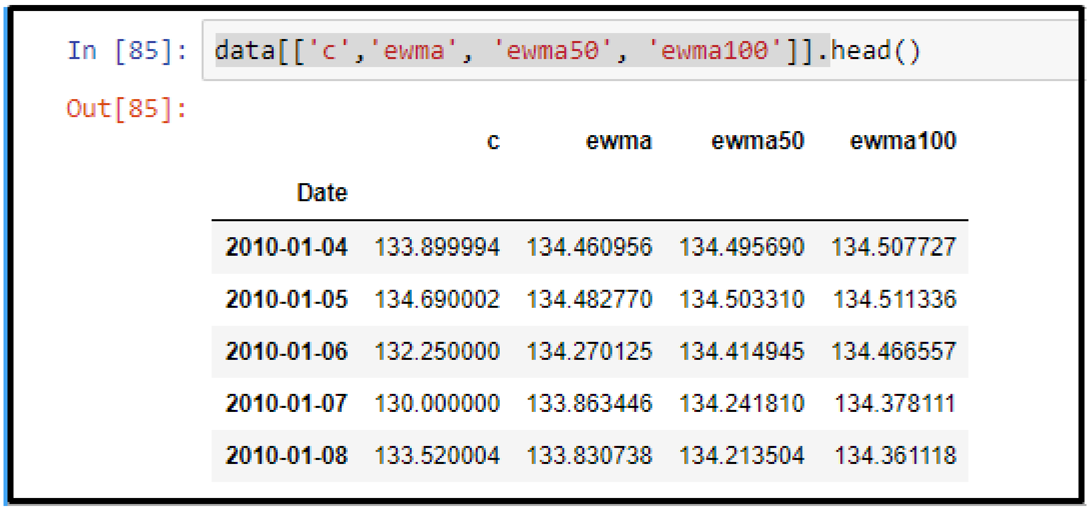
$$EMA_{\text{Today}} = \left(\text{Value}_{\text{Today}} * \left(\frac{\text{Smoothing}}{1+\text{Days}}\right)\right) + EMA_{\text{Yesterday}} * \left(1 - \left(\frac{\text{Smoothing}}{1+\text{Days}}\right)\right)$$

where:

$EMA = \text{Exponential moving average}$

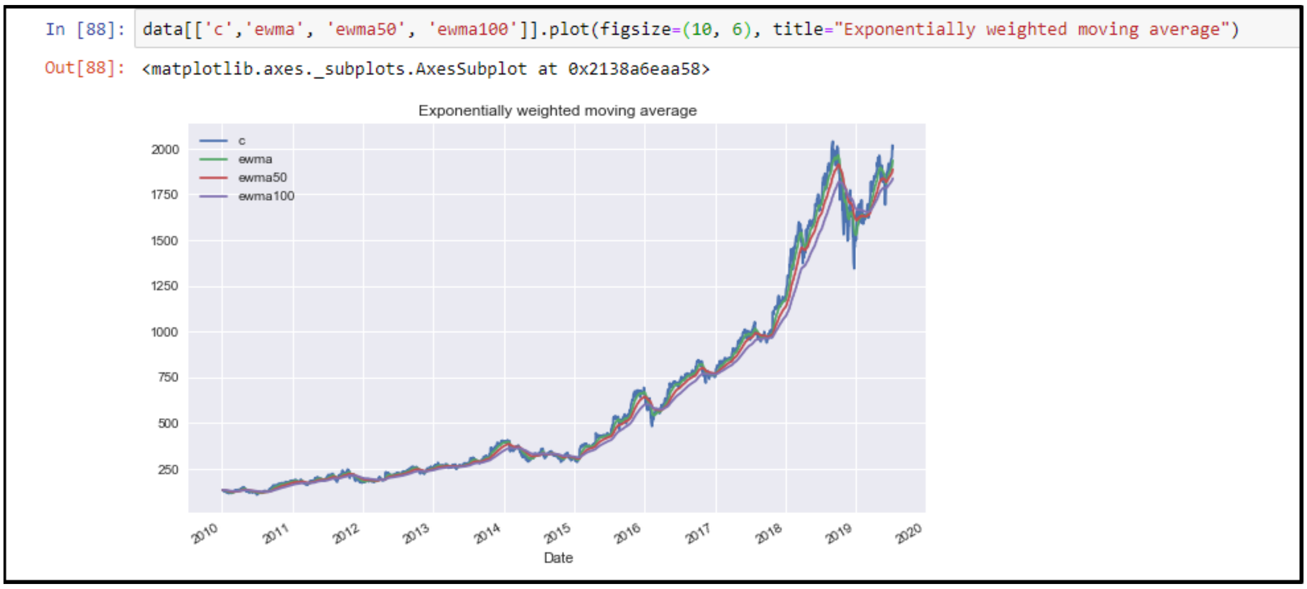
In python this can be computed very easily, the ewm method of dataframe takes span (which is period) as parameter and then specifying mean on it returns the EWMA.

```python
data['ewma'] = data['c'].ewm(span=20, adjust=False).mean()
data['ewma50'] = data['c'].ewm(span=50, adjust=False).mean()
data['ewma100'] = data['c'].ewm(span=100, adjust=False).mean()
```



```
In [85]: data[['c','ewma', 'ewma50', 'ewma100']].head()
Out[85]:
```

Date	c	ewma	ewma50	ewma100
2010-01-04	133.899994	134.460956	134.495690	134.507727
2010-01-05	134.690002	134.482770	134.503310	134.511336
2010-01-06	132.250000	134.270125	134.414945	134.466557
2010-01-07	130.000000	133.863446	134.241810	134.378111
2010-01-08	133.520004	133.830738	134.213504	134.361118



```
In [88]: data[['c','ewma', 'ewma50', 'ewma100']].plot(figsize=(10, 6), title="Exponentially weighted moving average")
Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0x2138a6eaa58>
```

2.5 RSI

The Relative Strength Index (RSI) is a momentum indicator developed by noted technical analyst Welles Wilder, that compares the magnitude of recent gains and losses over a specified time period to measure speed and change of price movements of a security. It is primarily used to identify overbought or oversold conditions in the trading of an asset.

The Relative Strength Index (RSI) is calculated as follows:

```
RSI = 100 - 100 / (1 + RS)
RS = Average gain in last 14 trading days / Average loss in last 14 trading days
```

RSI values range from 0 to 100. Traditional interpretation and usage of the RSI is that

- ➢ An RSI values of 70 or above indicate that a security is becoming overbought or overvalued, and therefore may be primed for a trend reversal or corrective pullback in price.
- ➢ An RSI reading of 30 or below is commonly interpreted as indicating an oversold or undervalued condition that may signal a trend change or corrective price reversal to the upside.

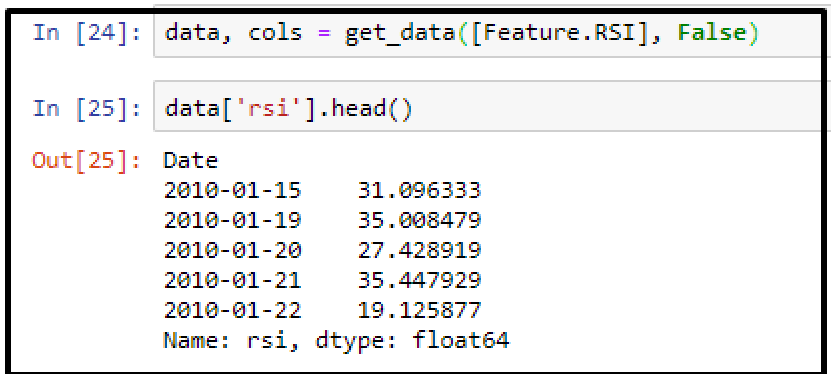
How does the RSI function work?

a. Function creates two series of daily differences.
b. One series is daily positive differences, i.e. gains.
c. One series is daily negative difference, i.e. losses.
d. Average daily positive differences for the period specified.
e. Average daily negative difference for the period specified.
f. RS = Exponential Moving Average of daily positive differences for the period specified / Exponential Moving Average of daily positive differences for the period specified.

Here is the code snippet

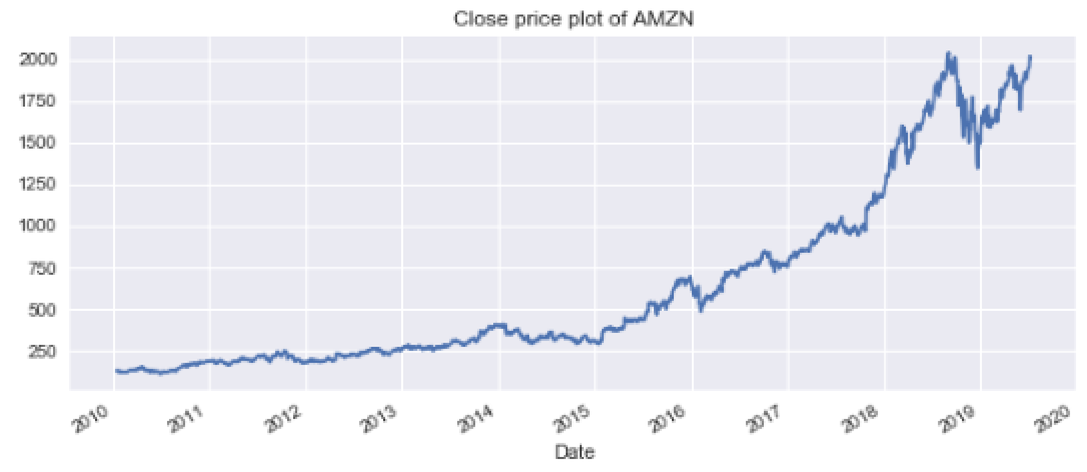
```
def RSI(series, period, normalize = True):
 delta = series.diff().dropna()
 u = delta * 0
 d = u.copy()
 u[delta > 0] = delta[delta > 0]
 d[delta < 0] = -delta[delta < 0]
 u[u.index[period-1]] = np.mean( u[:period] ) #This is sum of avg gains
 u = u.drop(u.index[:(period-1)])
 d[d.index[period-1]] = np.mean( d[:period] ) #This is sum of avg losses
 d = d.drop(d.index[:(period-1)])

 rs = u.ewm(span=period-1, adjust=False).mean() / d.ewm(span=period-1,
adjust=False).mean()

 rsi = 100 - 100 / (1 + rs)

 if normalize:
  rsi, mean, std  = gaussian(rsi)
 return rsi
```

Sample RSI data for AMZN stock

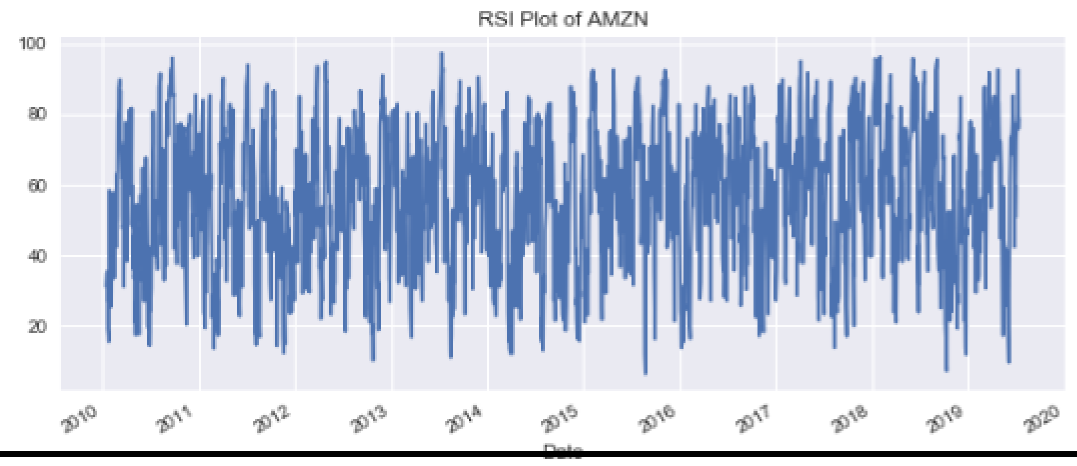


```
In [24]:  data, cols = get_data([Feature.RSI], False)

In [25]:  data['rsi'].head()

Out[25]:  Date
          2010-01-15    31.096333
          2010-01-19    35.008479
          2010-01-20    27.428919
          2010-01-21    35.447929
          2010-01-22    19.125877
          Name: rsi, dtype: float64
```

```
data['c'].plot(figsize=(10, 4), title="Close price plot of AMZN");
```



```
data['rsi'].plot(figsize=(10, 4), title="RSI Plot of AMZN");
```



2.6 MACD

Moving Average Convergence Divergence (MACD) is a trend-following momentum indicator that shows the relationship between two moving averages of a security's price. The MACD is calculated by subtracting the 26-period Exponential Moving Average (EMA) from the 12-period EMA.

The result of that calculation is the MACD line. A nine-day EMA of the MACD called the "signal line," is then plotted on top of the MACD line, which can function as a trigger for buy and sell signals. Traders may buy the security when the MACD crosses above its signal line and sell - or short - the security when the MACD crosses below the signal line. Moving Average Convergence Divergence (MACD) indicators can be interpreted in several ways, but the more common methods are crossovers, divergences, and rapid rises/falls.

The scope of this project is limited to MACD, we have used MACD as a feature in our training and classification

The Formula for MACD:

MACD = 12-Period EMA - 26-Period EMA

MACD is calculated by subtracting the long-term EMA (26 periods) from the short-term EMA (12 periods). An exponential moving average (EMA) is a type of moving average (MA) that places a greater weight and significance on the most recent data points. The exponential moving average is also referred to as the exponentially weighted moving average. An exponentially weighted moving average reacts more significantly to recent price changes than a simple moving average (SMA), which applies an equal weight to all observations in the period.

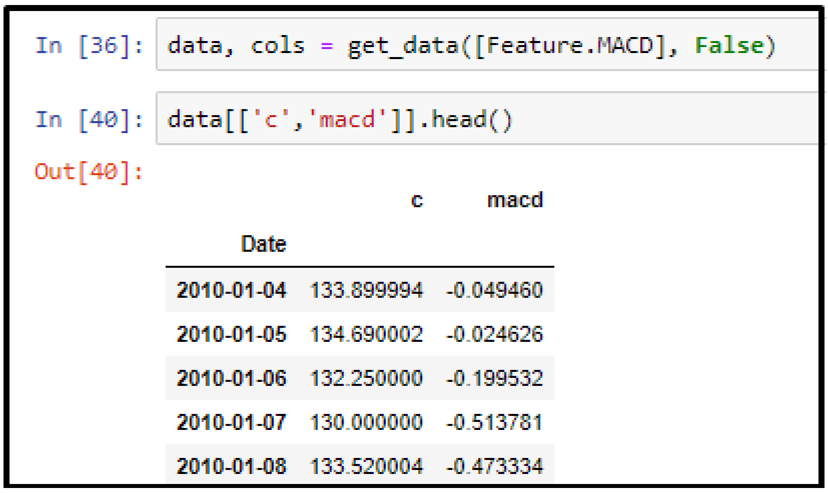
Here is the code snippet

```python
def MACD(series, period, normalize = True):
 exp1 = series.ewm(span=12, adjust=False).mean()
 exp2 = series.ewm(span=26, adjust=False).mean()
 macd = exp1-exp2

 if normalize:
  macd, mean, std = gaussian(macd)
 return macd
```

Interpreting values of MACD

The MACD has a positive value whenever the 12-period EMA (blue) is above the 26-period EMA (red) and a negative value when the 12-period EMA is below the 26-period EMA. The more distant the MACD is above or below its baseline indicates that the distance between the two EMAs is growing. In the following chart, you can see how the two EMAs applied to the price chart correspond to the MACD (blue) crossing above or below its baseline (red dashed) in the indicator below the price chart.

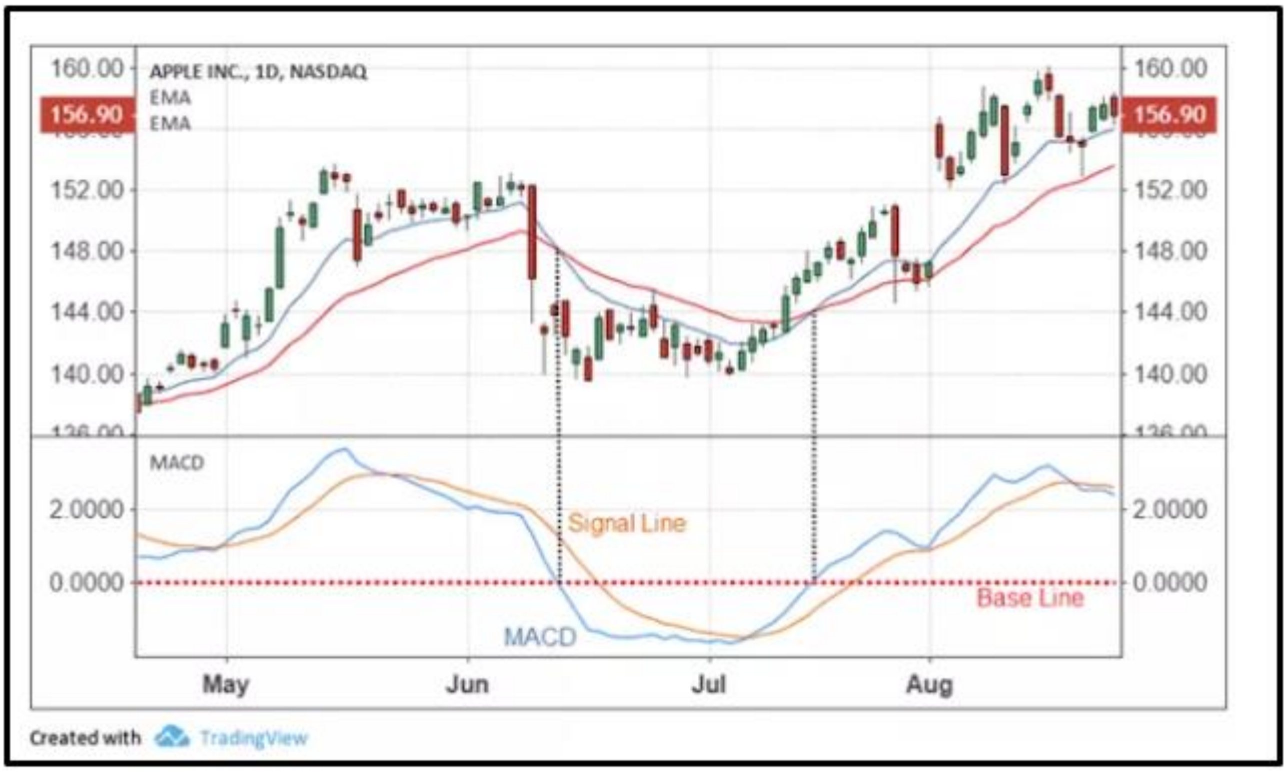
Sample MACD data for AMZN stock



```
In [36]: data, cols = get_data([Feature.MACD], False)

In [40]: data[['c','macd']].head()
Out[40]:
```

Date	c	macd
2010-01-04	133.899994	-0.049460
2010-01-05	134.690002	-0.024626
2010-01-06	132.250000	-0.199532
2010-01-07	130.000000	-0.513781
2010-01-08	133.520004	-0.473334

MACD and MACD signal line for APPL stock



2.7 Simple Moving average of Variance

As we used the simple moving average of the price of the stock as a feature. We are also interested in the simple moving average of the standard deviation or variance of the stock and this can form a feature in stock prediction.

This can be computed very easily in python.

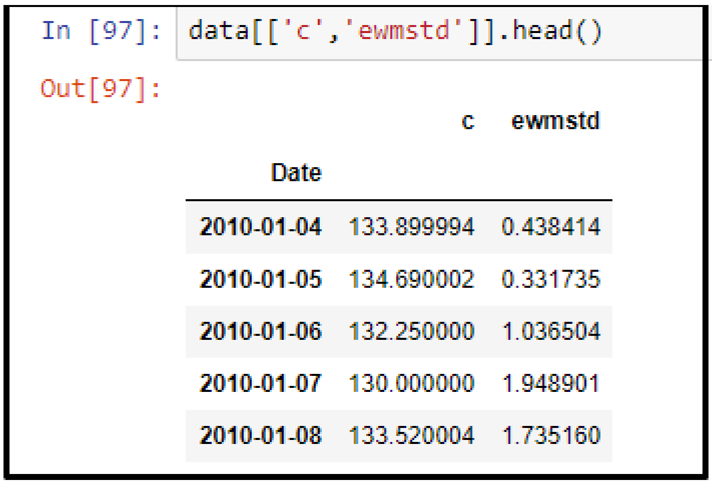
```
    data['var10'] = data['c'].rolling(10).std() # Simple moving average
    data['var25'] = data['c'].rolling(25).std()
```

2.8 EWMA on Variance

An exponential moving average (EMA) is a type of moving average (MA) that places a greater weight and significance on the most recent data points. In addition to the mean, we may also be interested in the variance and in the standard deviation to evaluate the statistical significance of a deviation from the mean.

In python the standard deviation of the Exponential moving average can be computed very easily

```
    data['ewmstd'] = data['c'].ewm(span=20, adjust=False).std()
```



```
In [97]: data[['c','ewmstd']].head()
Out[97]:
```

Date	c	ewmstd
2010-01-04	133.899994	0.438414
2010-01-05	134.690002	0.331735
2010-01-06	132.250000	1.036504
2010-01-07	130.000000	1.948901
2010-01-08	133.520004	1.735160

3 Model Evaluation

3.1 K-Fold Cross Validation

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample.

The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. When a specific value for k is chosen, it may be used in place of k in the reference to the model, such as k=10 becoming 10-fold cross-validation.

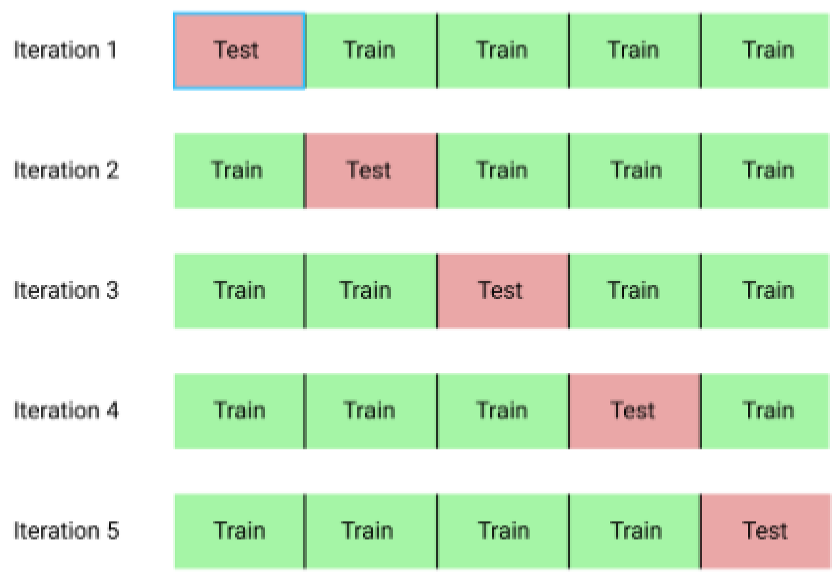
Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. That is, to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model.

It is a popular method because it is simple to understand and because it generally results in a less biased or less optimistic estimate of the model skill than other methods, such as a simple train/test split.

The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups
3. For each unique group:
   a. Take the group as a hold out or test data set
   b. Take the remaining groups as a training data set
   c. Fit a model on the training set and evaluate it on the test set
   d. Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

Below is the graphical illustration where the k=5 is set and the dataset will be divided into 5 equal parts and the below process will run 5 times, each time with a different holdout set.

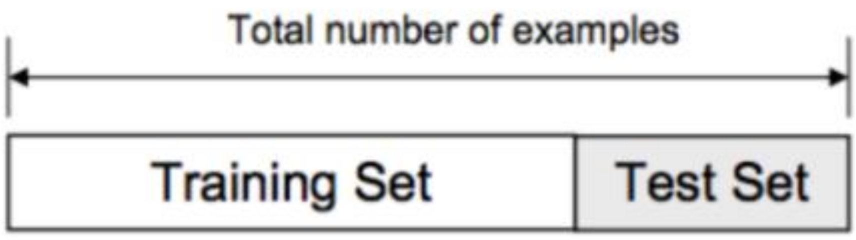


Here is the code snippet of K-Fold cross validation used in the Project

```
from sklearn import model_selection
from sklearn.model_selection import cross_val_score

kfold = model_selection.KFold(n_splits=5, random_state=7, shuffle=False)
# RandomState is the seed used by the RNG
crossval = model_selection.cross_val_score(lm, data[cols], data['d'], cv=kfold,
scoring='accuracy')
print("5-fold crossvalidation accuracy: %.4f" % (crossval.mean())) #average accuracy
```

3.2 Train Test split of data set

In statistics and machine learning the data we use is usually split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.



Train/Test Split

The python sklearn library offers convenient way to split the training data set into train and test sets. Here is the code snippet which split the data into 80% train and 20% test, the split ratio is controlled by the test_size variable.

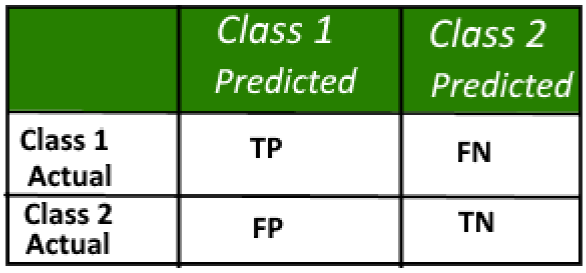
```
from sklearn.model_selection import train_test_split
from sklearn import model_selection
train, test = model_selection.train_test_split(data, test_size=0.2, shuffle=False)
```

Overfitting

The reason we do this split is to avoid Overfitting. Overfitting means that model we trained has trained "too well" and is now, well, fit too closely to the training dataset. This model will be very accurate on the training data but will probably be very not accurate on untrained or new data. So, we test on the untrained data to check the validity of the model.

3.3 Confusion Matrix

A confusion matrix is a summary of prediction results on a classification problem.
The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix.
The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.



	Class 1 Predicted	*Class 2 Predicted*
Class 1 Actual	**TP**	**FN**
Class 2 Actual	**FP**	**TN**

- True Positive (TP) : Observation is positive, and is predicted to be positive.
- False Negative (FN) : Observation is positive, but is predicted negative.
- True Negative (TN) : Observation is negative, and is predicted to be negative.
- False Positive (FP) : Observation is negative, but is predicted positive.

Here is the sample code from the project, this lines of codes are from LogisticRegression.

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(data['d'], data['predict'], labels=[-1,1])
print(confusion_matrix)
```

```
[[ 928  176]
 [ 161 1109]]
```

928 data samples corresponded to NegativeReturns(-1) and were predicted as NegativeReturns(-1)
176 data samples corresponded to NegativeReturns(-1) and were predicted as PositiveReturns(1)
161 data samples corresponds to PositiveReturns(1) and were predicted as NegativeReturns(-1)
1109 data samples corresponds to PositiveReturns(1) and were predicted as PositiveReturns(1)

3.4 Precision recall

As you train your classification predictive model, you will want to assess how good it is. Interestingly, there are many different ways of evaluating the performance. Most data scientists that use Python for predictive modeling use the Python package called scikit-learn. Scikit-learn contains many built-in functions for analyzing the performance of models. We will look at few of the metrics here.

Recall:
Recall can be defined as the ratio of the total number of correctly classified positive examples divide to the total number of positive examples. High Recall indicates the class is correctly recognized (small number of FN). Recall is given by the relation:

$$\textit{Recall} = \frac{TP}{TP + FN}$$

Precision:
To get the value of precision we divide the total number of correctly classified positive examples by the total number of predicted positive examples. High Precision indicates an example labeled as positive is indeed positive (small number of FP).
Precision is given by the relation:

$$\textit{Precision} = \frac{TP}{TP + FP}$$

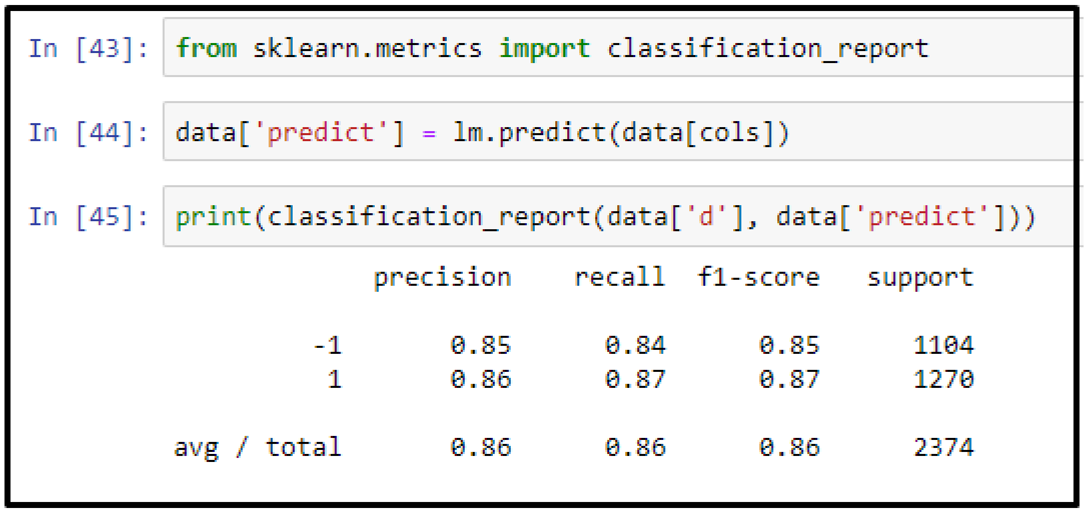
High recall, low precision: This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.

Low recall, high precision: This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP)

F-measure:
Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate an F-measure which uses Harmonic Mean in place of Arithmetic Mean as it punishes the extreme values more. The F-Measure will always be nearer to the smaller value of Precision or Recall.

$$F\text{ - }measure = \frac{2*\text{Recall}*\text{Precision}}{\text{Recall} + \text{Precision}}$$

Here is the code snippet from the project file '*1 Logistic Classifier and Bayesian Classifier.ipynb*'



```
In [43]: from sklearn.metrics import classification_report

In [44]: data['predict'] = lm.predict(data[cols])

In [45]: print(classification_report(data['d'], data['predict']))
                      precision    recall  f1-score   support

                   -1      0.85      0.84      0.85      1104
                    1      0.86      0.87      0.87      1270

          avg / total      0.86      0.86      0.86      2374
```

4 Predictions using Classifiers

This section explores various classifiers used in the project to predict the stock movement.

4.1 Logistic Regression

Regression methods are supervised-learning techniques. They try to explain a dependent variable in terms of independent variables. The independent variables are numerical and we fit straight lines, polynomials or other functions to predict dependent variables.

Logistic Regression: Logistic regression (or logit in econometrics) is used to forecast the probability of an event given historical sample data. For example, we may want to forecast the direction of nextday price movement given the observation of other market returns or value of quant signals today. By mapping the probability to either 0 or 1, logistic regression can also be used for classification problems, where we have to choose to either buy (mapped to 0) or sell (mapped to 1).

Logistic regression is derived via a simple change to ordinary linear regression.

We first form a linear combination of the input variables (as in conventional regression), and then apply a function that maps this number to a value between 0 and 1.

The mapping function is called the logistic function, and has the simple shape shown below (left). An alternative to the logistic regression would have to simply 'quantize' the output of a linear regressor to either 0 or 1, i.e. if the value was less than 0.5, we declare the output to be 0; and if its greater than 0.5, we declare it to be 1.

It turns out that such an approach is very sensitive to outliers. As shown in the figure below (right), the outliers cause the line to have an extremely low slope, which in turn leads to many points in the set being classified as 0, instead of being classified correctly as 1.

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

The implementation of logistic regression in scikit-learn can be accessed from class LogisticRegression. This implementation can fit binary, One-vs- Rest, or multinomial logistic regression with optional L2 or L1 regularization.

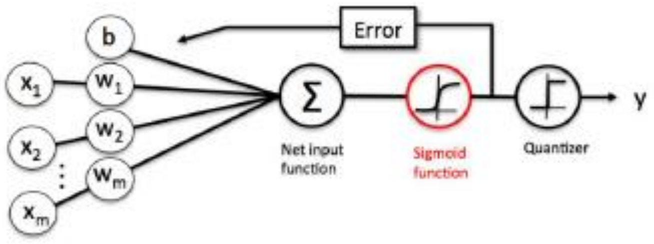
As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i (X_i^T w + c)) + 1).$$

Similarly, L1 regularized logistic regression solves the following optimization problem

$$\min_{w,c} \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i (X_i^T w + c)) + 1).$$

Note that, in this notation, it’s assumed that the observation yi takes values in the set –1, 1 at trial i.



$$\phi(z) = \frac{1}{1+e^{-z}},$$

where z is defined as the net input

$$z = w_0x_0 + w_1x_1 + \ldots + w_mx_m = \sum_{j=0}^{m} w_jx_j = \mathbf{w}^T\mathbf{x}.$$

The net input is in turn based on the logit function

$$logit(p(y=1 \mid \mathbf{x})) = z.$$

[Code snippet from Project for Logistic Regression]

In this project the file ‘’ has the code for the Logistic Regression based classification of AMZN Stock. As with any training the first step is to extract features, so the get_data method downloads the historical stock prices of AMZN and builds/extracts the features, the type of features to be extracted is passed as input. For the Logistic regression we extract the features of type OHLC, LAGGED_RETURNS, SMA(Simple moving average), RSI, MACD

```
data, cols = get_data(
    [Feature.OHLC,
     Feature.LAGGED_RETURNS,
     Feature.SMA_PRICE,
     Feature.SMA_VAR,
     Feature.RSI,
     Feature.MACD,
     Feature.EWMA_PRICE,
     Feature.EWMA_VAR])
```

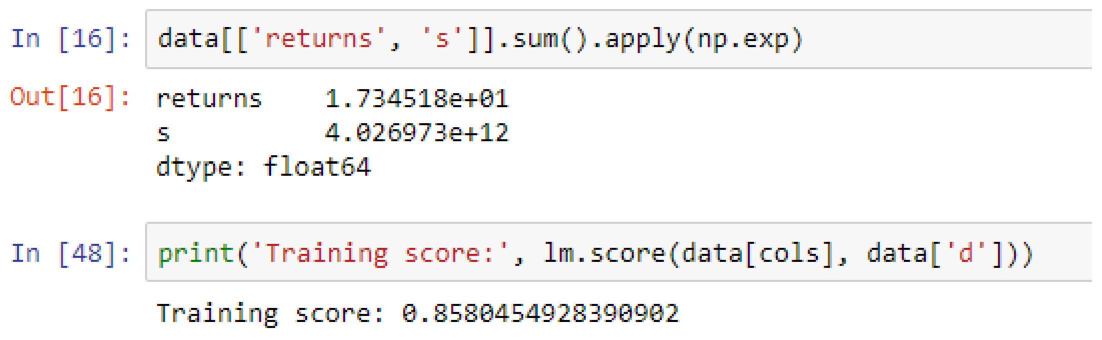
Train the model

```
from sklearn import linear_model
```

```
lm = linear_model.LogisticRegression(C = 1e6)
%time lm.fit(data[cols], data['d'])
```

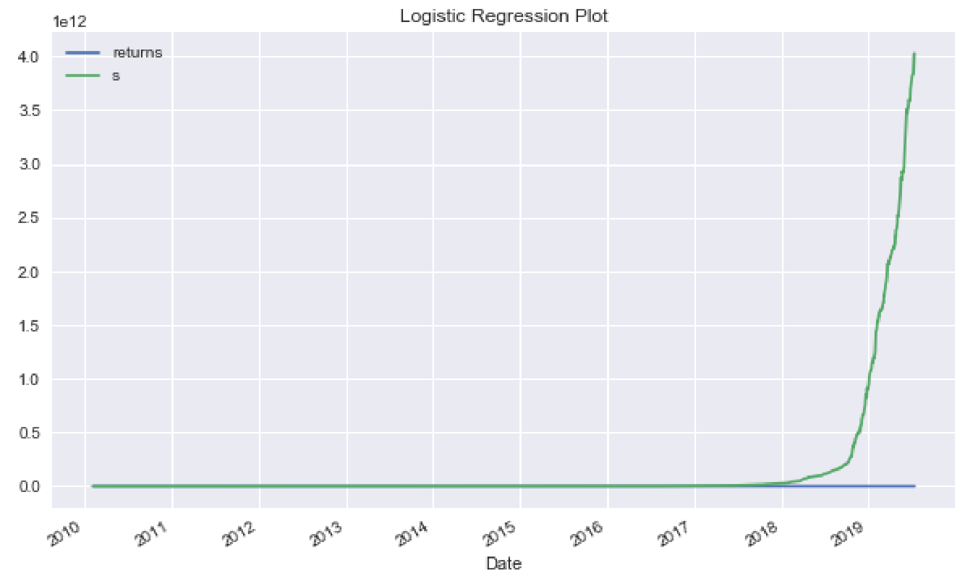
On the Trained model Predict the results

```
data['p'] = lm.predict(data[cols])
data['p'] = np.where(data['p'] > 0, 1, -1)
data['s'] = data['p'] * data['returns']
```



```
In [16]: data[['returns', 's']].sum().apply(np.exp)

Out[16]: returns    1.734518e+01
         s          4.026973e+12
         dtype: float64

In [48]: print('Training score:', lm.score(data[cols], data['d']))

         Training score: 0.8580454928390902
```

```
data[['returns', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6), title="Logistic Regression Plot");
```



Split the data b/w train and test

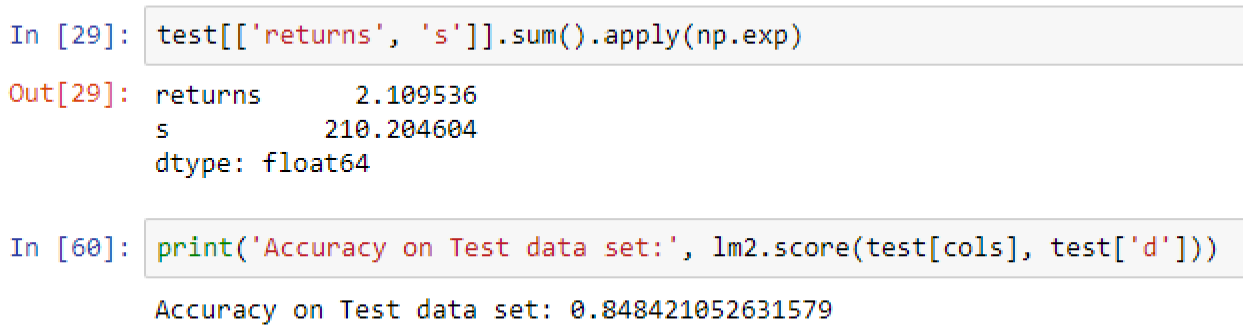
The data is split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.

```
from sklearn.model_selection import train_test_split
from sklearn import model_selection
train, test = model_selection.train_test_split(data, test_size=0.2, shuffle=False)
```

```
print("Train count {}, Test count {}".format(len(train), len(test)))
lm2 = linear_model.LogisticRegression(C = 1e6)
%time lm2.fit(train[cols], train['d'])
```

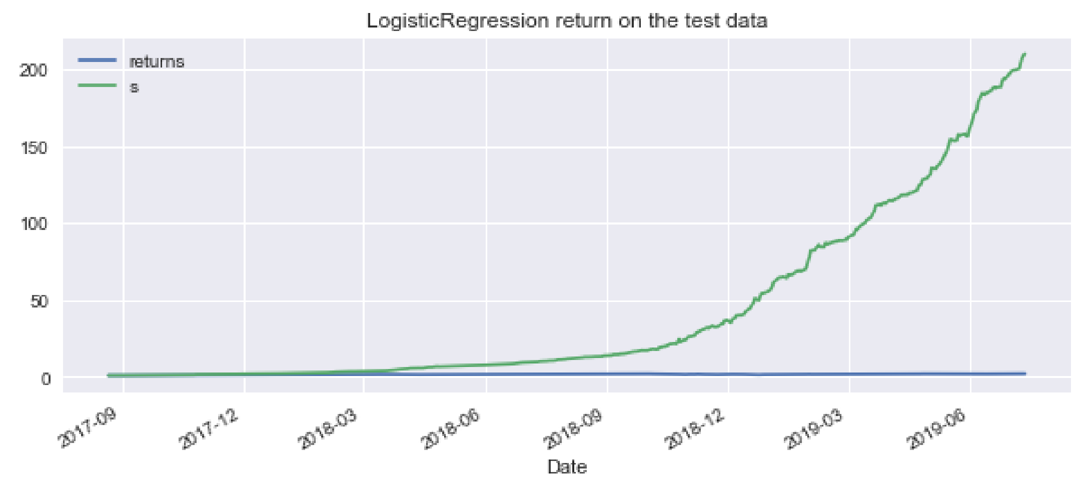
```
test['p'] = lm2.predict(test[cols])
test['p'] = np.where(test['p'] > 0, 1, -1)
```

```
test['s'] = test['p'] * test['returns']
```



```
In [29]:  test[['returns', 's']].sum().apply(np.exp)

Out[29]:  returns      2.109536
          s          210.204604
          dtype: float64

In [60]:  print('Accuracy on Test data set:', lm2.score(test[cols], test['d']))

          Accuracy on Test data set: 0.848421052631579
```

```
In [51]: test[['returns', 's']].cumsum().apply(np.exp).plot(figsize=(10, 4),
                  title="LogisticRegression return on the test data");
```



4.1.1 Logistic Regression with L1 Penalty & different Regularization strength

In the code below we run a logistic regression with a L1 penalty four times, each time decreasing the value of C. We should expect that as C decreases, more coefficients become 0.

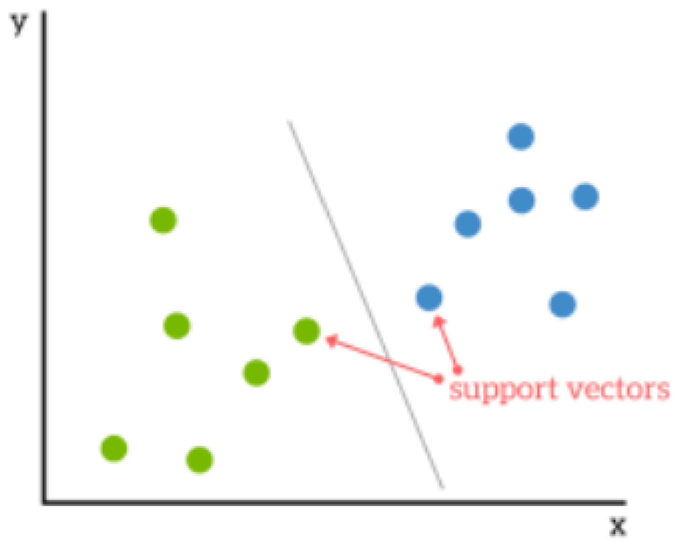
```
C = [100, 10, .1, 0.01]

for c in C:
    clf = linear_model.LogisticRegression(penalty='l1', C=c, solver='liblinear')
    clf.fit(data[cols], data['d'])
    print('C:', c)
    print('Coefficient of first 4 feature:', clf.coef_[0][1:5])
    print('Training accuracy:', clf.score(data[cols], data['d']))
    print('')
```

```
C: 100
Coefficient of first 4 feature: [ 0.27735806 -1.42605294 -1.15337605 -0.91393988]
Training accuracy: 0.858887952822241

C: 10
Coefficient of first 4 feature: [ 0.26561756 -1.40920142 -1.13828519 -0.90226635]
Training accuracy: 0.8597304128053918

C: 0.1
Coefficient of first 4 feature: [ 0.         -1.0925647  -0.85823433 -0.68128319]
Training accuracy: 0.8567818028643639

C: 0.01
Coefficient of first 4 feature: [ 0.         -0.13554079 -0.0476762   0.        ]
Training accuracy: 0.8066554338668913
```

4.2 Support Vector Machines

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems and as such, this is what we will focus on in this post.

SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



Support Vectors

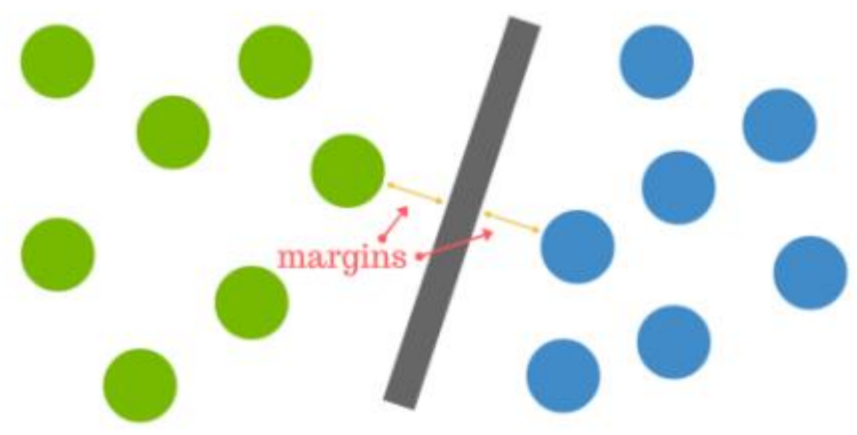
Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

What is a hyperplane?

As a simple example, for a classification task with only two features (like the image above), you can think of a hyperplane as a line that linearly separates and classifies a set of data. Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it. So when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.

How do we find the right hyperplane?

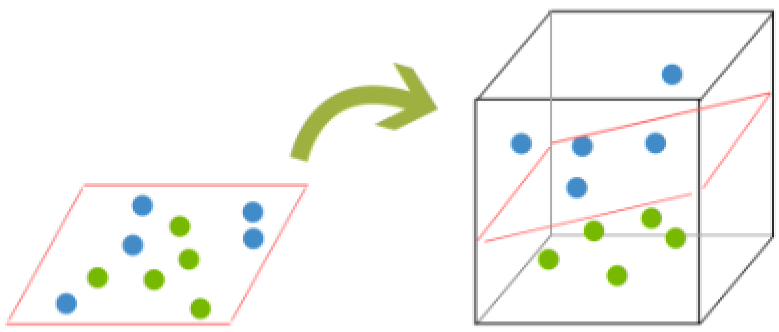
Or, in other words, how do we best segregate the two classes within the data?
The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly.



But what happens when there is no clear hyperplane?

This is where it can get tricky. Data is rarely ever as clean as our simple example above. A dataset will often look more like the jumbled balls below which represent a linearly non separable dataset.

In order to classify a dataset like the one above it's necessary to move away from a 2d view of the data to a 3d view. While the balls are up in the air, you use the sheet to separate them. This 'lifting' of the balls represents the mapping of data into a higher dimension. This is known as kernelling.

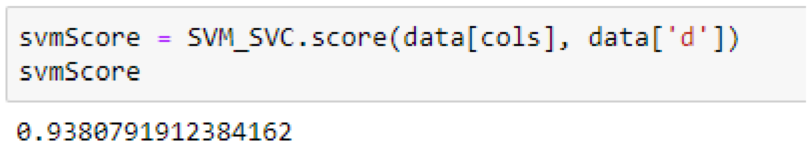


[Code snippet from Project for SVM]

In this project the file ‘’ has the code for the Logistic Regression based classification of AMZN Stock. As with any training the first step is to extract features, so the get_data method downloads the historical stock prices of AMZN and builds/extracts the features, the type of features to be extracted is passed as input. For the Logistic regression we extract the features of type OHLC, LAGGED_RETURNS, SMA(Simple moving average), RSI, MACD

```
data, cols = get_data(
    [Feature.OHLC,
     Feature.LAGGED_RETURNS,
     Feature.SMA_PRICE,
     Feature.SMA_VAR,
     Feature.RSI,
     Feature.MACD,
     Feature.EWMA_PRICE,
     Feature.EWMA_VAR])
```

```
from sklearn.svm import SVC
# Run Support Vector Classification
C=1e1
SVM_SVC = SVC(C=C, probability=True)
SVM_SVC.fit(data[cols], data['d']) ###FITTING DONE HERE
SVM_SVC.predict_proba(data[cols])
```



```
svmScore = SVM_SVC.score(data[cols], data['d'])
svmScore
```

```
0.9380791912384162
```

The score of the trained data is 93%, this is expected to be good as we are trying to predict the same data which we had used earlier for training.

Cross Validation Score

The ultimate goal of a Machine Learning Engineer or a Data Scientist is to develop a Model in order to get Predictions on New Data or Forecast some events for future on Unseen data. A Good Model is not the one that gives accurate predictions on the known data or training data but the one which gives good predictions on the new data and avoids overfitting and underfitting.

Cross-validation score provides a better measure of the skill of a machine learning model, as this runs on unseen data.

```
from sklearn import model_selection
from sklearn.model_selection import cross_val_score

# RandomState is the seed used by the RNG
kfold = model_selection.KFold(n_splits=5, random_state=7, shuffle=False)
crossval = model_selection.cross_val_score(SVM_SVC, data[cols], data['d'], cv=kfold,
scoring='accuracy')
```

```
print("5-fold crossvalidation accuracy: %.4f" % (crossval.mean())) #average accuracy
```

```
5-fold crossvalidation accuracy: 0.8496
```

The K-Fold cross validation score with K=5 for the KNN model is 84.5%

4.3 K-Nearest Neighbours (KNN)

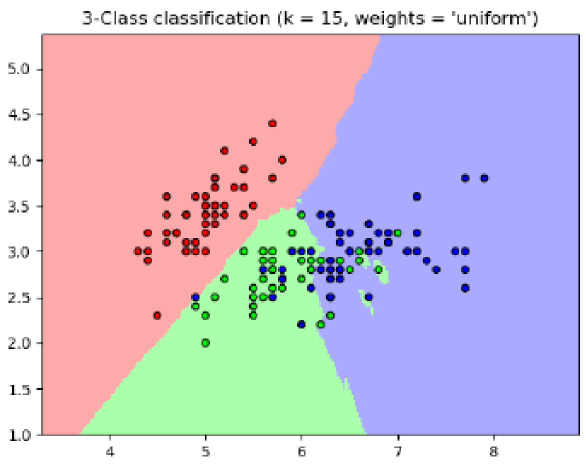
Neighbors-based classification is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

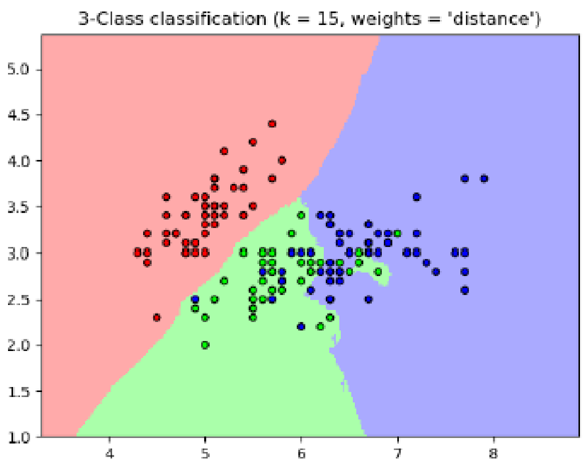
scikit-learn implements two different nearest neighbors classifiers: KNeighborsClassifier implements learning based on the nearest neighbors of each query point, where*k*is an integer value specified by the user. RadiusNeighborsClassifier implements learning based on the number of neighbors within a fixed radius of each training point, where is a floating-point value specified by the user.

The K-neighbors classification in KNeighborsClassifier is the most commonly used technique. The optimal choice of the value is highly data-dependent: in general, a larger suppresses the effects of noise, but makes the classification boundaries less distinct.

In cases where the data is not uniformly sampled, radius-based neighbors classification in RadiusNeighborsClassifier can be a better choice. The user specifies a fixed radius , such that points in sparser neighborhoods use fewer nearest neighbors for the classification. For high-dimensional parameter spaces, this method becomes less effective due to the so-called “curse of dimensionality”.

The basic nearest neighbors classification uses uniform weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors. Under some circumstances, it is better to weight the neighbors such that nearer neighbors contribute more to the fit. This can be accomplished through the weights keyword. The default value, weights = 'uniform', assigns uniform weights to each neighbor. weights = 'distance' assigns weights proportional to the inverse of the distance from the query point. Alternatively, a user-defined function of the distance can be supplied to compute the weights.





[Code snippet from Project for KNeighborsClassifier]

In this project the file ‘4 KNN Classifier.ipynb’ has the code for the Logistic Regression based classification of AMZN Stock. As with any training the first step is to extract features, so the get_data method downloads the historical stock prices of AMZN and builds/extracts the features, the type of features to be extracted is passed as input. For the Logistic regression we extract the features of type OHLC, LAGGED_RETURNS, SMA(Simple moving average), RSI, MACD

```
data, cols = get_data(
    [Feature.OHLC,
     Feature.LAGGED_RETURNS,
     Feature.SMA_PRICE,
     Feature.SMA_VAR,
     Feature.RSI,
     Feature.MACD,
     Feature.EWMA_PRICE,
     Feature.EWMA_VAR])
```

Train the model

The important part of the KNN is choosing the value of K, in this case we have made a guess and have started with the value of k = 3

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 3)
%time knn.fit(data[cols], data['d'])
```

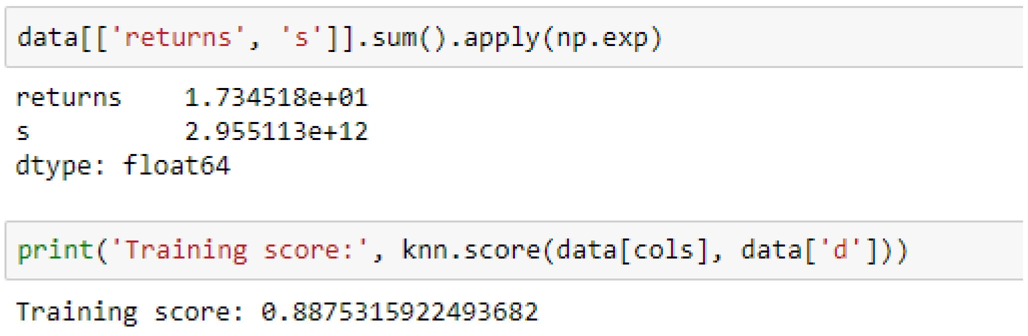
On the Trained model Predict the results

```
data['p'] = lm.predict(data[cols])
data['p'] = np.where(data['p'] > 0, 1, -1)
```

Compute the Returns from the prediction

```
data['s'] = data['p'] * data['returns']
```

When the prediction is +ve and the returns are +ve then we make a return also when the prediction is -ve and the returns are -ve then this will have a +ve results for us, because equipped with this knowledge we go short in the market if we know the market will have a negative return. Thus, using this knowledge, we will make better returns than the stock over the time period if our prediction accuracy is good.



```
data[['returns', 's']].sum().apply(np.exp)
```

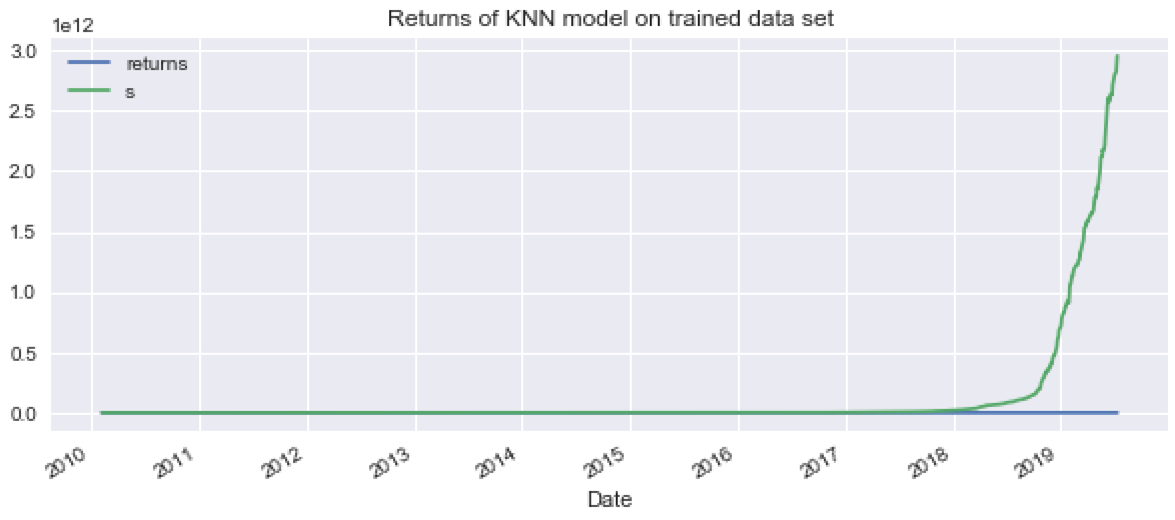
```
returns    1.734518e+01
s          2.955113e+12
dtype: float64
```

```
print('Training score:', knn.score(data[cols], data['d']))
```

```
Training score: 0.8875315922493682
```

This model has performed very well on the trained data set, the resultant accuracy is 88%

```
data[['returns', 's']].cumsum().apply(np.exp).plot(figsize=(10, 4),
                                     title="Returns of KNN model on trained data set");
```



Split the data b/w train and test

The data is split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.

```
from sklearn.model_selection import train_test_split
from sklearn import model_selection
train, test = model_selection.train_test_split(data, test_size=0.2, shuffle=False)
```

```
print("Train count {}, Test count {}".format(len(train), len(test)))
knn2 = KNeighborsClassifier(n_neighbors = 3)
%time knn2.fit(train[cols], train['d'])
```

```
test['p'] = knn2.predict(test[cols])
test['p'] = np.where(test['p'] > 0, 1, -1)
```

```
test['s'] = test['p'] * test['returns']
```

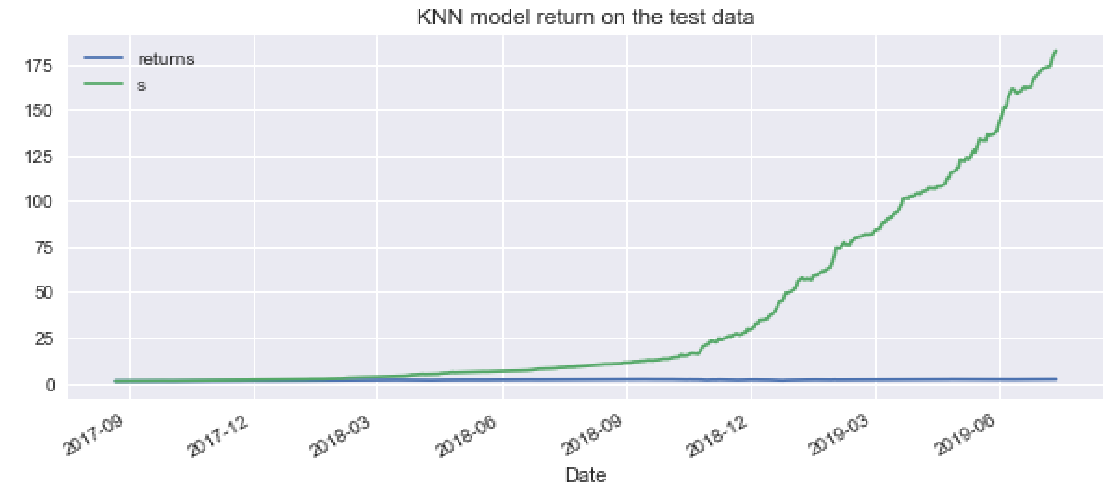
```
test[['returns', 's']].sum().apply(np.exp)
```

```
returns      2.109536
s          182.255880
dtype: float64
```

```
print('Accuracy on Test data set:', knn2.score(test[cols], test['d']))
```

```
Accuracy on Test data set: 0.7978947368421052
```

```
test[['returns', 's']].cumsum().apply(np.exp).plot(figsize=(10, 4),
        title="KNN model return on the test data");
```



K-Fold CrossValidation Score

```
kfold = model_selection.KFold(n_splits=5, random_state=7, shuffle=True) # RandomState is the seed used by the RNG
print(kfold)
print()
crossval = model_selection.cross_val_score(knn, data[cols], data['d'], cv=kfold, scoring='accuracy')
print("5-fold crossvalidation accuracy: %.4f" % (crossval.mean())) #average accuracy
```

```
KFold(n_splits=5, random_state=7, shuffle=True)

5-fold crossvalidation accuracy: 0.7864
```

Using cross-validation, our mean score is about 78.64%. This is a more accurate representation of how our model will perform on unseen data than our earlier testing using the holdout method.

Hypertuning model parameters using GridSearchCV

When we built our initial k-NN model, we set the parameter ‘n_neighbors’ to 3 as a starting point with no real logic behind that choice. Hypertuning parameters is when you go through a process to find the optimal parameters for your model to improve accuracy. In our case, we will use GridSearchCV to find the optimal value for 'n_neighbors'.

```
#create new a knn model
knn2 = KNeighborsClassifier()
#create a dictionary of all values we want to test for n_neighbors
param_grid = {'n_neighbors': np.arange(1, 25)}
#use gridsearch to test all values for n_neighbors
knn_gscv = GridSearchCV(knn2, param_grid, cv=5)
#fit model to data
knn_gscv.fit(data[cols], data['d'])
```

```
#check top performing n_neighbors value
knn_gscv.best_params_
```

```
{'n_neighbors': 17}
```

```
#check mean score for the top performing value of n_neighbors
knn_gscv.best_score_
```

```
0.7940185341196293
```

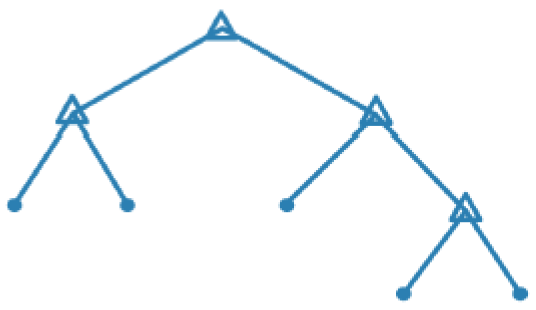
{'n_neighbors': 17}

We can see that 17 is the optimal value for **'n_neighbors'**. We can use the 'best_score_' function to check the accuracy of our model when 'n_neighbors' is 17. **'best_score_'** outputs the mean accuracy of the scores obtained through cross-validation which is **79.4%**

4.4 Decision Tree and Ensemble Methods (AdaBoost)

A **decision tree** lets you predict responses to data by following the decisions in the tree from the root (beginning) down to a leaf node. A tree consists of branching conditions where the value of a predictor is compared to a trained weight. The number of branches and the values of weights are determined in the training process. Additional modification, or pruning, may be used to simplify the model.

This is best used when you need an algorithm that is easy to interpret and fast to fit, to minimize memory usage



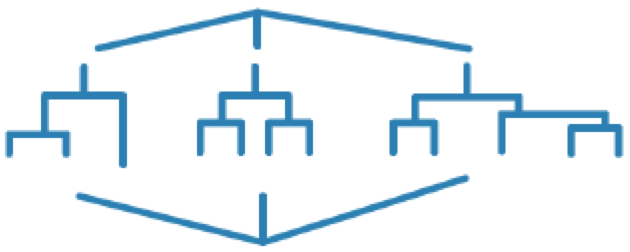
AdaBoost or Bagged Decision Tree

In these ensemble methods, several "weaker" decision trees are combined into a "stronger" ensemble. A bagged decision tree consists of trees that are trained independently on data that is bootstrapped from the input data. Boosting involves creating a strong learner by iteratively adding "weak" learners and adjusting the weight of each weak learner to focus on misclassified examples.

Boosting is an ensemble technique that creates strong classifier after combining several "weaker" decision trees.

- First, a model is fitted from the training data, then
- the second model created that attempts to correct the errors from the first model.

AdaBoost was the first successful boosting algorithm developed for binary classification. It is a good starting point for the understanding of boosting.



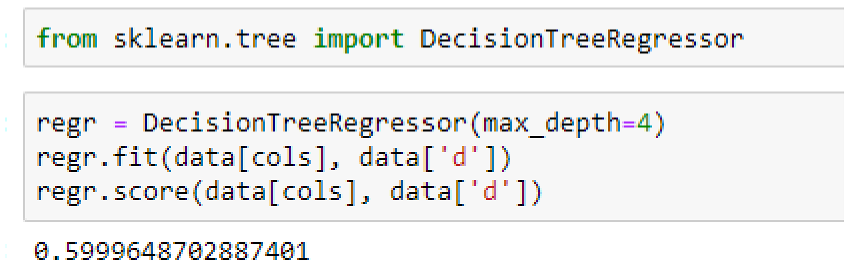
A pictorial representation of how AdaBoost several weak decision tree to create a strong decision tree.

The Jupyter notebook '3 Decision-Tree.ipynb' contains the code for the below Decision tree based algorithms.

a. DecisionTreeRegressor
b. AdaBoostRegressor
c. DecisionTreeClassifier
d. AdaBoostClassifier

DecisionTreeRegressor

Code snippet from the project's Jupyter notebook file '3 Decision-Tree.ipynb'



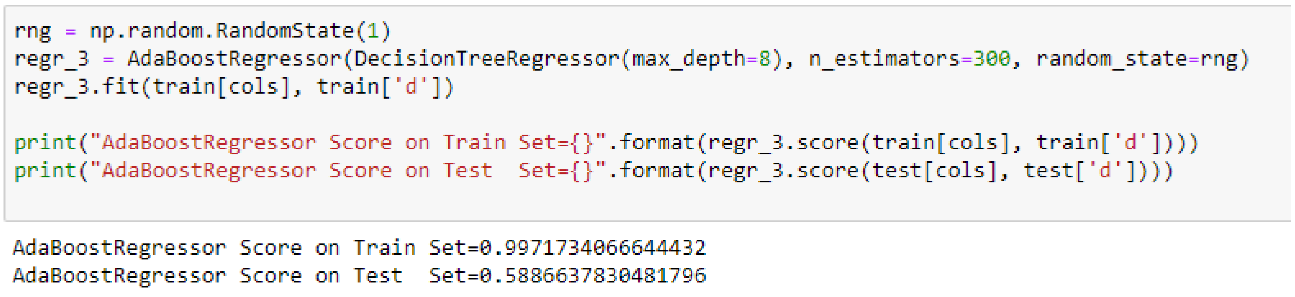
```
from sklearn.tree import DecisionTreeRegressor
```

```
regr = DecisionTreeRegressor(max_depth=4)
regr.fit(data[cols], data['d'])
regr.score(data[cols], data['d'])
```

```
0.5999648702887401
```

AdaBoostRegressor

Code snippet from the project's Jupyter notebook file '3 Decision-Tree.ipynb', this shows the performance of the AdaBoostRegressor. The score on the train set is whopping 99% and on the test set it is 58%



```
rng = np.random.RandomState(1)
regr_3 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=8), n_estimators=300, random_state=rng)
regr_3.fit(train[cols], train['d'])

print("AdaBoostRegressor Score on Train Set={}".format(regr_3.score(train[cols], train['d'])))
print("AdaBoostRegressor Score on Test  Set={}".format(regr_3.score(test[cols], test['d'])))
```

```
AdaBoostRegressor Score on Train Set=0.9971734066644432
AdaBoostRegressor Score on Test  Set=0.5886637830481796
```

4.5 ANN and Deep Learning

Inspired by the human brain, a neural network consists of highly connected networks of neurons that relate the inputs to the desired outputs. The network is trained by iteratively modifying the strengths of the connections so that given inputs map to the correct response.

Best Used...

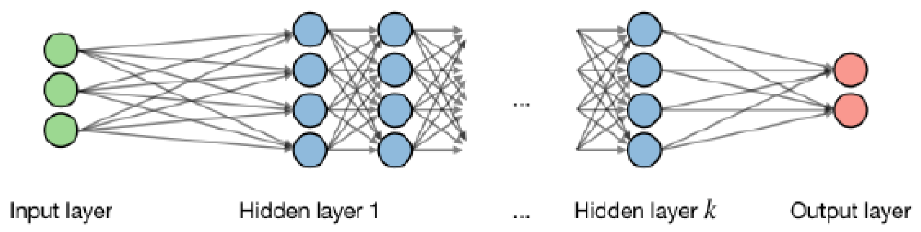
- For modeling highly nonlinear systems
- When data is available incrementally and you wish to constantly update the model
- When there could be unexpected changes in your input data
- When model interpretability is not a key concern

The classical approach to training neural networks is to minimize a (regularized) loss using backpropagation , a gradient descent method specialized to neural networks. Modern versions of backpropagation rely on stochastic gradient descent (SGD) to efficiently approximate the gradient for massive datasets. While SGD has been rigorously analyzed only for convex loss functions, in deep learning the loss is a non-convex function of the network parameters, hence there are no guarantees that SGD finds the global minimizer. In practice, there is overwhelming evidence that SGD routinely yields good solutions for deep networks.

Neural Networks

Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks.

❒ **Architecture** – The vocabulary around neural networks architectures is described in the figure below:



By noting i the i^{th} layer of the network and j the j^{th} hidden unit of the layer, we have:

$$\boxed{z_j^{[i]} = {w_j^{[i]}}^T x + b_j^{[i]}}$$

where we note w, b, z the weight, bias and output respectively.

[Code snippet of ANN and Deep Learning from Project]

In this project the file 'ANN Maket Prediction.ipynb' has the code for the ANN and deep learning based classification of AMZN Stock. As with any training the first step is to extract features, so the get_data method downloads the historical stock prices of AMZN and builds/extracts the features, the type of features to be extracted is passed as input. For the Logistic regression we extract the features of type OHLC, LAGGED_RETURNS, SMA(Simple moving average), RSI, MACD

```
data, cols = get_data(
    [Feature.OHLC,
     Feature.LAGGED_RETURNS,
     Feature.SMA_PRICE,
     Feature.SMA_VAR,
     Feature.RSI,
     Feature.MACD,
     Feature.EWMA_PRICE,
     Feature.EWMA_VAR])
```

Train the model

The Tensor Flow library has been used in this project for the DNN

```
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)
mean = data['returns'].mean()
std = data['returns'].std()
```

Set the boundaries of the neural network

```
fc = tf.contrib.layers.real_valued_column('returns', dimension=len(cols))
fcb = [tf.contrib.layers.bucketized_column(fc, boundaries=[mean-std, mean,
mean+std])]
```

The DNNClasifier class is created with hidden units value set

```
## The hidden_units values can be tweaked
model = tf.contrib.learn.DNNClassifier(hidden_units=[50, 50],
                                       feature_columns=fcb)
```

A method provides the features for training in the format required by the tensor flow

```
def get_input_data():
    fc = {'returns': tf.constant(data[cols].values)}
    # This gives us True or False and then we convert to 1 or 0
    la = tf.constant((data['returns'] > 0).astype(int).values,
                     shape=[len(data), 1])
    return fc, la
```

This is the training step

```
model.fit(input_fn=get_input_data, steps=100)
model.evaluate(input_fn=get_input_data, steps=1)
model.evaluate(input_fn=get_input_data, steps=1)
```

```
{'loss': 0.36898446,
 'accuracy': 0.853412,
 'labels/prediction_mean': 0.5343855,
 'labels/actual_label_mean': 0.5349621,
 'accuracy/baseline_label_mean': 0.5349621,
 'auc': 0.91231847,
 'auc_precision_recall': 0.9198653,
 'accuracy/threshold_0.500000_mean': 0.853412,
 'precision/positive_threshold_0.500000_mean': 0.8784893,
 'recall/positive_threshold_0.500000_mean': 0.8425197,
 'global_step': 100}
```

```
data['dnn_pred'] = list(model.predict(input_fn=get_input_data))
data['dnn_pred'] = np.where(data['dnn_pred'] > 0, 1.0, -1.0)
```

```
data['dnn_returns'] = data['returns'] * data['dnn_pred']
```

```
data[['returns', 'dnn_returns']].sum().apply(np.exp)
```

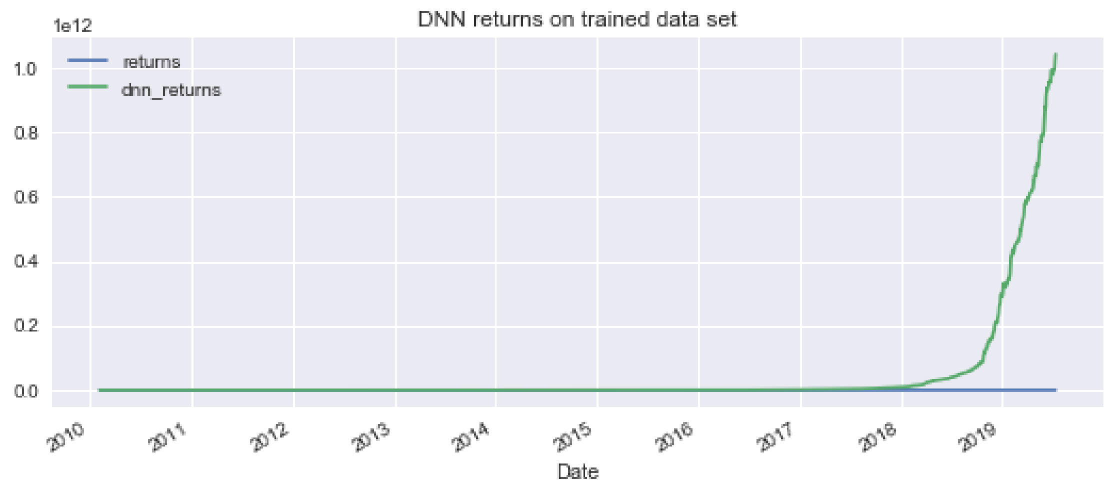
```
returns        1.734518e+01
dnn_returns    1.040908e+12
dtype: float64
```

Note the line

```
data['dnn_returns'] = data['returns'] * data['dnn_pred']
```

When the prediction is +ve and the returns are +ve then we make a return also when the prediction is -ve and the returns are -ve then this will have a +ve results for us, because equipped with this knowledge we go short in the market if we know the market will have a negative return. Thus, using this knowledge, we will make better returns than the stock over the time period if our prediction accuracy is good.

```
data[['returns', 'dnn_returns']].cumsum(
        ).apply(np.exp).plot(figsize=(10, 4), title="DNN returns on trained data set");
```



Split the data b/w train and test

The data is split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.

Split the data into train and test

```
import tensorflow as tf
from sklearn import model_selection
train, test = model_selection.train_test_split(data, test_size=0.2, shuffle=False)
train = train.copy()
test = test.copy()
mean2 = train['returns'].mean()
std2 = train['returns'].std()
```

Train the DNN model with the trained data set

```
tf.logging.set_verbosity(tf.logging.ERROR)
fc2 = tf.contrib.layers.real_valued_column('returns', dimension=len(cols))
fcb2 = [tf.contrib.layers.bucketized_column(fc2, boundaries=[mean2-std2, mean2,
mean2+std2])]

model_test = tf.contrib.learn.DNNClassifier(hidden_units=[50, 50],
                                       feature_columns=fcb2)

def get_train_data():
    fc = {'returns': tf.constant(train[cols].values)}
    # This gives us True or False and then we convert to 1 or 0
    la = tf.constant((train['returns'] > 0).astype(int).values,
                     shape=[len(train), 1])
    return fc, la

def get_test_data():
    fc = {'returns': tf.constant(test[cols].values)}
    # This gives us True or False and then we convert to 1 or 0
    la = tf.constant((test['returns'] > 0).astype(int).values,
                     shape=[len(test), 1])
    return fc, la

model_test.fit(input_fn=get_train_data, steps=100)
```

```
: model_test.evaluate(input_fn=get_test_data, steps=1)
```

```
: {'loss': 0.39896312,
   'accuracy': 0.8357895,
   'labels/prediction_mean': 0.5462776,
   'labels/actual_label_mean': 0.57473683,
   'accuracy/baseline_label_mean': 0.57473683,
   'auc': 0.897318,
   'auc_precision_recall': 0.9263208,
   'accuracy/threshold_0.500000_mean': 0.8357895,
   'precision/positive_threshold_0.500000_mean': 0.90123457,
   'recall/positive_threshold_0.500000_mean': 0.8021978,
   'global_step': 100}
```

```
: test['dnn_pred'] = list(model_test.predict(input_fn=get_test_data))
  test['dnn_pred'] = np.where(test['dnn_pred'] > 0, 1.0, -1.0)
```

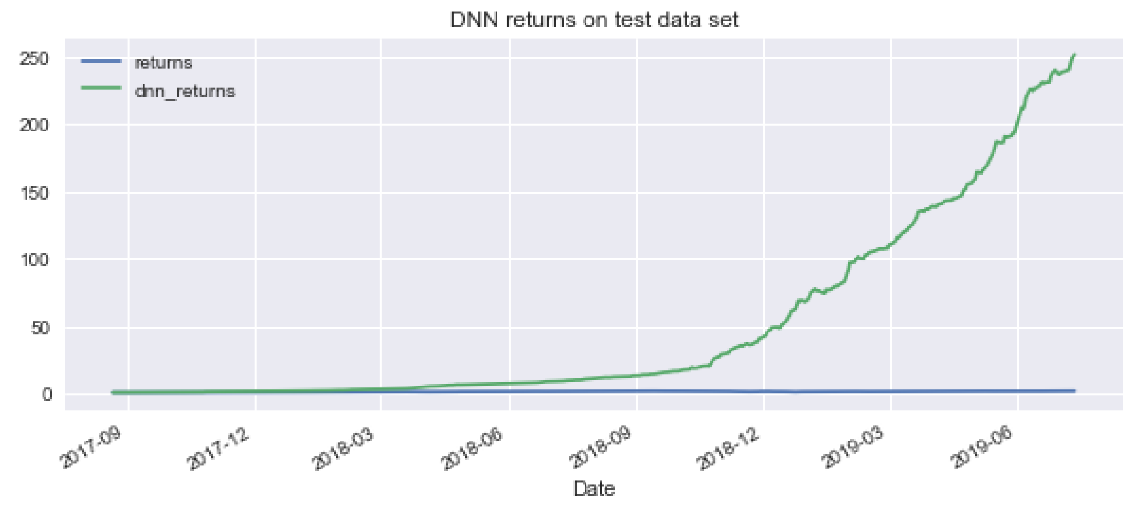
```
: test['dnn_returns'] = test['returns'] * test['dnn_pred']
```

```
: test[['returns', 'dnn_returns']].sum().apply(np.exp)
```

```
: returns          2.109536
  dnn_returns    251.442967
  dtype: float64
```

```
test[['returns', 'dnn_returns']].cumsum(
        ).apply(np.exp).plot(figsize=(10, 4),  title="DNN returns on test data set")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d8c817c1d0>
```



5 Python code

These are the list of Jupyter Notebooks present in this project

1. 1 Logistic Classifier and Bayesian Classifier.ipynb
2. 2 SVM.ipynb
3. 3 Decision-Tree.ipynb
4. 4 KNN Classifier.ipynb
5. 5 ANN Deep Learning.ipynb

5.1 Common Classes and Methods

1. **Feature Enum:** This is an enum class containing the types of Feature supported.
2. **get_data(features, normalize = True):** This method returns the Features DataFrames which can be sent as input to the various machine learning algorithms explored in this project. It take
   Input Arguments:
   features = This is a list of values from Feature type class, it contains the type of features to be extracted.
   normalize = A Boolean variable specifying if the feature values to be normalized (zero mean)
   Return Values:
   Returns the features

6 Conclusion

Predicted the direction of AMZN Stock using the below Feature Engineering techniques and using the below list of models

Feature Engineering used in this project

Feature Engineering
OHLC
LAGGED_RETURNS
SMA_PRICE
SMA_VAR
RSI
MACD
EWMA_PRICE
EWMA_VAR

Models used in this project

Model
Logistic Regression
SVM
KNN
Decision Tree
AdaBoost
Deep Learning (DNN)

7 Software Tools and Libraries

1. Python 3.6
2. Jupyter Notebook
3. Numpy, Pandas
4. Tensor flow

8 References

1. Machine Learning in Finance by Miquel Noguer i Alonso PhD
2. Book-1, Big Data and Machine Learning in Quantitative Investment by Tony Guida
3. Book-2, Pattern Recognition and Machine Learning by Christopher M.Bishop
4. Book-3, Advances in Financial Machine Learning by Marcos Lopez de Prado (Author)
5. Investopedia (https://www.investopedia.com/)