Estimating Financial Risk through Monte Carlo Simulation
2017 Edition

26/04/2017

Authors Ole Andreas Hansen | Alberto Ibarrondo Luis

Risk analysis is part of every decision we make when faced with uncertainty, ambiguity, and variability. Indeed, even though we have unprecedented access to information, we can't accurately predict the future. In finance, there is a fair amount of uncertainty and risk involved with estimating the future value of financial products, due to the wide variety of potential outcomes. Monte Carlo simulation (also known as the Monte Carlo Method) allows inspecting many possible outcomes of the decision making process, and can be used to assess the impact of risk: this, in turns, allows for better decision-making under uncertainty.

Goals

The main objectives we set for this Notebook are as follows:

1. Develop fundamental knowledge about Risk analysis
2. Understand Monte Carlo Simulation (MCS)
3. Apply Monte Carlo Simulation for predicting risk

Steps

1. First, in section 1, we introduce the basics of MCS
2. In section 2, we work on a simple example to where we apply the MCS method
3. In section 3, we briefly summarize the main characteristics of the Monte Carlo Simulation (MCS) technique
4. In section 4, we overview the common distributions which are often used in MCS
5. In section 5, we work on a real use case, that focuses on estimating financial risk. We will use techniques such as featurization (that is, generating additional features to improve model accuracy), linear regression, kernel density estimation, sampling distributions and so on ...

Reference

This Notebook is inspired by Chapter 9 of the book Advanced Analytics with Spark by Josh Wills, Sandy Ryza, Sean Owen, and Uri Laserson. It is strongly suggested to read this Chapter to get a general idea of the topic of this Notebook.

1. Introduction

1.1. Monte Carlo Simulation (MCS)

Monte Carlo simulation is a computerized mathematical technique that can be applied such that it is possible to account for risk in quantitative analysis and decision making. This technique is used in many different fields, such as R&D, risk management, portfolio management, pricing derivatives, strategic planning, project planning, cost modeling and many more.

In general, MCS is a technique that "converts" uncertainty on input variables of a model into **probability distributions**. By combining the distributions and randomly selecting values from them, it recalculates the simulated model many times, to determine the probability of the output.

Historically, this technique was first used by scientists working on the atomic bomb: it was named after Monte Carlo, the Monaco resort town renowned for its casinos. Since its introduction in World War II, Monte Carlo simulation has been used to model a variety of physical and conceptual systems.

1.2. How does it work?

Monte Carlo simulation performs risk analysis by building models of possible results by *substituting a range of possible input values, that constitute uncertainty, into a statistical distribution*. It then computes possible outcomes repeatedly, each time using a different set of random values from the probability functions that "model" the input. Depending upon the number of random input variables and their distribution, a Monte Carlo simulation could involve thousands or tens of thousands of "rounds" before it is complete. When complete, *Monte Carlo simulation produces distributions of possible outcome values*.

By using probability distributions instead of actual input samples, it is possible to model more accurately uncertainty: different choices of distributions will yield different outputs.

2. Illustrative example

Imagine you are the marketing manager for a firm that is planning to introduce a new product. You need to estimate the first-year net profit from this product, which might depend on:

- Sales volume in units
- Price per unit (also called "Selling price")
- Unit cost
- Fixed costs

Net profit will be calculated as $Net Profit = Sales Volume* (Selling Price - Unit cost) - Fixed costs$. Fixed costs (accounting for various overheads, advertising budget, etc.) are known to be \$ 120,000, which we assume to be deterministic. All other factors, instead, involve some uncertainty: *sales volume* (in units) can cover quite a large range, the *selling price* per unit will depend on competitor actions, which are hard to predict, and *unit costs* will also vary depending on vendor prices and production experience, for example.

Now, to build a risk analysis model, we must first identify the uncertain variables -- which are essentially random variables. While there's some uncertainty in almost all variables in a business model, we want to focus on variables where the range of values is significant.

2.1. Unit sales and unit price

Based on a hypothetical market research you have done, you have beliefs that there are equal chances for the market to be `slow`, `normal`, or `hot`:

- In a "slow" market, you expect to sell 50,000 units at an average selling price of \$11.00 per unit
- In a "normal" market, you expect to sell 75,000 units, but you'll likely realize a lower average selling price of \$10.00 per unit
- In a "hot" market, you expect to sell 100,000 units, but this will bring in competitors, who will drive down the average selling price to \$8.00 per unit

Question 1

Calculate the average units and the unit price that you expect to sell, which depend on the market state. Use the assumptions above to compute the expected quantity of products and their expected unit price.

In [432]:

```
prices = [11, 10, 8]
units = [50000, 75000, 100000]

average_unit = round(sum(units)/3, 2)
average_price = round(sum(prices)/3, 8)

print("average unit:", average_unit)
print("average_price:", average_price)

#Clearly the second option is the closest one to the average (75k units
and 10€ per unit)
```

```
average unit: 75000.0
average_price: 9.66666667
```

2.2. Unit Cost

Another uncertain variable is Unit Cost. In our illustrative example, we assume that your firm's production manager advises you that unit costs may be anywhere from \$5.50 to \$7.50, with a most likely expected cost of \$6.50. In this case, the most likely cost can be considered as the average cost.

2.3. A Flawed Model: using averages to represent our random variables

Our next step is to identify uncertain functions -- also called functions of a random variable. Recall that Net Profit is calculated as $Net Profit = Sales Volume * (Selling Price - Unit cost) - Fixed costs$. However, Sales Volume, Selling Price and Unit Cost are all uncertain variables, so Net Profit is an uncertain function.

The simplest model to predict the Net Profit is using average of sales volume, average of selling price and average of unit cost for calculating. So, if only consider averages, we can say that the $Net Profit = 75,000*(9.66666666 - 6.5) - 120,000 \sim 117,500$.

However, as Dr. Sam Savage warns, "Plans based on average assumptions will be wrong on average." The calculated result is far from the actual value: indeed, the **true average Net Profit** is roughly \$93,000, as we will see later in the example.

Question 2

Question 2.1

Write a function named `calNetProfit` to calculate the Net Profit using the average of sales volume, the average of selling price and the average of unit cost.

In [433]:

```
def calNetProfit(average_unit, average_price, average_unitcost, fixed_cost):
    return average_unit * (average_price - average_unitcost) - fixed_cost

average_unitcost = 6.5
fixed_cost = 120000
NetProfit = calNetProfit(average_unit, average_price, average_unitcost, fixed_cost)
print("Net profit:", NetProfit)
```

```
Net profit: 117500.00024999998
```

Question 2.2

Verify the warning message of Dr. Sam Savage by calculating the error of our estimated Net Profit using averages only. Recall that the true value is roughly \$93,000, so we are interested in:
$$ error = \frac{your\_value - true\_value}{true\_value}$$

Note also we are interested in displaying the error as a percentage.

Looking at the error we make, do you think that we can use the current model that only relies on averages?

In [434]:

```
trueNetProfit = 93000
error = (NetProfit - trueNetProfit) / trueNetProfit
print("Error: %.2f %%"% (error*100))
```

```
Error: 26.34 %
```

COMMENT
We see that our naive model gives an error of 26%. A deviation of 25% in the estimated profit over a year can definitely kill a company. This situation calls for a new upgraded model.

2.4. Using the Monte Carlo Simulation method to improve our model

As discussed before, the selling price and selling volume both depend on the state of the market scenario (slow/normal/hot). So, the net profit is the result of two random variables: `market scenario` (which in turn determines `sales volumes` and `selling price`) and `unit cost`.

Now, let's assume (this is an *a-priori* assumption we make) that `market scenario` follows a discrete, uniform distribution and that `unit cost` also follows a uniform distribution. Then, we can compute directly the values for selling price and selling volumes based on the outcome of the random variable `market scenario`, as shown in Section 2.1.

From these a-priori distributions, in each run (or trial) of our Monte Carlo simulation, we can generate the sample value for each random variable and use it to calculate the Net Profit. The more simulation runs, the more accurate our results will be. For example, if we run the simulation 100,000 times, the average net profit will amount to roughly \$92,600. Every time we run the simulation, a different prediction will be output: the average of such predictions will consistently be less than \$117,500, which we predicted using averages only.

Note also that in this simple example, we generate values for the `market scenario` and `unit cost` independently: we consider them to be **independent random variables**. This means that the eventual (and realistic!) correlation between the `market scenario` and `unit cost` variables is ignored. Later, we will learn how to be more precise and account for dependency between random variables.

Question 3

Question 3.1

Write a function named `get_sales_volume_price` that returns the sales volume and price based on the market scenario. In particular, the scenario can get one of three values:

- 0: Slow market
- 1: Normal market
- 2: Hot market

The return value is a tuple in the form: `(sales_volume, price)`

In [435]:

```
# Get sales volume and  price based on market scenario
# the function returns a tuple of (sales_volume, price)
def get_sales_volume_price(scenario):
    # Slow market
    if scenario == 0:
        return (50000, 11)
    # Normal market
    if scenario == 1:
        return (75000, 10)
    # Hot market
    if scenario == 2:
        return(100000, 8)
```

Question 3.2

Run 100,000 Monte Carlo simulations and calculate the average net profit they produce. Then, compare the result to the "average model" we used in the previous questions (the one we called "flawed" model). Put your comments about the discrepancies between a simplistic model, and the more accurate MCS approach.
Note that in each iteration, the `unit_cost` and `market_scenario` are generated according to their distributions. Also, recall what we have seen in Section 2.2: your firm account manager helped you with some research, to determine the variability of your random variables.

HINT

Function `uniform(a,b)` in module `random` generates a number $a<=c<=b$, which is drawn from a uniform distribution.

Function `randint(a,b)` helps you generating an integer number $a<=c<=b$

In [436]:

```
import random

total = 0.0
num_simulation = 1000000
for i in range(0,num_simulation):
    unit_cost = random.uniform(5.50, 7.50)
    market_scenario = random.randint(0, 2)
    sales_volume, price = get_sales_volume_price(market_scenario)
    netProfit = calNetProfit(sales_volume, price, unit_cost, fixed_cost)
    total += netProfit

print("average net profit:", total/num_simulation)
print("Error: %.2f %% "%((total/num_simulation - trueNetProfit) / trueNe
tProfit*100))
```

```
average net profit: 92445.65606423779
Error: -0.60 %
```

COMMENT

We have jumped from 24% error to 0.5%! This is clearly manageable by the company as a deviation from the true results without setting the firm into bankrupcy. Nevertheless, there is still room for improvement...

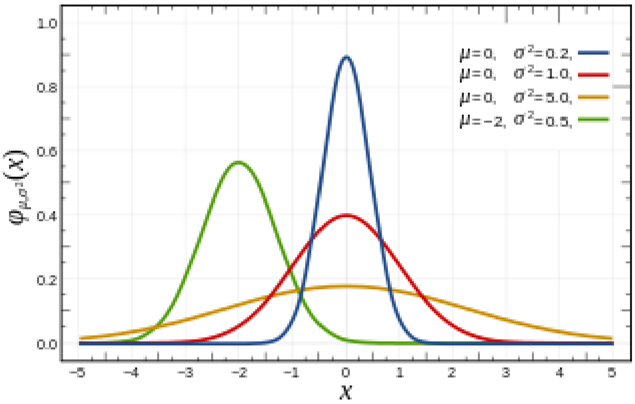
3. A brief summary of the Monte Carlo Simulation (MCS) technique

- A MCS allows several inputs to be used at the same time to compute the probability distribution of one or more outputs
- Different types of probability distributions can be assigned to the inputs of the model, depending on any *a-priori* information that is available. When the distribution is completely unknown, a common technique is to use a distribution computed by finding the best fit to the data you have
- The MCS method is also called a **stochastic method** because it uses random variables. Note also that the general assumption is for input random variables to be independent from each other. When this is not the case, there are techniques to account for correlation between random variables.
- A MCS generates the output as a range instead of a fixed value and shows how likely the output value is to occur in that range. In other words, the model outputs a probability distribution.

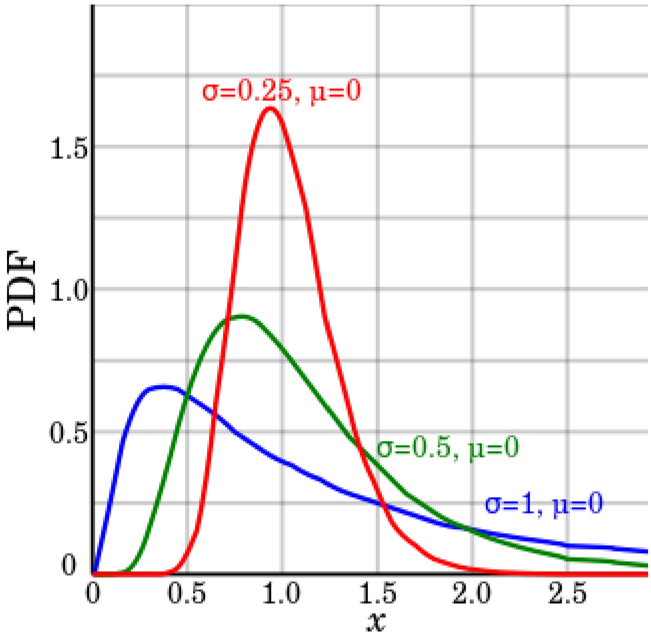
4. Common distributions used in MCS

In what follows, we summarize the most common probability distributions that are used as *a-priori* distributions for input random variables:

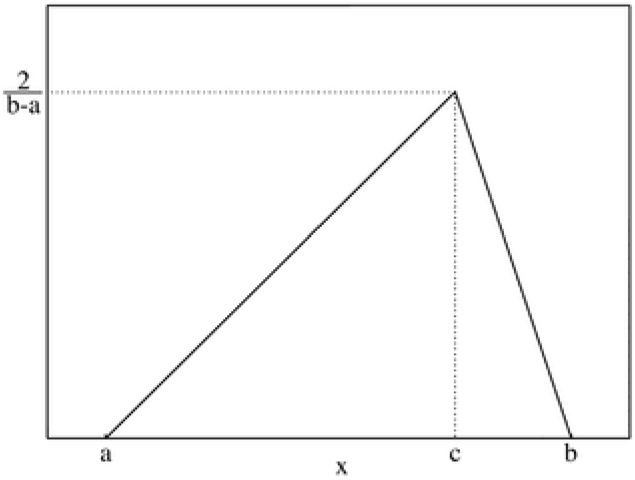
- *Normal/Gaussian Distribution*: this is a continuous distribution applied in situations where the mean and the standard deviation of a given input variable are given, and the mean represents the most probable value of the variable. In other words, values "near" the mean are most likely to occur. This is symmetric distribution, and it is not bounded in its co-domain. It is very often used to describe natural phenomena, such as people’s heights, inflation rates, energy prices, and so on and so forth. An illustration of a normal distribution is given below:



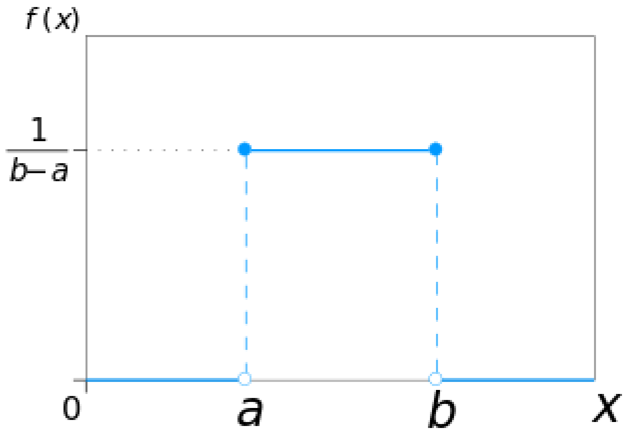
- *Lognormal Distribution*: this is a distribution which is appropriate for variables taking values in the range $[0, \infty]$. Values are positively skewed, not symmetric like a normal distribution. Examples of variables described by some lognormal distributions include, for example, real estate property values, stock prices, and oil reserves. An illustration of a lognormal distribution is given below:



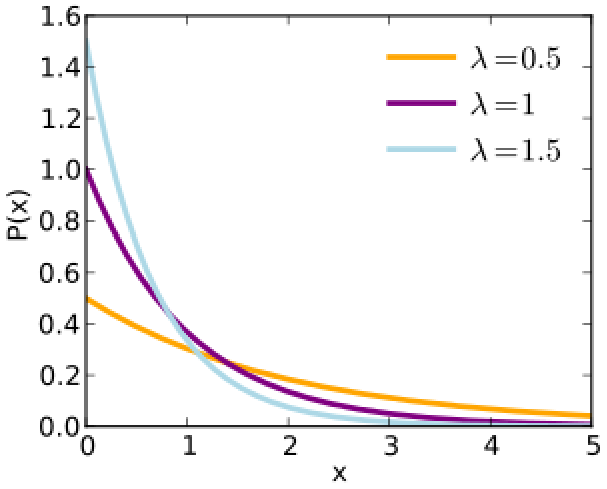
- *Triangular Distribution*: this is a continuous distribution with fixed minimum and maximum values. It is bounded by the minimum and maximum values and can be either symmetrical (the most probable value = mean = median) or asymmetrical. Values around the most likely value (e.g. the mean) are more likely to occur. Variables that could be described by a triangular distribution include, for example, past sales history per unit of time and inventory levels. An illustration of a triangular distribution is given below:



- *Uniform Distribution*: this is a continuous distribution bounded by known minimum and maximum values. In contrast to the triangular distribution, the likelihood of occurrence of the values between the minimum and maximum is the same. In other words, all values have an equal chance of occurring, and the distribution is simply characterized by the minimum and maximum values. Examples of variables that can be described by a uniform distribution include manufacturing costs or future sales revenues for a new product. An illustration of the uniform distribution is given below:



- *Exponential Distribution*: this is a continuous distribution used to model the time that pass between independent occurrences, provided that the rate of occurrences is known. An example of the exponential distribution is given below:



- *Discrete Distribution* : for this kind of distribution, the "user" defines specific values that may occur and the likelihood of each of them. An example might be the results of a lawsuit: 20% chance of positive verdict, 30% change of negative verdict, 40% chance of settlement, and 10% chance of mistrial.

5. A real use case: estimating the financial risk of a portfolio of stocks

We hope that by now you have a good understanding about Monte Carlo simulation. Next, we apply this method to a real use case: *financial risk estimation*.

Imagine that you are an investor on the stock market. You plan to buy some stocks and you want to estimate the maximum loss you could incur after two weeks of investing. This is the quantity that the financial statistic "Value at Risk" (VaR) seeks to measure. VaR is defined as a measure of investment risk that can be used as a reasonable estimate of the maximum probable loss for a value of an investment portfolio, over a particular time period. A VaR statistic depends on three parameters: a portfolio, a time period, and a confidence level. A VaR of 1 million dollars with a 95% confidence level over two weeks, indicates the belief that the portfolio stands only a 5% chance of losing more than 1 million dollars over two weeks. VaR has seen widespread use across financial services organizations. This statistic plays a vital role in determining how much cash investors must hold to meet the credit ratings that they seek. In addition, it is also used to understand the risk characteristics of large portfolios: it is a good idea to compute the VaR before executing trades, such that it can help take informed decisions about investments.

Our goal is calculating VaR of two weeks interval with 95% confidence level and the associated VaR confidence interval.

5.1. Terminology

In this use case, we will use some terms that might require a proper definition, given the domain. This is what we call the *Domain Knowledge*.

- **Instrument**: A tradable asset, such as a bond, loan, option, or stock investment. At any particular time, an instrument is considered to have a value, which is the price for which it can be sold. In the use case of this notebook, instruments are stock investments.
- **Portfolio**: A collection of instruments owned by a financial institution.
- **Return**: The change in an instrument or portfolio's value over a time period.
- **Loss**: A negative return.
- **Index**: An imaginary portfolio of instruments. For example, the NASDAQ Composite index includes about 3,000 stocks and similar instruments for major US and international companies.
- **Market factor**: A value that can be used as an indicator of macro aspects of the financial climate at a particular time. For example, the value of an index, the Gross Domestic Product of the United States, or the exchange rate between the dollar and the euro. We will often refer to market factors as just factors.

5.2. The context of our use case

We have a list of instruments that we plan to invest in. The historical data of each instrument has been collected for you. For simplicity, assume that the returns of instruments at a given time, depend on 4 market factors only:

- GSPC value
- IXIC value
- The return of crude oil
- The return of treasury bonds

Our goal is building a model to predict the loss after two weeks' time interval with confidence level set to 95%.

As a side note, it is important to realize that the approach presented in this Notebook is a simplified version of what would happen in a real Financial firm. For example, the returns of instruments at a given time often depend on more than 4 market factors only! Moreover, the choice of what constitute an appropriate market factor is an art!

5.3. The Data

The stock data can be downloaded (or scraped) from Yahoo! by making a series of REST calls. The data includes multiple files. Each file contains the historical information of each instrument that we want to invest in. The data is in the following format (with some samples):

```
Date, Open, High, Low, Close, Volume, Adj Close
2016-01-22,66.239998,68.07,65.449997,67.860001,137400,67.860001
2016-01-21,65.410004,66.18,64.459999,65.050003,148000,65.050003
2016-01-20,64.279999,66.32,62.77,65.389999,141300,65.389999
```

```
2016-01-19,67.720001,67.989998,64.720001,65.379997,178400,65.3799
97
```

The data of GSPC and IXIC values (our two first market factors) are also available on Yahoo! and use the very same format.

The crude oil and treasure bonds data is collected from investing.com, and has a different format, as shown below (with some samples):

```
Date    Price   Open    High    Low     Vol.    Change %
Jan 25, 2016    32.17   32.36   32.44   32.10   -       -0.59%
Jan 24, 2016    32.37   32.10   32.62   31.99   -       0.54%
Jan 22, 2016    32.19   29.84   32.35   29.53   -       9.01%
Jan 21, 2016    29.53   28.35   30.25   27.87   694.04K 11.22%
Jan 20, 2016    26.55   28.33   28.58   26.19   32.11K  -6.71%
Jan 19, 2016    28.46   29.20   30.21   28.21   188.03K -5.21%
```

In our use case, the factors' data will be used jointly to build a statistical model: as a consequence, we first need to preprocess the data to proceed.

5.4. Data preprocessing

In this Notebook, all data files have been downloaded for you, such that you can focus on pre-processing. Next, we will:

- Read the factor data files which are in two different formats, process and merge them together
- Read the stock data and pre-process it
- Trim all data into a specific time region
- Fill in the missing values
- Generate the data of returns in each two weeks' time interval window

Factor data pre-processing

We need two functions to read and parse data from Yahoo! and Investing.com respectively. We are interested only in information about the time and the corresponding returns of a factor or an instrument: as a consequence, we will project away many columns of our RAW data, and keep only the information we are interested in.

The 3000-instrument and the 4-factor history are small enough to be read and processed locally: we do not need to use the power of parallel computing to proceed. Note that this is true also for larger cases with hundreds of thousands of instruments and thousands of factors. The need for a distributed system like Spark comes in when actually **running** the Monte Carlo simulations, which can require massive amounts of computation on each instrument.

Question 4

Question 4.1

Write a function named `readInvestingDotComHistory` to parse data from investing.com based on the format specified above (see Section 5.3). Recall that we use two factors here: one that is related to the price of crude oil, one that is related to some specific US bonds. Print the first 5 entries of the first factor (crude oil price) in the parsed data.

Note that we are only interested in the date and price of stocks.

HINT

You can parse a string to `datetime` object by using the function `strptime(, )`. In this case, the datetime format is `"%b %d, %Y"`. For more information, please follow this link.

In the next cell, we simply copy data from our HDFS cluster (that contains everything we need for this Notebook) to the instance (a Docker container) running your Notebook. This means that you will have "local" data that you can process without using Spark. Note the folder location: find and verify that you have correctly downloaded the files!

In [437]:

```
! [ -d monte-carlo-risk ] || (echo "Downloading prepared data from HDFS. Please wait..." ; hdfs dfs -copyToLocal /datasets/monte-carlo-risk . ; echo "Done!";)
```

In [438]:

```
from datetime import datetime
from datetime import timedelta
from itertools import islice
%matplotlib inline
import numpy as np
import statsmodels.api as sm

base_folder = "monte-carlo-risk/"

factors_folder= base_folder + "factors/"

# read data from local disk
def readInvestingDotComHistory(fname):
    def process_line(line):
        cols = line.split('\t')
        date = datetime.strptime(cols[0], "%b %d, %Y")
        value = float(cols[2]) # open value
        return (date, value)

    with open(fname) as f:
        content_w_header = f.readlines()
        # remove the first line
        # and reverse lines to sort the data by date, in ascending order
        content = content_w_header[1:]
        return sorted(list(map(process_line , content)), key=lambda x: x[0])

factor1_files = ['crudeoil.tsv', 'us30yeartreasurybonds.tsv']
factor1_files = map(lambda fn: factors_folder + fn, factor1_files)
factors1 = [readInvestingDotComHistory(f) for f in factor1_files]


print('First factor first values:', factors1[0][:5])
print('First factor last values:', factors1[0][-5:])

print('Second factor first values:', factors1[1][:5])
print('Second factor last values:', factors1[1][-5:])
```

```
First factor first values: [(datetime.datetime(2006, 1, 26, 0, 0), 65.85
), (datetime.datetime(2006, 1, 27, 0, 0), 66.49), (datetime.datetime(200
6, 1, 30, 0, 0), 67.85), (datetime.datetime(2006, 1, 31, 0, 0), 68.4), (
datetime.datetime(2006, 2, 1, 0, 0), 67.8)]
First factor last values: [(datetime.datetime(2016, 1, 20, 0, 0), 28.33)
, (datetime.datetime(2016, 1, 21, 0, 0), 28.35), (datetime.datetime(2016
, 1, 22, 0, 0), 29.84), (datetime.datetime(2016, 1, 24, 0, 0), 32.1), (d
atetime.datetime(2016, 1, 25, 0, 0), 32.36)]
Second factor first values: [(datetime.datetime(2008, 2, 12, 0, 0), 4.40
1), (datetime.datetime(2008, 2, 13, 0, 0), 4.46), (datetime.datetime(200
8, 2, 14, 0, 0), 4.533), (datetime.datetime(2008, 2, 15, 0, 0), 4.626),
(datetime.datetime(2008, 2, 19, 0, 0), 4.583)]
Second factor last values: [(datetime.datetime(2016, 1, 20, 0, 0), 2.825
), (datetime.datetime(2016, 1, 21, 0, 0), 2.766), (datetime.datetime(201
6, 1, 22, 0, 0), 2.81), (datetime.datetime(2016, 1, 24, 0, 0), 2.828), (
datetime.datetime(2016, 1, 25, 0, 0), 2.825)]
```

Now, the data structure `factors1` is a list, containing data that pertains to two (out of a total of four) factors that influence the market, as obtained by investing.com. Each element in the list is a tuple, containing some sort of timestamp, and the value of one of the two factors discussed above. From now on, we call these elements "**records**" or "**entries**". Visually, `factors1` looks like this:

0 (crude oil)	**1 (US bonds)**
time_stamp, value	time_stamp, value
...	...
time_stamp, value	time_stamp, value
...	...

Question 4.2

Write a function named `readYahooHistory` to parse data from yahoo.com based on its format, as described in Section 5.3.
Print the first 5 entries of the first factor (namely GSPC). Comment the time range of the second batch of data we use in our Notebook.

Note that we are only interested in the date and price of stocks.

NOTE The datetime format now is in a different format than the previous one.

HINT Use a terminal (or put the bash commands inline in your Notebook) to list filenames in your local working directory to find and have a look at your local files.

In [439]:

```
# read data from local disk
def readYahooHistory(fname):
    def process_line(line):
        cols = line.split(',')
        date = datetime.strptime(cols[0], '%Y-%m-%d')
        value = float(cols[1]) # open value
        return (date, value)

    with open(fname) as f:
        content_w_header = f.readlines()
        # remove the first line
        # and reverse lines to sort the data by date, in ascending order
        content = content_w_header[1:]
        return sorted(list(map(process_line , content)), key=lambda x: x
[0])
#'COPPER.csv', 'LEAD.csv', 'IRON.csv', , 'USD-EUR.csv'

factor2_files = ['GSPC.csv', 'IXIC.csv']
factor2_files = map(lambda fn: factors_folder + fn, factor2_files)

factors2 = [readYahooHistory(f) for f in factor2_files]

print(factors2[0][:5])
print(factors2[0][-5:])

print(factors2[1][:5])
print(factors2[1][-5:])
```

```
[(datetime.datetime(1950, 1, 3, 0, 0), 16.66), (datetime.datetime(1950,
1, 4, 0, 0), 16.85), (datetime.datetime(1950, 1, 5, 0, 0), 16.93), (date
time.datetime(1950, 1, 6, 0, 0), 16.98), (datetime.datetime(1950, 1, 9,
0, 0), 17.08)]
[(datetime.datetime(2016, 1, 15, 0, 0), 1916.680054), (datetime.datetime
(2016, 1, 19, 0, 0), 1888.660034), (datetime.datetime(2016, 1, 20, 0, 0)
, 1876.180054), (datetime.datetime(2016, 1, 21, 0, 0), 1861.459961), (da
tetime.datetime(2016, 1, 22, 0, 0), 1877.400024)]
[(datetime.datetime(1971, 2, 5, 0, 0), 100.0), (datetime.datetime(1971,
2, 8, 0, 0), 100.839996), (datetime.datetime(1971, 2, 9, 0, 0), 100.7600
02), (datetime.datetime(1971, 2, 10, 0, 0), 100.690002), (datetime.datet
ime(1971, 2, 11, 0, 0), 101.449997)]
[(datetime.datetime(2016, 1, 15, 0, 0), 4464.370117), (datetime.datetime
(2016, 1, 19, 0, 0), 4548.049805), (datetime.datetime(2016, 1, 20, 0, 0)
, 4405.220215), (datetime.datetime(2016, 1, 21, 0, 0), 4480.700195), (da
tetime.datetime(2016, 1, 22, 0, 0), 4557.390137)]
```

COMMENT:
The factors *GSPC.csv* and *IXIC.csv* has a history of 66 and 45 years respectively. The two other factors has much less history, with 10 and 8 years. Since we need a complete history for our estimations we need to limit the data to the factor with the shortest history, ie 8 years.

Now, the data structure `factors2` is again list, containing data that pertains to the next two (out of a total of four) factors that influence the market, as obtained by Yahoo!. Each element in the list is a tuple, containing some sort of timestamp, and the value of one of the two factors discussed above. Visually, `factors2` looks like this:

0 (GSPC)	1 (IXIC)
time_stamp, value	time_stamp, value
...	...

time_stamp, value	time_stamp, value
...	...

Stock data pre-processing

Next, we prepare the data for the instruments we consider in this Notebook (i.e., the stocks we want to invest in).

Question 4.3

> In this Notebook, we assume that we want to invest on the first 35 stocks out of the total 3000 stocks present in our datasets.
> Load and prepare all the data for the considered instruments (the first 35 stocks) which have historical information for more than 5 years. This means that all instruments with less than 5 years of history should be removed.

HINT we suggest to open a terminal window (not on your local machine, but the Notebook terminal that you can find on the Jupyter dashboard) and visually check the contents of the directories holding our dataset, if you didn't do this before! Have a look at how stock data is organized!

In [440]:

```
from os import listdir
from os.path import isfile, join

stock_folder = base_folder + 'stocks'
num_stocks = 100

def process_stock_file(fname):
    try:
        yahooData = readYahooHistory(fname)
        return yahooData
    except Exception as e:
        raise e
        return None

# select path of all stock data files in "stock_folder"
files = [join(stock_folder, f) for f in listdir(stock_folder) if isfile(
join(stock_folder, f))]

# assume that we invest only the first 35 stocks (for faster computation
)
files = files[:num_stocks]

# read each line in each file, convert it into the format: (date, value)
rawStocks = [process_stock_file(f) for f in files]

# select only instruments which have more than 5 years of history
# Note: the number of business days in a year is 260
number_of_years = 5
rawStocks = list(filter(lambda instrument: len(instrument)>number_of_yea
rs*260 , rawStocks))

# For testing, print the first 5 entry of the first stock
```

```
#print(rawStocks[0][:10])

#for s in rawStocks[0][:1000]:
#    print(s)

print("\nInstruments with more that 5 years:", len(rawStocks))

# What the fuck happend here?
#(datetime.datetime(1998, 5, 15, 0, 0), 56.375)
#(datetime.datetime(1998, 5, 18, 0, 0), 28.75)
```

```
Instruments with more that 5 years: 75
```

Time alignment for our data

Different types of instruments may trade on different days, or the data may have missing values for other reasons, so it is important to make sure that our different histories align. First, we need to trim all of our time series to the same region in time. Then, we need to fill in missing values. To deal with time series that have missing values at the start and end dates in the time region, we simply fill in those dates with nearby values in the time region.

Question 4.4

Assume that we only focus on the data from 23/01/2009 to 23/01/2014. Write a function named `trimToRegion` to select only the records in that time interval. **Requirements**: after processing, each instrument $i$ has a list of records: $[r_0, r_2,...,r_{m_i}]$ such that $r_0$ and $r_{m_i}$ are assigned, respectively, the first and the last values corresponding to the extremes of the given time interval. For example: $r_0$ should contain the value at date 23/01/2009.

In [441]:

```
# note that the data of crude oild and treasury is only available starti
ng from 26/01/2006
start = datetime(year=2009, month=1, day=23)
end = datetime(year=2014, month=1, day=23)

def trimToRegion(history, start, end):
    def isInTimeRegion(entry):
        (date, value) = entry
        return date >= start and date <= end

    # only select entries which are in the time region
    trimmed = list(filter(isInTimeRegion, history))

    # if the data has incorrect time boundaries, add time boundaries
    if trimmed[0][0] != start:
        trimmed.insert(0, (start, trimmed[0][1]))
    if trimmed[-1][0] != end:
        trimmed.append((end, trimmed[-1][1]))
    return trimmed

# test our function
trimmedStock0  = trimToRegion(rawStocks[0], start, end)
# the first 5 records of stock 0
print(trimmedStock0[:5])
# the last 5 records of stock 0
```

```
print(trimmedStock0[-5:])

assert (trimmedStock0[0][0] == start), "the first record must contain th
e price in the first day of time interval"
assert (trimmedStock0[-1][0] == end), "the last record must contain the
price in the last day of time interval"
```

```
[(datetime.datetime(2009, 1, 23, 0, 0), 19.4), (datetime.datetime(2009,
1, 26, 0, 0), 19.67), (datetime.datetime(2009, 1, 27, 0, 0), 19.809999),
 (datetime.datetime(2009, 1, 28, 0, 0), 20.469999), (datetime.datetime(2
009, 1, 29, 0, 0), 21.41)]
[(datetime.datetime(2014, 1, 16, 0, 0), 37.369999), (datetime.datetime(2
014, 1, 17, 0, 0), 37.470001), (datetime.datetime(2014, 1, 21, 0, 0), 37
.73), (datetime.datetime(2014, 1, 22, 0, 0), 37.779999), (datetime.datet
ime(2014, 1, 23, 0, 0), 37.59)]
```

Dealing with missing values

We expect that we have the price of instruments and factors **in each business day**.
Unfortunately, there are many missing values in our data: this means that we miss data for some days, e.g. we have data for the Monday of a certain week, but not for the subsequent Tuesday.
So, we need a function that helps filling these missing values.

Next, we provide to you the function to fill missing value: read it carefully!

In [442]:

```
def fillInHistory(history, start, end):
    curr = history
    filled = []
    idx = 0
    curDate = start
    numEntries = len(history)
    while curDate < end:

        # if the next entry is in the same day
        # or the next entry is at the weekend
        # but the curDate has already skipped it and moved to the next m
onday
        # (only in that case, curr[idx + 1][0] < curDate )
        # then move to the next entry
        while idx + 1 < numEntries and curr[idx + 1][0] == curDate:
            idx +=1

        # only add the last value of instrument in a single day
        # check curDate is weekday or not
        # 0: Monday -> 5: Saturday, 6: Sunday
        if curDate.weekday() < 5:

            filled.append((curDate, curr[idx][1]))
            # move to the next business day
            curDate += timedelta(days=1)

        # skip the weekends
        if curDate.weekday() >= 5:
            # if curDate is Sat, skip 2 days, otherwise, skip 1 day
            curDate += timedelta(days=(7-curDate.weekday()))

    return filled
```

Question 4.5

Trim data of stocks and factors into the given time interval.

In [443]:

```
#print rawStocks[0]

# trim into a specific time region
# and fill up the missing values
stocks = list(map(lambda stock: \
            fillInHistory(
                trimToRegion(stock, start, end),
            start, end),
        rawStocks))



# merge two factors, trim each factor into a time region
# and fill up the missing values
allfactors = factors1 + factors2
factors = list(map(lambda factor: \
            fillInHistory(
                trimToRegion(factor, start, end),
                start, end),
        allfactors))

# test our code
print("the first 5 records of stock 0:", stocks[0][:5], "\n")
print("the last 5 records of stock 0:", stocks[0][-5:], "\n")
print("the first 5 records of factor 0:", factors[0][:5], "\n")
print("the first 5 records of factor 0:", factors[0][-5:], "\n")
```

```
the first 5 records of stock 0: [(datetime.datetime(2009, 1, 23, 0, 0),
19.4), (datetime.datetime(2009, 1, 26, 0, 0), 19.67), (datetime.datetime
(2009, 1, 27, 0, 0), 19.809999), (datetime.datetime(2009, 1, 28, 0, 0),
20.469999), (datetime.datetime(2009, 1, 29, 0, 0), 21.41)]

the last 5 records of stock 0: [(datetime.datetime(2014, 1, 16, 0, 0), 3
7.369999), (datetime.datetime(2014, 1, 17, 0, 0), 37.470001), (datetime.
datetime(2014, 1, 20, 0, 0), 37.470001), (datetime.datetime(2014, 1, 21,
 0, 0), 37.73), (datetime.datetime(2014, 1, 22, 0, 0), 37.779999)]

the first 5 records of factor 0: [(datetime.datetime(2009, 1, 23, 0, 0),
 43.26), (datetime.datetime(2009, 1, 26, 0, 0), 46.05), (datetime.dateti
me(2009, 1, 27, 0, 0), 45.65), (datetime.datetime(2009, 1, 28, 0, 0), 41
.99), (datetime.datetime(2009, 1, 29, 0, 0), 42.22)]

the first 5 records of factor 0: [(datetime.datetime(2014, 1, 16, 0, 0),
 94.29), (datetime.datetime(2014, 1, 17, 0, 0), 94.17), (datetime.dateti
me(2014, 1, 20, 0, 0), 94.31), (datetime.datetime(2014, 1, 21, 0, 0), 94
.0), (datetime.datetime(2014, 1, 22, 0, 0), 95.2)]
```

Recall that Value at Risk (VaR) deals with **losses over a particular time horizon**. We are not concerned with the absolute prices of instruments, but how those prices **change over** a given period of time. In our project, we will set that length to two weeks: we use the sliding window method to transform time series of prices into an overlapping sequence of price change over two-week intervals.

The figure below illustrates this process. The returns of market factors after each two-week interval is calculated in the very same way.

```
In [444]: def buildWindow(seq, k=2):
              "Returns a sliding window (of width k) over data from iterable data
          structures"
              "   s -> (s0,s1,...s[k-1]), (s1,s2,...,sk), ...                    "
              it = iter(seq)
              result = tuple(islice(it, k))
              if len(result) == k:
                  yield result
              for elem in it:
                  result = result[1:] + (elem,)
                  yield result
```

Question 4.6

Compute the returns of the stocks after each two-week time window.

```
In [445]: def calculateReturn(window):
              # return the change of value after two weeks
              return window[-1][1] - window[0][1]

          def twoWeekReturns(history):
              # we use 10 instead of 14 to define the window
              # because financial data does not include weekends
              return [calculateReturn(entry) for entry in buildWindow(history, 10)
          ]

          stocksReturns = list(map(twoWeekReturns, stocks))
          factorsReturns = list(map(twoWeekReturns, factors))

          # test our functions
          print("the first 5 returns of stock 0:", stocksReturns[0][:5])
          print("the last 5 returns of stock 0:", stocksReturns[0][-5:])
```

```
the first 5 returns of stock 0: [0.8000010000000017, 1.0, 0.720001999999
9974, -0.2799980000000263, -1.5800000000000018]
the last 5 returns of stock 0: [-1.1599999999999966, -1.4599989999999963
, -0.8399999999999963, -0.4599990000000034, 0.0]
```

Alright! Now we have data that is properly aligned to start the training process: stocks' returns and factors' returns, per time windows of two weeks. Next, we will apply the MCS method.

5.5. Summary guidelines to apply the MCS method on the data we prepared

Next, we overview the steps that you have to follow to build a model of your data, and then use Monte Carlo simulations to produce output distributions:

- **Step 1**: Defining the relationship between the market factors and the instrument's returns. This relationship takes the form of a model fitted to historical data.

- **Step 2**: Defining the distributions for the market conditions (particularly, the returns of factors) that are straightforward to sample from. These distributions are fitted to historical data.
- **Step 3**: Generate the data for each trial of a Monte Carlo run: this amount to generating the random values for market conditions along with these distributions.
- **Step 4**: For each trial, from the above values of market conditions, and using the relationship built in step 1, we calculate the return for each instrument and the total return. We use the returns to define an empirical distribution over losses. This means that, if we run 100 trials and want to estimate the 5% VaR, we would choose it as the loss from the trial with the fifth greatest loss.
- **Step 5**: Evaluating the result

5.6. Applying MCS

Step 1: Defining relationship between market factors and instrument's returns

In our simulation, we will use a simple linear model. By our definition of return, a factor return is a **change** in the value of a market factor **over a particular time period**, e.g. if the value of the S&P 500 moves from 2000 to 2100 over a time interval, its return would be 100.

A vector that contains the return of 4 market factors is called a *market factor vector*. Generally, instead of using this vector as features, we derive a set of features from simple transformation of it. In particular, a vector of 4 values is transformed into a vector of length $m$ by function $F$. In the simplest case $F(v) = v$.

Denote $v_t$ the market factor vector, and $f_t$ the transformed features of $v_t$ at time $t$.

$f_{tj}$ is the value of feature $j$ in $f_t$.

Denote $r_{it}$ the return of instrument $i$ at time $t$ and $c_i$ the intercept term of instrument $i$.

We will use a simple linear function to calculate $r_{it}$ from $f_t$:
$$ r_{it} = c_i + \sum_{j=1}^{m}{w_{ij}*f_{tj}} $$

where $w_{ij}$ is the weight of feature $j$ for instrument $i$.

All that above means that given a market factor vector, we have to apply featurization and then use the result as a surrogate for calculating the return of the instruments, using the above linear function.

There are two questions that we should consider: **how we apply featurization to a factor vector?** and **how to pick values for $w_{ij}$?**

How we apply featurization to a factor vector? In fact, the instruments' returns may be non-linear functions of the factor returns. So, we should not use factor returns as features in the above linear function. Instead, we transform them into a set of features with different size. In this Notebook, we can include some additional features in our model that we derive from non-linear transformations of the factor returns. We will try adding two more features for each factor return: its square and its square root values. So, we can still assume that our model is a linear model in the sense that the response variable is a linear function of the new features. *Note that the particular feature transformation described here is meant to be an illustrative example of some of* *the options that are available: it shouldn't be considered as the state of the art in predictive financial modeling!!*.

How to pick values for w_{ij}?

For all the market factor vectors in our historical data, we transform them to feature vectors. Now, we have feature vectors in many two-week intervals and the corresponding instrument's returns in these intervals. We can use Ordinary Least Square (OLS) regression model to estimate the weights for each instrument such that our linear function can fit to the data. The parameters for OLS function are:

- x: The collection of columns where **each column** is the value of **a feature** in many two-week interval
- y: The return of an instrument in the corresponding time interval of x.

The figure below shows the basic idea of the process to build a statistical model for predicting the returns of stock X.

Question 5

Question 5.1

Currently, our data is in form of: $$ factorsReturns= \begin{bmatrix} r_{00} & r_{01} & r_{02} & ... & r_{0k} \\ r_{10} & r_{11} & r_{12} & ... & r_{1k} \\ ... & ... & ... & ... & ... \\ r_{n0} & r_{n1} & r_{n2} & ... & r_{nk}\\ \end{bmatrix} $$
$$ stocksReturns= \begin{bmatrix} s_{00} & s_{01} & s_{02} & ... & s_{0k} \\ s_{10} & s_{11} & s_{12} & ... & s_{1k} \\ ... & ... & ... & ... & ... \\ s_{n0} & s_{n1} & s_{n2} & ... & s_{nk}\\ \end{bmatrix} $$

Where, r_{ij} is the return of factor i^{th} in time window j^{th}, k is the number of time windows, and n is the number of factors. A similar definition goes for s_{ij}.

In order to use OLS, the parameter must be in form of:

$$ x=factorsReturns^T = \begin{bmatrix} r_{00} & r_{10} & ... & r_{n0} \\ r_{01} & r_{11} & ... & r_{n1} \\ r_{02} & r_{12} & ... & r_{n2}\\ ... & ... & ... & ... \\ r_{0k} & r_{1k} & ... & r_{nk}\\ \end{bmatrix} $$

Whereas, y can be any row in `stocksReturns`.

So, we need a function to transpose a matrix. Write a function named `transpose` to do just that.

In [446]:
```
def transpose(matrix):
    m_tr = [list(x) for x in zip(*matrix)]
    return m_tr

# test function
assert (transpose([[1,2,3], [4,5,6], [7,8,9]]) == [[1, 4, 7], [2, 5, 8],
 [3, 6, 9]]), "Function transpose runs incorrectly"
```

Question 5.2

Write a function named `featurize` that takes a list factor's returns $[x_1, x_2,...,x_k]$ and transform it into a new list of features $[u_1,u_2,..,u_k, v_1, v_2,..,v_k, x_1,x_2,...,x_k]$. Where, $u_i$ = $\left\{ \begin{array}{ll} x_i^2 & \mbox{if } x_i \geq 0 \\ -x_i^2 & \mbox{if } x_i < 0 \end{array} \right. $

and $v_i$ = $\left\{ \begin{array}{ll} \sqrt{x_i} & \mbox{if } x_i \geq 0 \\ -\sqrt{x_i} & \mbox{if } x_i < 0 \end{array} \right. $

In [447]:

```
import math
def featurize(factorReturns):
    squaredReturns = [(x**2)*(1 if x>0 else -1) for x in factorReturns]
    squareRootedReturns = [math.sqrt(abs(x))*(1 if x>0 else -1) for x in
 factorReturns]
    # concat new features
    return squaredReturns + squareRootedReturns + factorReturns

# test our function
assert (featurize([4, -9, 25]) == [16, -81, 625, 2, -3, 5, 4, -9, 25]),
"Function runs incorrectly"
```

Question 5.3

Using OLS, estimate the weights for each feature on each stock. What is the shape of `weights` (size of each dimension)? Explain it.

In [448]:

```
factorMat[0]
```

Out[448]:

```
[-2.9600000000000001,
 0.41800000000000015,
 9.590026999999964,
 57.80993699999999,
 44.25,
 0.0,
 0.0]
```

In [449]:

```
factor_columns[0]
```

Out[449]:

```
array([  1.00000000e+00,  -8.76160000e+00,   1.74724000e-01,
         9.19686179e+01,   3.34198882e+03,   1.95806250e+03,
        -0.00000000e+00,  -0.00000000e+00,  -1.72046505e+00,
         6.46529195e-01,   3.09677687e+00,   7.60328462e+00,
         6.65206735e+00,  -0.00000000e+00,  -0.00000000e+00,
        -2.96000000e+00,   4.18000000e-01,   9.59002700e+00,
         5.78099370e+01,   4.42500000e+01,   0.00000000e+00,
                    0.00000000e+00])
```

In [450]:

```
def estimateParams(y, x):
    return sm.OLS(y, x).fit().params

# transpose factorsReturns
factorMat = transpose(factorsReturns)

# featurize each row of factorMat
factorFeatures = list(map(featurize,factorMat))

# OLS require parameter is a numpy array
factor_columns = np.array(factorFeatures)

#add a constant - the intercept term for each instrument i.
factor_columns = sm.add_constant(factor_columns, prepend=True)

# estimate weights
weights = [estimateParams(stockRet, factor_columns) for stockRet in stoc
ksReturns]



print("factor_columns:", len(factor_columns),len(factor_columns[0]))
print("stocksReturns:", len(stocksReturns),len(stocksReturns[0]))

print("weights length:", len(weights),len(weights[0]))
print("weights first row:", weights[0])
```

```
factor_columns: 1295 13
stocksReturns: 75 1295
weights length: 75 13
weights first row: [ -4.35466694e-03  -3.64753449e-04  -8.03349863e+00
  7.93239441e-05
   -2.17409832e-05   8.81875154e-02  -1.40571325e+00  -8.49863338e-03
   -4.99535942e-02  -3.69387457e-02   4.72385267e+00   7.23621329e-03
    1.06892611e-02]
```

COMMENT:

The two dimensions of our weight vector are the number of instruments (75, section 4.3) after filtering out those with less than 5 years of history and the number of features after feature expansion: Four initial features, with their respective squares and square roots, and an extra interception feature to reduce the bias (4+4+4+1 = 13).

Step 2: Defining the distributions for the market conditions

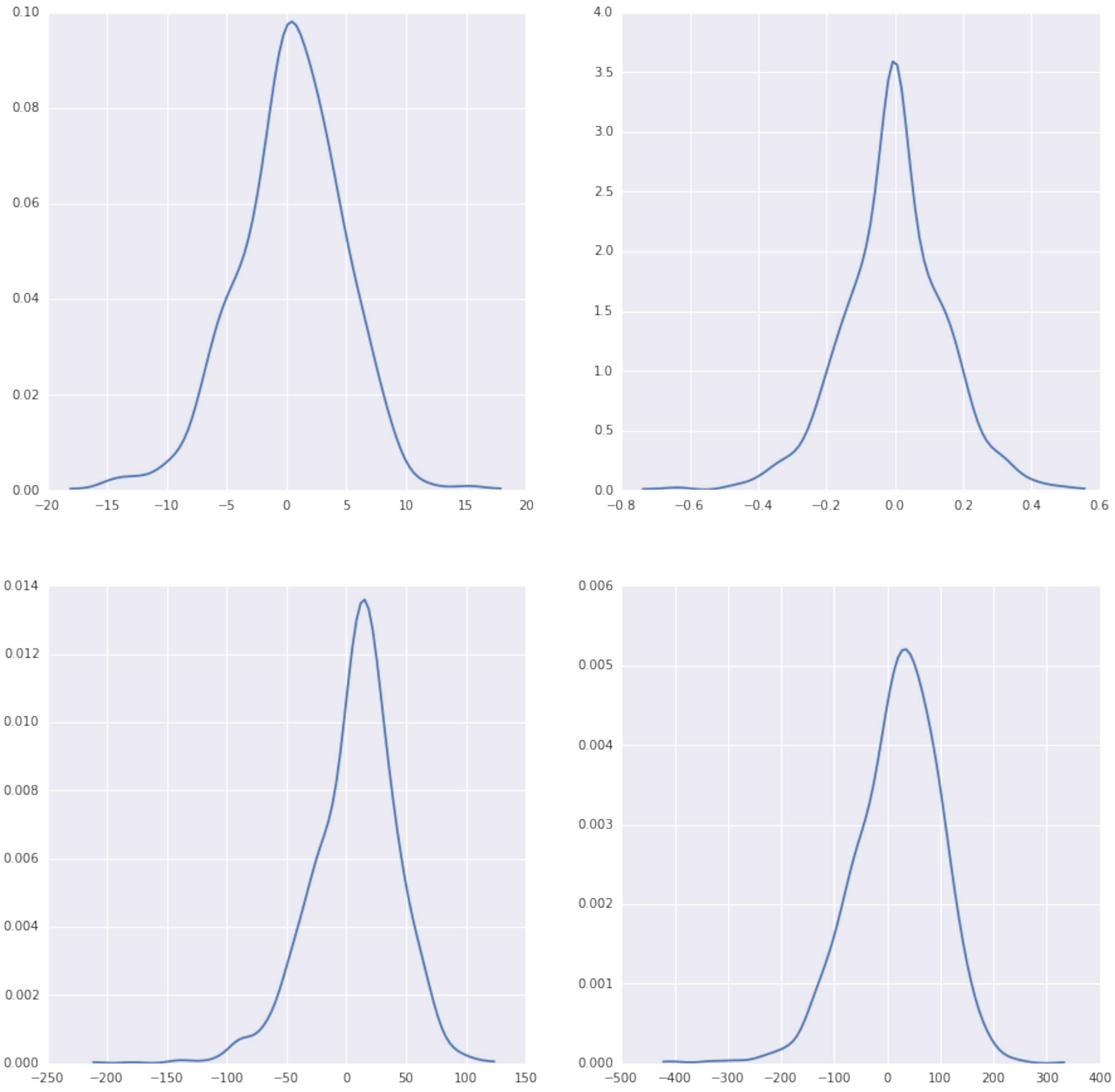
Since we cannot define the distributions for the market factors directly, we can only approximate their distribution. The best way to do that, is plotting their value. However, these values may fluctuate quite a lot.

Next, we show how to use the Kernel density estimation (KDE) technique to approximate such distributions. In brief, kernel density estimation is a way of smoothing out a histogram: this is achieved by assigning (or centering) a probability distribution (usually a normal distribution) to each data point, and then summing. So, a set of two-week-return samples would result in a large number of "super-imposed" normal distributions, each with a different mean.

To estimate the probability density at a given point, KDE evaluates the PDFs of all the normal distributions at that point and takes their average. The smoothness of a kernel density plot depends on its *bandwidth*, and the standard deviation of each of the normal distributions. For a brief introduction on KDE, please refer to this link.

In [451]:

```
from statsmodels.nonparametric.kernel_density import KDEMultivariate
from statsmodels.nonparametric.kde import KDEUnivariate
import matplotlib.pyplot as plt
import scipy


f, axarr = plt.subplots(2, 2)
f.set_figheight(15)
f.set_figwidth(15)

def plotDistribution(samples, a, b):
    vmin = min(samples)
    vmax = max(samples)
    stddev = np.std(samples)

    domain = np.arange(vmin, vmax, (vmax-vmin)/100)

    # a simple heuristic to select bandwidth
    bandwidth = 1.06 * stddev * pow(len(samples), -.2)

    # estimate density
    kde = KDEUnivariate(samples)
    kde.fit(bw=bandwidth)
    density = kde.evaluate(domain)

    # plot
    axarr[a, b].plot(domain, density)

plotDistribution(factorsReturns[0], 0, 0)
plotDistribution(factorsReturns[1], 0, 1)
plotDistribution(factorsReturns[2], 1, 0)
plotDistribution(factorsReturns[3], 1, 1)
plt.show()
```



For the sake of simplicity, we can say that our smoothed versions of the returns of each factor can be represented quite well by a normal distribution. Of course, more exotic distributions, perhaps with fatter tails, could fit more closely the data, but it is outside the scope of this Notebook to proceed in this way.

Now, the simplest way to sample factors returns is to use a normal distribution for each of the factors, and sample from these distributions independently. However, this approach ignores the fact that market factors are often correlated. For example, when the price of crude oil is down, the price of treasury bonds is down too. We can check our data to verify about the correlation.

Question 6

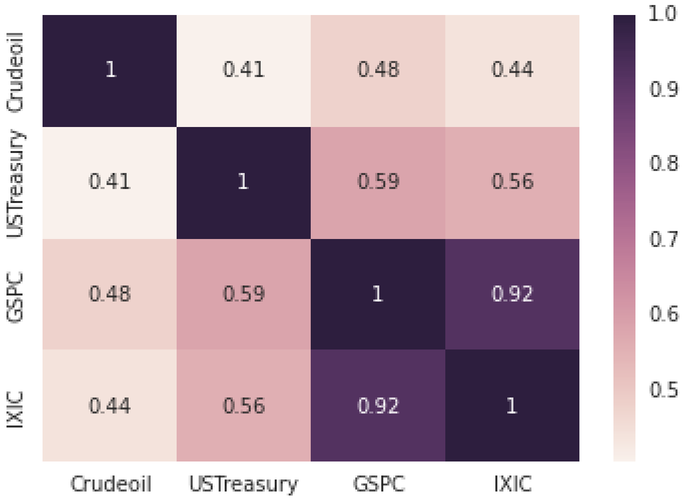
Question 6.1

Calculate the correlation between market factors and explain the result.

HINT function `np.corrcoef` might be useful.

In [452]:
```
correlation = np.corrcoef(factorsReturns)
factor_names = ['Crudeoil', 'USTreasury', 'GSPC', 'IXIC']
correlation
sns.heatmap(correlation,
            xticklabels=factor_names,
            yticklabels=factor_names, annot=True)
```

Out[452]:



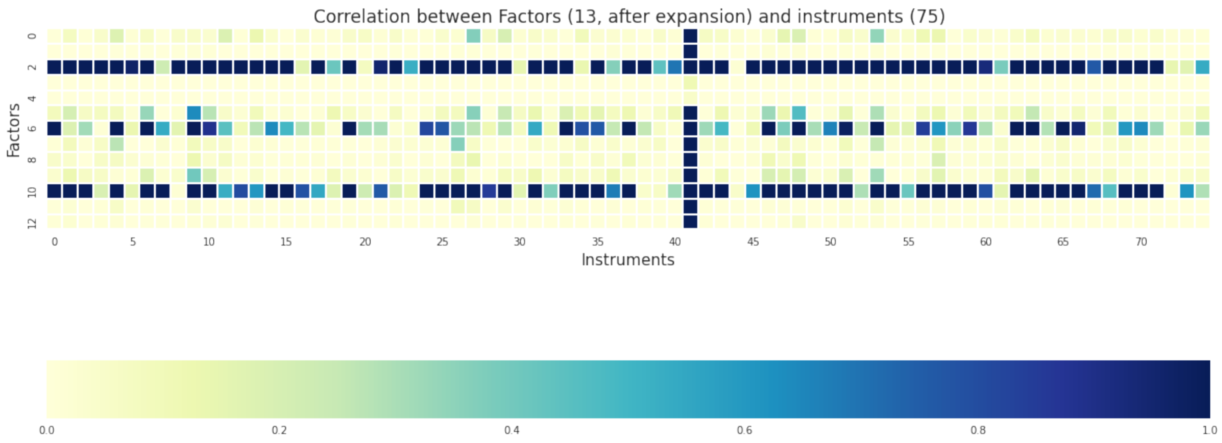
COMMENT:

It's a symmetric matrix, as expected from correlation. Values range between 0 and 1, being 0 the absolute independency and 1 the absoulte linear correlation. As expected, the cross-correlation between each feature and himself is 1 (hence the diagonal filled with ones).

Other than that, we see a very strong correlation between GSPC and IXIC, which was expected since they are two stock indicators: S&P500 and Nasdaq. USTreasury is expected to be much more stable, and crudeoil doesn't affect all stock values equally (e.g.: a company based on clean energies will be affected inversely to one based on transportation)

In [453]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# Generate a mask for the upper triangle
mask = np.zeros_like(correlation, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=[20,15])

absWeights = transpose(list(map(lambda x: list(map(abs ,x.tolist())), we
ights)))

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(absWeights,  cmap="YlGnBu", vmax=1.0,square=True, xticklabel
s=5, yticklabels=2,linewidths=.5,
            cbar_kws={"orientation": "horizontal"}, ax=ax)
ax.set_title('Correlation between Factors (13, after expansion) and inst
ruments (75)', fontsize=17)
ax.set_xlabel('Instruments', fontsize=15)
ax.set_ylabel('Factors', fontsize=15)
```

Out[453]:



Factors										
Interception	square				sqrt				Linear	
	CrudeOil	USTreas	GSCP	IXIC	CrudeOil	USTreas	GSCP	IXIC	CrudeOil	USTr
0	1	2	3	4	5	6	7	8	9	10

COMMENT:

We decided to analyze the weights between factors and instruments, in order to assess the relative importance of each factor. Let's analyze the results one by one, remembering the order of the factors:

- **Crudeoil** -> Only the sqrt series gives away a bit of information; the rest could be discarded
- **USTreasury** -> With a great difference among the rest, this is the most influential feature in all its expansions. Squared is the most significant value, linear is a bit less significant and sqrt is the least of the three (nevertheless it's much more important than any other features)
- **GSPC** -> Almost insignificant.
- **IXIC** -> Almost insignificant.
- **Interception** -> Doesn't affect almost at all

We should consider getting rid of the instrument number **41** since it displays some strange behaviour when compared to the rest.

If we would have an arsenal of one hundred factors, we could use this tool/technique to determine which factors should we keep and which ones don't impact the value of the instruments at all, and thus the VaR.

The multivariate normal distribution can help here by taking the correlation information between the factors into account. Each sample from a multivariate normal distribution can be thought of as a vector. Given values for all of the dimensions but one, the distribution of values along that dimension is normal. But, in their joint distribution, the variables are not independent.

For this use case, we can write:

$$ \left(\begin{array}{c}f_{1}\\f_{2}\\f_{3}\\f_{4} \end{array}\right) \sim N \left[\left(\begin{array}{c} \mu_1\\ \mu_2 \\ \mu_3 \\ \mu_4 \end{array} \right), \left(\begin{array}{cccc} \sigma^2_1 & \rho_{12} \sigma_1\sigma_2 & \rho_{13} \sigma_1\sigma_3 & \rho_{14} \sigma_1\sigma_4 \\ \rho_{12}\sigma_2\sigma_1 & \sigma^2_2 & \rho_{23} \sigma_2\sigma_3 & \rho_{24} \sigma_2\sigma_4\\ \rho_{13} \sigma_3\sigma_1 & \rho_{23} \sigma_3\sigma_2 & \sigma^2_3 & \rho_{34} \sigma_3\sigma_4 \\ \rho_{14} \sigma_4\sigma_1 & \rho_{24} \sigma_4\sigma_2 & \rho_{34} \sigma_3\sigma_4 & \sigma_4^2 \\ \end{array} \right) \right] $$

Or,

$$ f_t \sim N(\mu, \sum) $$

Where f_1, f_2, f_3 and f_4 are the market factors, σ_i is the standard deviation of factor i, μ is a vector of the empirical means of the returns of the factors and $\sum$ is the empirical covariance matrix of the returns of the factors.

The multivariate normal is parameterized with a mean along each dimension and a matrix describing the covariance between each pair of dimensions. When the covariance matrix is diagonal, the multivariate normal reduces to sampling along each dimension independently, but placing non-zero values in the off-diagonals helps capture the relationships between variables. Whenever having the mean of this multivariate normal distribution and its covariance matrix, we can generate the sample values for market factors.

Next, we will calculate the mean and the covariance matrix of this multivariate normal distribution from the historical data.

Question 6.2

Calculate the covariance matrix $\sum$ and the means μ of factors' returns then generate a random vector of factors return that follows a multivariate normal distribution $\sim N(\mu, \sum)$

HINT Function `np.cov` can help calculating covariance matrix. Function `np.random.multivariate_normal(, )` is often used for generating samples.

In [454]:

```
factorCov = np.cov(factorsReturns)
factorMeans = [sum(factorsReturns[i])/len(factorsReturns[i]) for i in ra
nge(len(factorsReturns))]
sample = np.random.multivariate_normal(factorMeans, factorCov)
print('Factor Covariance:\n', factorCov, "\n")
print('Factor Means:\n',factorMeans, "\n")
print('Sample: \n',sample)
```

```
Factor Covariance:
 [[  1.99479662e+01   2.70103191e-01   7.70665225e+01   1.61846128e+02]
 [  2.70103191e-01   2.22473953e-02   3.14993992e+00   6.81903796e+00]
 [  7.70665225e+01   3.14993992e+00   1.29672509e+03   2.71521187e+03]
 [  1.61846128e+02   6.81903796e+00   2.71521187e+03   6.70934736e+03]]

Factor Means:
 [0.3532664092664094, -0.0030517374517374466, 6.970339789189207, 18.7377
21800000003]

Sample:
 [  5.16111351  -0.0844882  -10.85362392   8.33807266]
```

Step 3&4: Generating samples, running simulation and calculating the VaR

We define some functions that helps us calculating VaR 5%. You will see that the functions below are pretty complicated! This is why we provide a solution for you: however, study them well!!

The basic idea of calculating VaR 5% is that we need to find a value such that only 5% of the losses are bigger than it. That means the 5th percentile of the losses should be VaR 5%.

VaR can sometimes be problematic though, since it does give any information on the extent of the losses which can exceed the VaR estimate. CVar is an extension of VaR that is introduced to deal with this problem. Indeed, CVaR measures the expected value of the loss in those cases where VaR estimate has been exceeded.

In [455]:

```python
def fivePercentVaR(trials):
    numTrials = trials.count()
    topLosses = trials.takeOrdered(max(round(numTrials/20.0), 1))
    return topLosses[-1]

# an extension of VaR
def fivePercentCVaR(trials):
    numTrials = trials.count()
    topLosses = trials.takeOrdered(max(round(numTrials/20.0), 1))
    return sum(topLosses)/len(topLosses)

def bootstrappedConfidenceInterval(
      trials, computeStatisticFunction,
      numResamples, pValue):
    stats = []
    for i in range(0, numResamples):
        resample = trials.sample(True, 1.0)
        stats.append(computeStatisticFunction(resample))
    sorted(stats)
    lowerIndex = int(numResamples * pValue / 2 - 1)
    upperIndex = int(np.ceil(numResamples * (1 - pValue / 2)))
    return (stats[lowerIndex], stats[upperIndex])
```

Next, we will run the Monte Carlo simulation 10,000 times, in parallel using Spark. Since your cluster has 12 cores (two Spark worker nodes, each with 6 cores), we can set `parallelism = 12` to dispatch simulation on these cores, across the two machines (remember, those are not really "physical machines", they are Docker containers running in our infrastructure).

Question 7

Complete the code below to define the simulation process and calculate VaR 5%.

In [456]:
```
# RUN SILMULATION
def simulateTrialReturns(numTrials, factorMeans, factorCov, weights):
    trialReturns = []
    for i in range(0, numTrials):
        # generate sample of factors' returns
        trialFactorReturns = np.random.multivariate_normal(factorMeans,
factorCov)

        # featurize the factors' returns
        trialFeatures = featurize(trialFactorReturns.tolist())

        # insert weight for intercept term
        trialFeatures.insert(0,1)

        trialTotalReturn = 0

        # calculate the return of each instrument
        # then calulate the total of return for this trial features

        #trialTotalReturn = sum(np.dot(weights,trialFeatures).tolist())
        trialreturns = np.array(weights).dot(trialFeatures)
        for ret in trialreturns:
            trialTotalReturn += ret

        trialReturns.append(trialTotalReturn)
    return trialReturns



parallelism = 12
numTrials = 10000
trial_indexes = list(range(0, parallelism))
seedRDD = sc.parallelize(trial_indexes, parallelism)
bFactorWeights = sc.broadcast(weights)

trials = seedRDD.flatMap(lambda idx: \
                simulateTrialReturns(
                    max(int(numTrials/parallelism), 1),
                    factorMeans, factorCov,
                    bFactorWeights.value
                ))
trials.cache()
```

Out[456]:
```
PythonRDD[1242] at RDD at PythonRDD.scala:48
```

In [457]:
```
valueAtRisk = fivePercentVaR(trials)
```

```
conditionalValueAtRisk = fivePercentCVaR(trials)

print("Value at Risk(VaR) 5: %.3f"% valueAtRisk)
print("Conditional Value at Risk(CVaR) 5: %.3f"% conditionalValueAtRisk)
```

```
Value at Risk(VaR) 5: -586.548
Conditional Value at Risk(CVaR) 5: -988.260
```

The value of VaR depends on how many invested stocks and the chosen distribution of random variables. Assume that we get VaR 5% = -2.66, that means that there is a 0.05 probability that the portfolio will fall in value by more than \$2.66 over a two weeks' period if there is no trading. In other words, the loses are less than \$2.66 over two weeks' period with 95% confidence level. When a loss over two weeks is more than \$2.66, we call it **failure** (or **exception**). Informally, because of 5% probability, we expect that there are only $0.05*W$ failures out of total W windows.

Step 5: Evaluating the results using backtesting method

In general, the error in a Monte Carlo simulation should be proportional to 1/sqrt(n), where n is the number of trials. This means, for example, that quadrupling the number of trials should approximately cut the error in half. A good way to check the quality of a result is backtesting on historical data. Backtesting is a statistical procedure where actual losses are compared to the estimated VaR. For instance, if the confidence level used to calculate VaR is 95% (or VaR 5%), we expect only 5 failures over 100 two-week time windows.

The most common test of a VaR model is counting the number of VaR failures, i.e., in how many windows, the losses exceed VaR estimate. If the number of exceptions is less than selected confidence level would indicate, the VaR model overestimates the risk. On the contrary, if there are too many exceptions, the risk is underestimated. However, it's very hard to observe the amount of failures suggested by the confidence level exactly. Therefore, people try to study whether the number of failures is reasonable or not, or will the model be accepted or rejected.

One common test is Kupiec's proportion-of-failures (POF) test. This test considers how the portfolio performed at many historical time intervals and counts the number of times that the losses exceeded the VaR. The null hypothesis is that the VaR is reasonable, and a sufficiently extreme test statistic means that the VaR estimate does not accurately describe the data. The test statistic is computed as:
$$ -2ln\Bigg(\frac{(1-p)^{T-x}p^x}{(1-\frac{x}{T})^{T-x}(\frac{x}{T})^x}\Bigg) $$

where:

p is the quantile-of-loss of the VaR calculation (e.g., in VaR 5%, p=0.05),

x (the number of failures) is the number of historical intervals over which the losses exceeded the VaR

T is the total number of historical intervals considered

Or we can expand out the log for better numerical stability:
$$ \begin{equation} -2\Big((T-x)ln(1-p)+x*ln(p)-(T-x)ln(1-\frac{x}{T})-x*ln(\frac{x}{T})\Big) \end{equation} $$

If we assume the null hypothesis that the VaR is reasonable, then this test statistic is drawn from a chi-squared distribution with a single degree of freedom. By using Chi-squared distribution, we can find the `p-value` accompanying our test statistic value. If `p-value` exceeds the critical value of the Chi-squared distribution, we do have sufficient evidence to reject the null hypothesis that the model is reasonable. Or we can say, in that case, the model is considered as inaccurate.

For example, assume that we calculate VaR 5% (the confidence level of the VaR model is 95%) and get value VaR = 2.26. We also observed 50 exceptions over 500 time windows. Using the formula above, the test statistic `p-value` is calculated and equal to `8.08`. Compared to `3.84`, the critical value of Chi-squared distribution with one degree of freedom at probability 5%, the test statistic is larger. So, the model is rejected. The critical values of Chi-squared can be found by following this link. However, in this Notebook, it's not a good idea to find the corresponding critical value by looking in a "messy" table, especially when we need to change the confidence level. Instead, from `p-value`, we will calculate the probability of the test statistic in Chi-square thanks to some functions in package `scipy`. If the calculated probability is smaller than the quantile of loss (e.g, 0.05), the model is rejected and vice versa.

Question 8

Question 8.1

Write a function to calculate the number of failures, that is when the losses (in the original data) exceed the VaR.

HINT

- First, we need to calculate the total loss in each 2-week time interval
- If the total loss of a time interval exceeds VaR, then we say that our VaR fails to estimate the risk in that time interval
- Return the number of failures

NOTE The loss is often having negative value, so, be careful when compare it to VaR.

In [458]:

```
from scipy import stats
import math

def countFailures(stocksReturns, valueAtRisk):
    failures = 0
    # iterate over time intervals
    for i in range(0, len(stocksReturns[0])):
        # calculate the losses in each time interval
        loss = sum([value[i] for value in stocksReturns])

        # if the loss exceeds VaR
        if loss < valueAtRisk:
            failures += 1
    return failures
```

Question 8.2

Write a function named `kupiecTestStatistic` to calculate the test statistic which was described in the above equation.

In [459]:

```python
def kupiecTestStatistic(total, failures, confidenceLevel):
    failureRatio = failures/total
    logNumer = (total - failures) * np.log(1 - confidenceLevel) + failures * np.log(confidenceLevel)
    logDenom = (total - failures) * np.log(1 - failureRatio) + failures * np.log(failureRatio)
    return -2 * (logNumer - logDenom)

# test the function
assert (round(kupiecTestStatistic(250, 36, 0.1), 2) == 4.80), "function kupiecTestStatistic runs incorrectly"
```

Now we can find the p-value accompanying our test statistic value.

In [460]:

```python
def kupiecTestPValue(stocksReturns, valueAtRisk, confidenceLevel):
    failures = countFailures(stocksReturns, valueAtRisk)
    print("num failures:", failures)
    if failures == 0:
        # the model is very good
        return 1
    total = len(stocksReturns[0])
    testStatistic = kupiecTestStatistic(total, failures, confidenceLevel)
    #return 1 - stats.chi2.cdf(testStatistic, 1.0)
    return stats.chisqprob(testStatistic, 1.0)

varConfidenceInterval = bootstrappedConfidenceInterval(trials, fivePercentVaR, 100, 0.05)
cvarConfidenceInterval = bootstrappedConfidenceInterval(trials, fivePercentCVaR, 100, .05)
print("VaR confidence interval: " , varConfidenceInterval)
print("CVaR confidence interval: " , cvarConfidenceInterval)
print("Kupiec test p-value: " , kupiecTestPValue(stocksReturns, valueAtRisk, 0.05))
```

```
VaR confidence interval:  (-615.64631098428345, -624.82407827385191)
CVaR confidence interval:  (-999.73198389203014, -963.96432441143213)
num failures: 241
Kupiec test p-value:  7.55594138964e-69
```

Question 8.3

Discuss the results you have obtained

COMMENT:

Investment in 35 stocks	

VaR confidence interval	(-19.954194600895239, -19.936524213023823)
CVaR confidence interval	(-24.93940415592181, -25.805224770328739)
Num failures	104 - 1.04%
Kupiec test p-value	3.87023503002e-06

Now, what does this tell us?
The range in the VaR confidence interval is 0.0177 meaning that the 95% of the values will be close to our predictions.

However, since the p-value is far from 0.05 (the minimum value required for a good model) probably because of simplifications, our model has a high bias, and thus it doesn't represent the real world results with accuracy.

Question 9

Assume that we invest in more than 100 stocks. Use the same market factors as for the previous questions to estimate VaR by running MCS, then validate your result. What is the main observation you have, once you answer this question? When you plan to invest in more instruments, how is your ability to predict the risk going to be affected?

COMMENT:

Investment in 100 stocks	
VaR confidence interval	(-696.91913934748061, -696.12843994443017)
CVaR confidence interval	(-1012.7641215060884, -977.57349530421345)
Num failures	214 - 2.14%
Kupiec test p-value	2.14454989582e-52

Now, what does this tell us?
The Var increased in -21 to -696, and all the other values are also increased substantially -> much higher error rates. In short, increasing our portfolio size really affects our variance negatively,increasing the risk considerably. If we already had a bad model, using it masively is a definitely a bad idea.

NOTE: we re-ran all the sections in the notebook with a hundred instruments. This changes slightly the results in some sections, which we modified accordingly.

Question 10

In the previous questions, we used the normal distributions to sample the factors returns. Try to study how results vary when selecting other probability distributions: our goal is to improve the result of our MCS.

COMMENT:

There are several combinations to be considered:

- Triangular: oversimplifies the variance, thus it's hard to fit distributions in it. Not very suitable.
- Uniform: Never! You will be losing all the information from the distribution. Only usable in case we had absolutely no information from factors.
- Normal: since most of the factors come from values ranging 0 to infinite, it's more corect to ose lognormal instead (otherwise we would have to filter negative values).

Also there is a problem with most random variables implemented in numpy: they don't have multivariate option!

Our best solution would be to implement a linear combination of two (or more) gaussians. In order to do that, the best approach is to look at the distributions on the factors and try to emmulate what we see (check section 6 for details). Our best candidate for a sumation of two gaussians is the US Treasury data, with a small-variation gaussian and a high-variation one. It would be as simple as having two models, one with the first low covariance and the other with the highest covariance, and then do the average of both values.

Regrettably, we didn't have the time to implement it. Nevertheless, we leave a comment on where should it be changed.

In [461]:
```
def simulateTrialReturnsTriang(numTrials, factorMeans, factorCov, weight
s):
    trialReturns = []
    signFactorMeans = np.sign(factorMeans)
    absFactorMeans = np.absolute(factorMeans)

    for i in range(0, numTrials):
        # generate sample of factors' returns
        trialFactorReturns = np.exp(np.random.triangular(np.log(absFacto
rMeans), factorCov))
        # HERE WE SHOULD ADD A SECOND LINE WITH A CHANGE IN THE factorCo
```

```
v only for the USTreasury
        # HERE WE SHOULD AVERAGE BOTH MODELS.
        trialFactorReturns = trialFactorReturns*signFactorMeans

        # featurize the factors' returns
        trialFeatures = featurize(trialFactorReturns.tolist())

        # insert weight for intercept term
        trialFeatures.insert(0,1)

        # calculate the return of each instrument
        # then calulate the total of return for this trial features
        trialTotalReturn = sum(np.dot(weights,trialFeatures).tolist())

        trialReturns.append(trialTotalReturn)
    return trialReturns
```

6 Adding Extra Factors

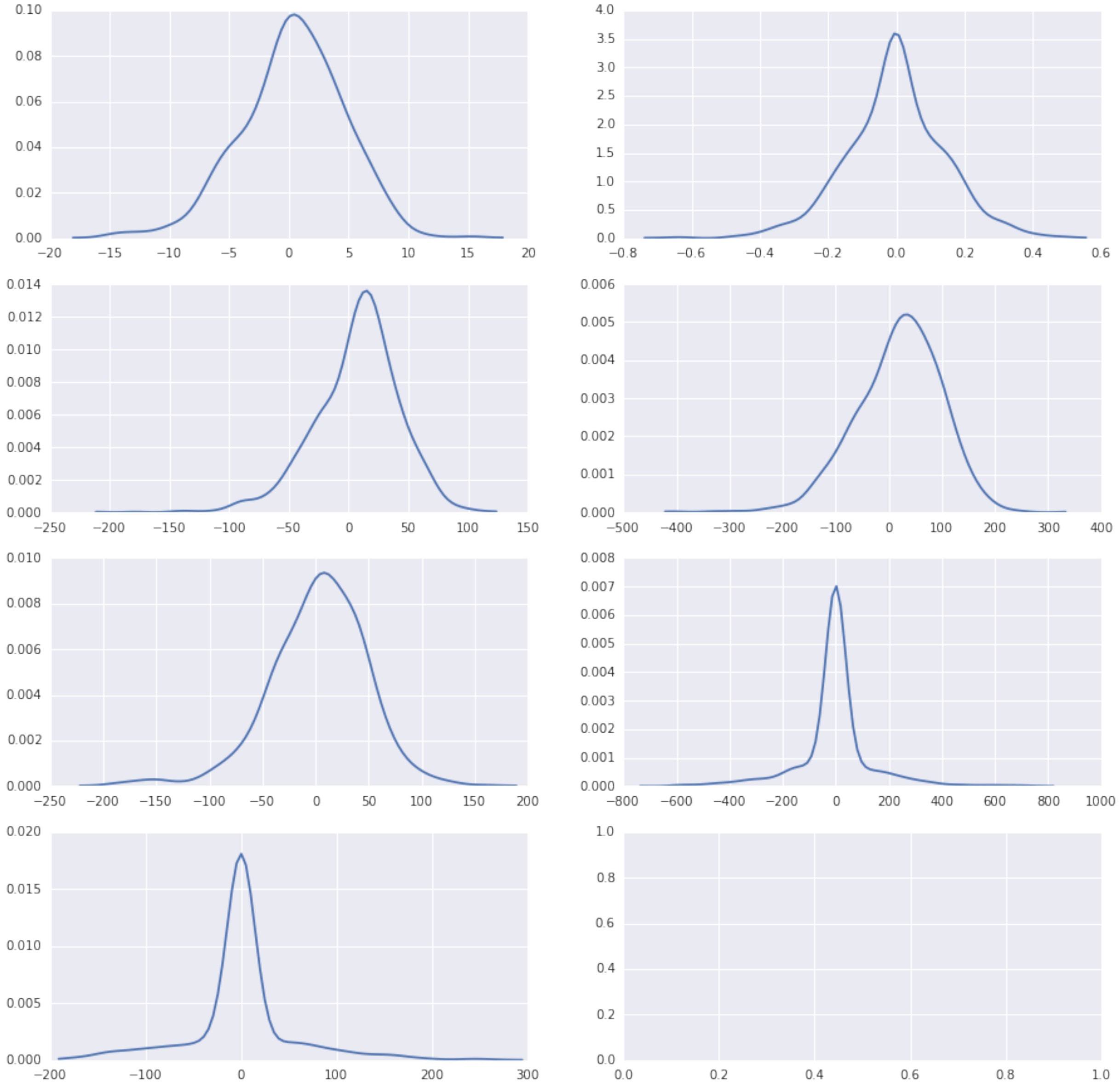
We wanted to exsperimebt with several other factors. We went for this three:

- Copper
- Gold
- Lead

Beneath you can see the different distributions and the correlation+impact on the instruments. Finally, we analyze the VaR with all of them.

In [343]:

```
f, axarr = plt.subplots(4, 2)
f.set_figheight(15)
f.set_figwidth(15)

def plotDistribution(samples, a, b):
    vmin = min(samples)
    vmax = max(samples)
    stddev = np.std(samples)

    domain = np.arange(vmin, vmax, (vmax-vmin)/100)

    # a simple heuristic to select bandwidth
    bandwidth = 1.06 * stddev * pow(len(samples), -.2)

    # estimate density
    kde = KDEUnivariate(samples)
    kde.fit(bw=bandwidth)
    density = kde.evaluate(domain)

    # plot
    axarr[a, b].plot(domain, density)

plotDistribution(factorsReturns[0], 0, 0)
plotDistribution(factorsReturns[1], 0, 1)
plotDistribution(factorsReturns[2], 1, 0)
plotDistribution(factorsReturns[3], 1, 1)
plotDistribution(factorsReturns[4], 2, 0)
plotDistribution(factorsReturns[5], 2, 1)
plotDistribution(factorsReturns[6], 3, 0)
plt.show()
```



COMMENT:

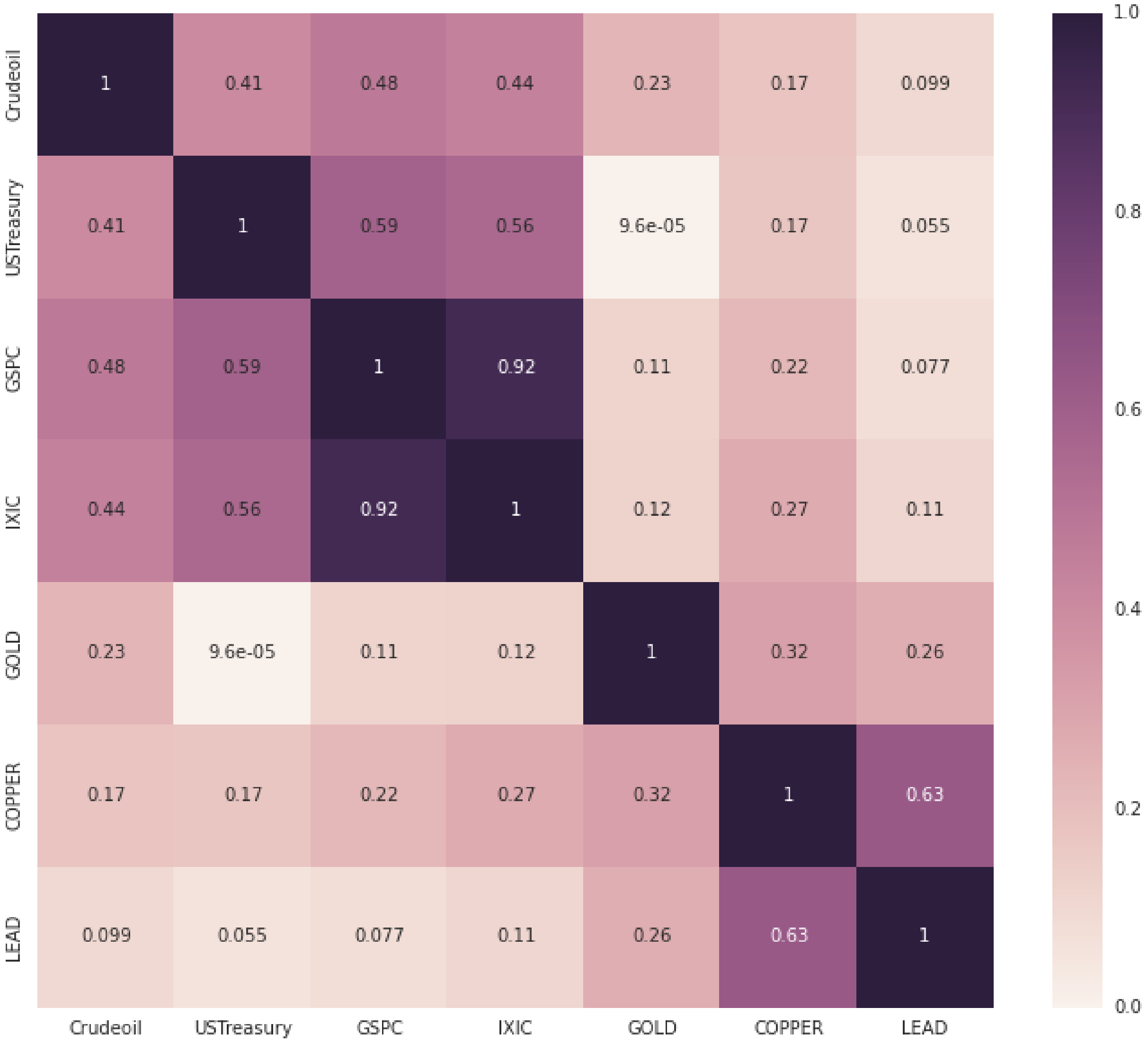
The distributions are in this order (left to right, up to bottom): 'Crudeoil', 'USTreasury', 'GSPC', 'IXIC', 'GOLD', 'COPPER', 'LEAD'

Clearly the distribution that can be turned into a double gaussian model is the USTreasury (for previous section).

In [421]:

```
correlation = np.corrcoef(factorsReturns)
factor_names = ['Crudeoil', 'USTreasury', 'GSPC', 'IXIC', 'GOLD', 'COPPE
R', 'LEAD']
correlation
plt.figure(figsize=[12,10])
sns.heatmap(correlation,
            xticklabels=factor_names,
            yticklabels=factor_names, annot=True)
```

Out[421]:



	Crudeoil	USTreasury	GSPC	IXIC	GOLD	COPPER	LEAD
Crudeoil	1	0.41	0.48	0.44	0.23	0.17	0.099
USTreasury	0.41	1	0.59	0.56	9.6e-05	0.17	0.055
GSPC	0.48	0.59	1	0.92	0.11	0.22	0.077
IXIC	0.44	0.56	0.92	1	0.12	0.27	0.11
GOLD	0.23	9.6e-05	0.11	0.12	1	0.32	0.26
COPPER	0.17	0.17	0.22	0.27	0.32	1	0.63
LEAD	0.099	0.055	0.077	0.11	0.26	0.63	1

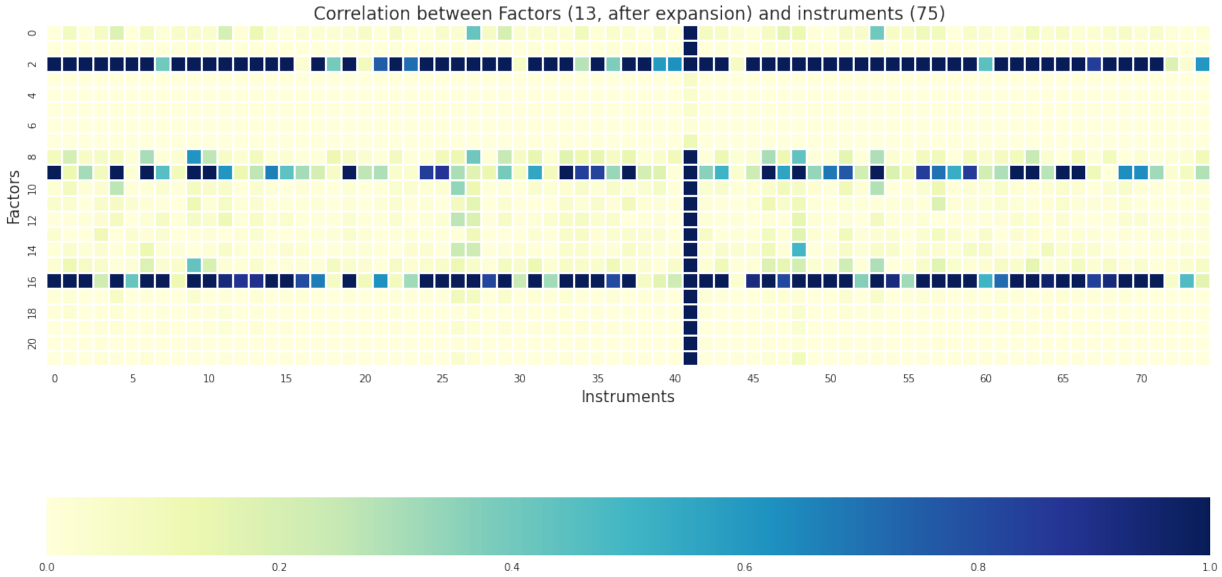
COMMENT:

Most of the values are uncorrelated with each other, except copper and lead that seem to have a high correlation.

In [422]:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# Generate a mask for the upper triangle
mask = np.zeros_like(correlation, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=[20,15])

absWeights = transpose(list(map(lambda x: list(map(abs ,x.tolist())), we
ights)))

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(absWeights,  cmap="YlGnBu", vmax=1.0,square=True, xticklabel
s=5, yticklabels=2,linewidths=.5,
                cbar_kws={"orientation": "horizontal"}, ax=ax)
ax.set_title('Correlation between Factors (13, after expansion) and instruments (75)', fontsize=17)
ax.set_xlabel('Instruments', fontsize=15)
ax.set_ylabel('Factors', fontsize=15)
```

Out[422]:



COMMENT:

As we can see, most of them have very low correlation to the actual instruments we are considering. Nevertheless, the squared version of them does affect some certain instruments.

Our Var results are lower than before:

Value at Risk(VaR) 5: -1056.852
Conditional Value at Risk(CVaR) 5: -1396.144
VaR confidence interval: (-1059.0618529045107, -1042.1359638067859)
CVaR confidence interval: (-1351.1185455516584, -1416.9473957553221)
num failures: 117
Kupiec test p-value: 1.78525505493e-09

7. Summary

In this lecture, we studied the Monte Carlo Simulation method and its application to estimate financial risk. To apply it, first, we needed to define the relationship between market factors and the instruments' returns. In such step, you must define the model which maps the market factors' values to the instruments' values: in our use case, we used a linear regression function for building our model. Next, we also had to find the parameters of our model, which are the weights of the factors we considered. Then, we had to study the distribution of each market factor. A good way to do that is using Kernel density estimation to smooth the distribution and plot it. Depending on the shape of each figure, we had to guess the best fit distribution for each factor: in our use case, we used a very simple approach, and decided that our smoothed distributions all looked normal distributions.

Then, the idea of Monte Carlo simulation was to generate many possible values for each factor and calculate the corresponding outcomes by a well-defined model in each trial. After many trials, we were able to calculate VaR from the sequences of outcome's values. When the number of trials is large enough, the VaR converges to reasonable values, that we could validate using well-known statistical hypothesis.

References

- The example in section 2 is inspired from this article.
- Backtesting Value-at-Risk models (Kansantaloustiede, 2009) - (A good reference to study Backtesting).

```
In [ ]:
```