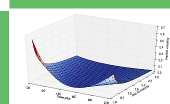
CISC 5352 FINANCIAL PROGRAMMING AND DATA ANALYTICS LECTURE NOTE (5)



Henry Han Ph.D.
Department of Computer and Information Science
Fordham University, New York NY 10023

Last class review

Last class review

1. pandas-datareader package: an update for pandas.io.data
2. Python visualization modules
   1. plot in DataFrame (return object is an ax object)
   2. matplotlib.pylab (pylab) (syntax=matlab's plot)
3. Retrieve option data using Pandas (Yahoo Finance has a temporal unmatched issue)
4. Implied volatility pricing: model based approaches

Q1: Does python has special visualization module for Finance?

Q1: Does python has special visualization module for Finance?

matplotlib.finance

It is still a module in evaluation due to python's fast updates:
In addition to a data retrieval function: **quotes_historical_yahoo_ohlc**, It includes candlestick plot functions:

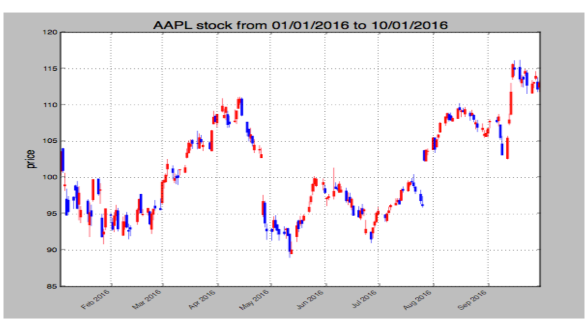
candlestick_ochl(ax, opens, closes, highs, lows, width, colorup, colordown', ticksize, alpha)

A tick/ticksize is the minimum up or down unit in the price of a security.

```
import numpy as np
import matplotlib.pylab   as pylab
from   matplotlib.dates   import *
import matplotlib.finance as finance
import time

# start and end date: (Year, month, day)

start       = (2016, 1,  1)
end         = (2016, 10, 1)
security_symbol ='AAPL'

# Retrieve a security 's historical data from Yahoo finance
quotes      = finance.quotes_historical_yahoo_ohlc(security_symbol, start, end)
quote_siz    = len(quotes)
```

```
if (quote_siz<1):
   print("doule check data size!\n")
   raise SystemExit
else:
   print(security_symbol + " has {:5d}".format(quote_siz) + "  transaction days")
   time.sleep(1)

# quotes is a list
quotes[3:5]

pylab.close("all")  # bookkeeping

fig  = pylab.subplot(1,1,1)

# candlestick plot
finance.candlestick_ohlc(fig,quotes, width=1.2, colorup='r', colordown='b')
pylab.grid('on')
fig.xaxis_date()       # add date
fig.autoscale_view()   # auto scale
```

```
# gca()--> get current axis (graphics object)
# rotate xlabel 45'

pylab.setp(pylab.gca().get_xticklabels(), FontSize=8, rotation=45,
horizontalalignment='right')
pylab.setp(pylab.gca().get_yticklabels(), FontSize=8.5)

pylab.ylabel("price")
pylab.title('AAPL stock from 01/01/2016 to 10/01/2016')
pylab.show()
```



Note: The same plot can be also obtained by using module 'pyplot'

```
In [254]: quotes[3:5]
Out[254]:
[(735970.0,
  97.027853245863767,
  98.45357371731636,
  94.815523799135008,
  94.835185999999993,
  81094400.0),
 (735971.0,
  96.900032403671403,
  97.450654653231268,
  95.140000444031543,
  95.336648999999994,
  70798000.0)]
```

Unix epoch time: is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT),

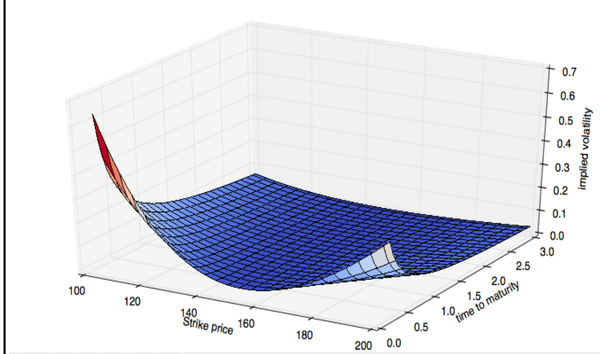
Open High Low Close and Volume

Q2: How about 3D plots?

1. There are quite a few 3D plots from **mpl_toolkits.mplot3d** module
2. We only introduce plot_surface() function.
3. It will be used in your coming implied volatility surface plot!

```
import numpy as np
import matplotlib.pyplot   as plot
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# generate strike prices between 100 and 200

strike_price      = np.linspace(100, 200, 50)
time_to_maturity  = np.linspace(0.25, 3,  50)

# build a coordinate system with 'x' and 'y' variables
strike_price, time_to_maturity =np.meshgrid(strike_price, time_to_maturity)

# generate pseudo-implied volatility by using strike price and time_to_maturity as parameters
implied_vol = ((strike_price -150)**2)/(150*strike_price)/(np.power(time_to_maturity,0.95))
```

```
fig = plot.figure(figsize=(10,5)) # a plot object
ax  = Axes3D(fig)              # create 3D object/handle

# plot surface: array row/column stride (step size):2

surf = ax.plot_surface(strike_price, time_to_maturity, implied_vol,
rstride=2, cstride=2, cmap=cm.coolwarm, linewidth=0.5, antialiased=False)

# set x ,y z labels

ax.set_xlabel('Strike price')
ax.set_ylabel('time to maturity')
ax.set_zlabel('implied volatility')

plot.show()
```



Implied volatility is a forward-looking measure

① It is the spread degree of a stock in the future based on its option price in the market

② It is the value that makes the theoretical price of an option under an option pricing model equal to its current market price.

③ The solution of the following equation:

ModelOptionPrice=OptionMarketPrice

BSModelOptionPrice(S,K,r, T, σ_{imp})=OptionMarketPrice

Implied volatility is a solution to the equation such that the theoretical value equal to market value

f(x) = BSMPrice(S,K,r, T, x) – MarketPrice

$f(\sigma_{imp})=0$

Example for Europen call: the implied volatility σimp is the quantity that solve the equation, where C* is the current market (call) option price

$$C\left(S_t, K, t, T, r, \sigma^{imp}\right) = C^*$$

There are at least three methods to solve this nonlinear equation

- 1. Bisection Method (linear convergence)
- 2. Newtown method (Quasi-Newton method: quadratic convergence)
- 3. Muller-Bisection (superlinear between linear and quadratic convergence)

NOTE: there are a family of root-finding methods

http://mathworld.wolfram.com/Root-FindingAlgorithm.html

The convergence order is NOT similar to big *O* analysis:

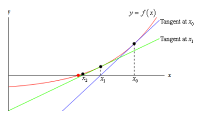
Given the true folution x^* obtained from an iteration algorithm A for $f(x)=0$, x_n is the n^{th} approximation for x^*, the convergence order of the algorithm A is defined as

$$\boxed{q_A = \lim_{n\to\infty} \frac{||x^* - x_{n+1}||}{||x^* - x_n||}}$$

- $q_A = 1$: linear convergence
- $1 < q_A < 2$: Superlinear convergence
- $q_A = 2$: quadratic convergence
- $q_A = k$: k^{th}order convergence

Newton-Raphson (Newton) method

1. It is a root-finding method faster than bisection method with higher convergent rate (convergence order: quadratic convergence: 2)
2. It assumes f(x) is differentiable
3. It starts with a first guess x_0 for the root
4. It moves to find a possible root
   1. $x_1=x_0-f(x_0)/f'(x_0)$
   2. Keep going
   3. $x_n=x_{n-1}-f(x_{n-1})/f'(x_{n-1})$ until $f(x_n)=0$ or $<$tolerance



Pros and cons of Newton method

1. Newton's method may not converge if started too far away from a root (NOT STABLE SOMETIMES!).
2. When it does converge, it is faster than the bisection method, and is usually quadratic.
3. For some functions, it is difficult to calculate the derivative.

Come back our "fancy version": C →f(x)→ can we use newton method? If so, how?

$$C(S_t, K, t, T, r, \sigma^{imp}) = C^*$$

Come back our "fancy version": C →f(x)→ can we use newton method? If so, how?

$$C(S_t, K, t, T, r, \sigma^{imp}) = C^*$$

$x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1})$

$$\sigma_{n+1}^{imp} = \sigma_n^{imp} - \frac{C(\sigma_n^{imp}) - C^*}{\boxed{\partial C(\sigma_n^{imp}) / \partial \sigma_n^{imp}}}$$

The partial derivative of the option pricing formula with respect to the volatility is f'(x) in the Newton method

Can we get the f'(x) from the BSM model?

$$f(x) = C(S, K, T, r, x) - C^*$$

f(x) is differentiable due to the nature of the BSM model

Can we get the f'(x) from the BSM model?

The partial derivative of the option pricing formula with respect to the volatility is f'(x) in the Newton method

It has an official name: vega in the BSM model: the change rate of option price w.r.t. volatility

$$\frac{\partial f}{\partial \sigma} = SN'(d_1)\sqrt{T} = Sn(d_1)\sqrt{T}$$

Note: N'(x) =n(x): the density function of standard normal distribution

$$n(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Compute Vega

Given an option with stock price: $80 and its strike price $100 and time to maturity is ¾ years. Suppose the interest rate is 10.5%. Write a program to compute its Vega

Note: Vega is independent of option put/call type

```
from math import log, sqrt
from scipy import stats

def bsm_vega(S, K, T, r, sigma):

    d1   = (log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))

     vega = S * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(T)

     return vega
```

How about the newton method to compute the implied volatility?

$$\sigma_{n+1}^{imp} = \sigma_n^{imp} - \frac{C(\sigma_n^{imp}) - C^*}{\partial C(\sigma_n^{imp}) / \partial \sigma_n^{imp}}$$

$X_{n+1} = x_n - f(x_n)/f'(x_n)$

Note: we assume we are working for call options

Code sketch for Newton method to compute the implied volatility

```
def bsm_call_imp_vol(S, K, T, r, C_star, sigma_est, iter):
# INPUT
        # S, K, T, r, C_star, iter

# OUTPUT
        # sigma_est: implied volatility

for i in range(iter):

    f       = bsm_call_value(S, K, T, r, sigma_est) – C_star

    f_prime = bsm_vega(S, K, T, r, sigma_est)

    sigma_est =   sigma_est -(f/f_prime)

    return sigma_est
```

Note: You need to give defined information about the parameters in your coding

Now you can use Newton method to predict implied volatility

Newton method is a little bit unpredictable/instable!

It can be very slow or even not to converge if you have a bad initial point!

Can we use a superlinear convergence method?

A superlinear method: Muller method

① Its generalizes the secant method of root finding by using quadratic 3-point interpolation

② It constructs a parabola through three points, and takes the intersection of the *x*-axis with the parabola to be the next approximation.

③ The order of convergence is approximately 1.84.

④ Only locally convergent.

A superlinear method: Muller method

Generalizes the secant method of root finding by using quadratic 3-point interpolation

$$q \equiv \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}.$$

Then define

$$\begin{aligned} A &\equiv q\,P(x_n) - q\,(1+q)\,P(x_{n-1}) + q^2\,P(x_{n-2}) \\ B &\equiv (2\,q+1)\,P(x_n) - (1+q)^2\,P(x_{n-1}) + q^2\,P(x_{n-2}) \\ C &\equiv (1+q)\,P(x_n), \end{aligned}$$

and the next iteration is

$$x_{n+1} = x_n - (x_n - x_{n-1})\,\frac{2\,C}{\max\left(B \pm \sqrt{B^2 - 4\,A\,C}\right)}.$$

Credit to http://mathworld.wolfram.com/MullersMethod.html

Constructs a parabola through three points, and takes the intersection of the x-axis with the parabola to be the next approximation.

a. Set three initial value x_0, x_1, x_2 (x_0 and x_1 and x_2 can determines a quadratic parabola, where x_2 is the intersection of the x-axis with the line through $(x_0, f(x_0))$ and $(x_1, f(x_1))$.

b. Create the parabola which passes through $(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2))$

c. The corresponding quadratic polynomial is

$$P(x) = A(x - x_2)^2 + B(x - x_2) + C$$

It satisfies,

$$\begin{cases} f(x_0) = A(x_0 - x_2)^2 + B(x_0 - x_2) + C \\ f(x_1) = A(x_1 - x_2)^2 + B(x_1 - x_2) + C \\ f(x_2) = C \end{cases}$$

d. So the next approximation x_3 which is closer to the root then x_2 can be compute as the following equation:

$$x_3 = x_2 - \frac{2C}{B + sign(B)\sqrt{B^2 - 4AC}}$$

e. We can now use x_1, x_2, x_3 to calculate the next approximation x_4.

f. Repeat above steps until we reach the given definition.

More detailed Muller method

Muller-Bisection: Improved Muller method and Bisection method

Xinyuan Wu, *Applied Mathematic and Computation* , 2005

It combines the convergent efficiency of Muller's method and global convergence of the Bisection method!

The order convergence is almost 1.84

Muller-Bisection Algorithm

1. Set two initial value a,b, such that f(a) and f(b) have opposite signs.
2. Calculate the midpoint between a and b, c=(a+b)/2.

Muller-Bisection Algorithm Cont'd

3. Use Muller's method to create a quadratic polynomial P(x) based on (a,f(a)),(b,f(b)) and (c,f(c)), then get the next approximation c_2

Muller-Bisection Algorithm Cont'd

4. Create subinterval $[a_2,b_2]$

If $f(a)*f(c)<0$, then $[a,c]$ is the subinterval
If $f(b)*f(c)<0$, then $[c,b]$ is the subinterval

5. Compare c_2 with $[a_2,b_2]$
If c_2 is within $[a_2,b_2]$, then keep c_2
If c_2 is out of $[a_2,b_2]$, then change into $(a_2+b_2)/2$, which is the midpoint of the subinterval

Muller-Bisection Algorithm Cont'd

6. Use Muller's method to get the next approximation c_3 based on a_2, b_2 and c_2.

7. Repeat steps 4~6, until finding a sufficiently accurate solution c_n , such that $f(c_n)=0$ or less than tolerant error ε.

Muller-Bisection Algorithm

a. Set two initial value a, b, such that $f(a)$ and $f(b)$ have opposite signs.

b. Calculate the midpoint between a and b and $c = (a + b)/2$.

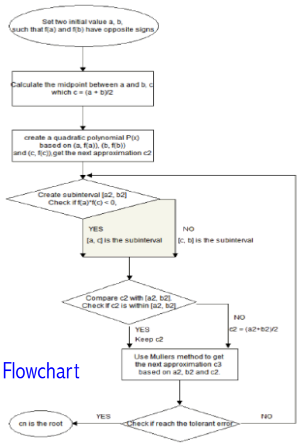
c. Use Mullers method to create a quadratic polynomial $P(x)$ based on $(a, f(a)), (b, f(b))$ and $(c, f(c))$, then get the next approximation c_2.

d. Create subinterval $[a_2, b_2]$. If $f(a)*f(c) < 0$, then $[a, c]$ is the subinterval;if $f(b)*f(c) < 0$, then $[c, b]$ is the subinterval.

e. Compare c_2 with $[a_2, b_2]$.If c_2 is within $[a_2, b_2]$, then keep c_2; if c_2 is out of $[a_2, b_2]$, then change c_2 into $(a_2 + b_2)/2$, which is the midpoint of the subinterval.

f. Use Mullers method to get the next approximation c_3 based on a_2, b_2 and c_2.

Muller-Bisection Algorithm Flowchart



Implied Volatility Problem

Dataset

Source: Yahoo Finance

Google's call and put option prices on 08/20/2013, which will expire on 09/13/2013

Implied Volatility Problem

Relevant inputs:

- Google's stock price on 08/20/2013 is 865.42
- Market option prices and corresponding strike prices
  - (323 observations for call option, 337 observations for put option, from Yahoo Finance)
- Expiration time is 24/365=0.0658
- Risk-free rate is 0.02
  - (3-month T-bill rate, from US Department of Treasury)

Implied Volatility Calculation

① We use Black-Scholes model to calculate theoretical option prices.

② Implied volatility is the volatility that makes theoretical prices equal to market prices.

③ For each market option price and corresponding strike price, we calculate one implied volatility

Compare the running time of both Bisection method and Muller-Bisection method (Matlab),

			Average Iteration Number		Running Time (seconds)	
	Observations	Tolerant Error	Bisection	Muller-Bisection	Bisection	Muller-Bisection
Sep 13 Call	323	10e-5	67.4364	63.7492	7.533	5.152
Sep 13 Put	337	10e-5	41.1513	19.6706	8.306	5.595

Another implied volatility case

Data : all call/put options for AAPL (Apple stock), Yahoo, Microsoft, ORACLE, and Intel. Choose the expiration time by using Sept 2015,Oct 2015 and Jan 2016 options.

Methods: Bisection, Newton, Muller-bisection (Muller)

		Implied volatility			Running time		
	Tolerant error	Bisection	Muller-bisection	Newton	Bisection	Muller-bisection	Newton
Oct 26 Call	10^{-5}	0.82619307	0.82619336	0.82619540	0.0118	0.0031	0.0019
Oct 26 Put	10^{-5}	0.44573414	0.44571323	0.44570652	0.0159	0.0101	0.0049

Other methods to use 2nd order derivatives: Halley's Irrational Formula

A root-finding algorithm which makes use of a third-order Taylor series

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2} f''(x_n)(x - x_n)^2 + \ldots.$$

A root of $f(x)$ satisfies $f(x) = 0$, so

$$0 \approx f(x_n) + f'(x_n)(x_{n+1} - x_n) + \frac{1}{2} f''(x_n)(x_{n+1} - x_n)^2.$$

Using the quadratic equation then gives

$$x_{n+1} = x_n + \frac{-f'(x_n) \pm \sqrt{[f'(x_n)]^2 - 2 f(x_n) f''(x_n)}}{f''(x_n)}.$$

Picking the plus sign gives the iteration function

$$C_f(x) = x - \frac{1 - \sqrt{1 - \frac{2 f(x) f''(x)}{[f'(x)]^2}}}{\frac{f''(x)}{f'(x)}}.$$

Credit to http://mathworld.wolfram.com/

We will compare these methods in implied volatility computing

Halley's Irrational Formula needs another greek: Vomma (volga): $Vomma = Vega(\frac{d_1 d_2}{\sigma})$

Python modules: Optimization and root finding (scipy.optimize)

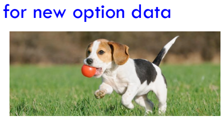
http://docs.scipy.org/doc/scipy-0.14.0/reference/optimize.html

Data-driven approach Using machine learning models to predict implied volatility

Idea: learn knowledge from known data, then make prediction for new data!

Training: using known option data with implied volatility to train a statistical learning model (e.g. NN), the model will learn knowledge from training.

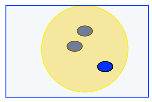
Test (Prediction): used the trained model to predict implied volatility for new option data



Machine learning Methods

① K-NN (k-nearest neighbors)
② NN (neural network)
③ Support vector Machines (SVM)
④ Random Forest Trees (RF)
⑤ Gradient boosting (GB)
⑥ Discrimination Analysis
⑦ Bayesian classification

Introduction to the K-Nearest Neighbor (KNN) Classification



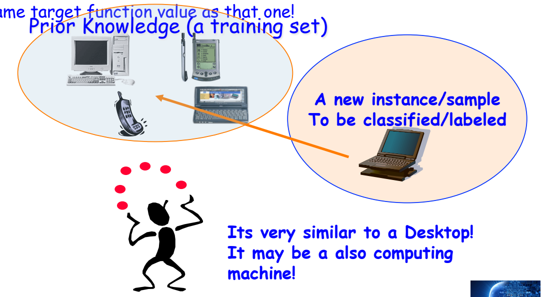
What ‘s the K-Nearest neighbor classification?

① A supervised learning algorithm for qualitative variables
② More exactly, it is an instance based learning algorithm
③ Instance based learning is also called lazy learning
④ Why?

Instance based learning: It ‘s lazy!

uses specific training instances (samples) to predict labels for output variables instead of building a mathematical model to conduct classification!

Idea of the kNN： which sample in the training set is 'mostly similar' to me, then I will take its label: i will have the same target function value as that one!



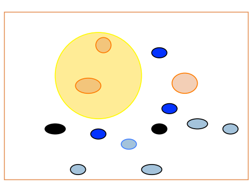
More information about the KNN

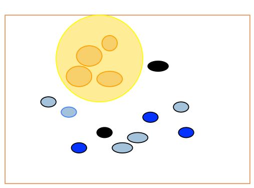
① An instance/sample = a point in R^n space
② A training set: a set of instances with labels (target function values)

$$\begin{pmatrix} x_1 - y_1 \\ x_2 - y_2 \\ \vdots \\ x_n - y_n \end{pmatrix}$$

③ The labels takes discrete values. for example {0,1}
④ For a new instance, classification is done by finding k-nearest instances in the training set.
⑤ The final label will be decided by "majority voting"

1-NN and 3-NN: K-NN is just the extension of 1-NN





How to find the nearest neighbors for a new sample to be classified ?

Which distances measure should be used?

- Euclidean distance (mostly used)

$$d(x,y) = (\sum_{i=1}^{n}(x_i - y_i)^2)^{1/2}$$

- Correlation distance (mostly used in gene/protein pattern classification)

$$corr\_dist(x,y) = 1 - corr(x,y)$$

- Cosine distance:

$$\cos\_dist(x,y) = 1 - \cos(x \wedge y)$$

How to decide the label for a new instance?

Majority voting:

the label of the new instance is decided the label of the samples which have the largest votes in the neighborhood

How to break the tie-vote?

① Random: pick any label of the equal groups of samples in the tie-voting

② Nearest: pick the label of the equal group which has the nearest distance to the new instance

③ Other methods…

What are the possible weak points of the majority voting?

It treats all k nearest neighbors uniformly. No preference given to any nearest neighbors.

Every neighbor has the same impact on the classification!

A little bit optimization: Weighted voting

- Idea: give more weights to the samples with nearer distances to the new instance!
- For a new instance z=(x',y'), the majority voting can be expressed as

Label y' will be the label which has the maximum votes among k-nearest neighbors of z

$$y' = \arg\max_{v} \sum_{(x_i, y_i) \in N(z,k)} I(v = y_i)$$

V: class label
I(v=y_i)=1 if the label is y_i, otherwise I(v=y_i)=0

Weighted voting…

Label y' will be the label which has the maximum weights among k-nearest neighbors of z

$$y' = \arg\max_{v} \sum_{(x_i, y_i) \in N(z,k)} w_i I(v = y_i)$$

How to assign weights for each neighbor?

Basic idea: the samples closer to the new instance will have more influences on the classification than the samples far to the new instance

$$w_i = 1/dist(x', x_i)$$

Advantages of KNN

① The idea is very intuitive! It is similar to winners-take-all competitive learning.

② A simple Bayesian classification method.

③ It can achieve good classification results as some complicate classification algorithms without building a mathematical model!

④ Target function of the whole dataset can be represented as a combination of less complex local approximations implicitly.

Disadvantages of kNN

① kNN classification is based on local information: labels of k nearest neighbors. The knn classification is susceptible to noise.

② Low learning speed and curse of dimensions

③ Need to record the relationships between testing sample and all training data→large storage requirements

④ A large K will lead to the loss of the locality

Speed-up version kNN: kd-tree-NN

① K-d tree based speedup.

② K-d tree (k-dimensional tree) is a hierarchical data structure to organized data points in R^k space. It is like octree.

③ With help of k-d tree, the samples in the training samples are organized in a sorted way, which makes the nearest neighbors search fast: from the linear complexity to log complexity.