Big Data Analytics

Chapter 8: Financial Risk Analysis via Monte Carlo Simulations

Following: [3] "**Advanced Analytics with Spark**", Chapter 9

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

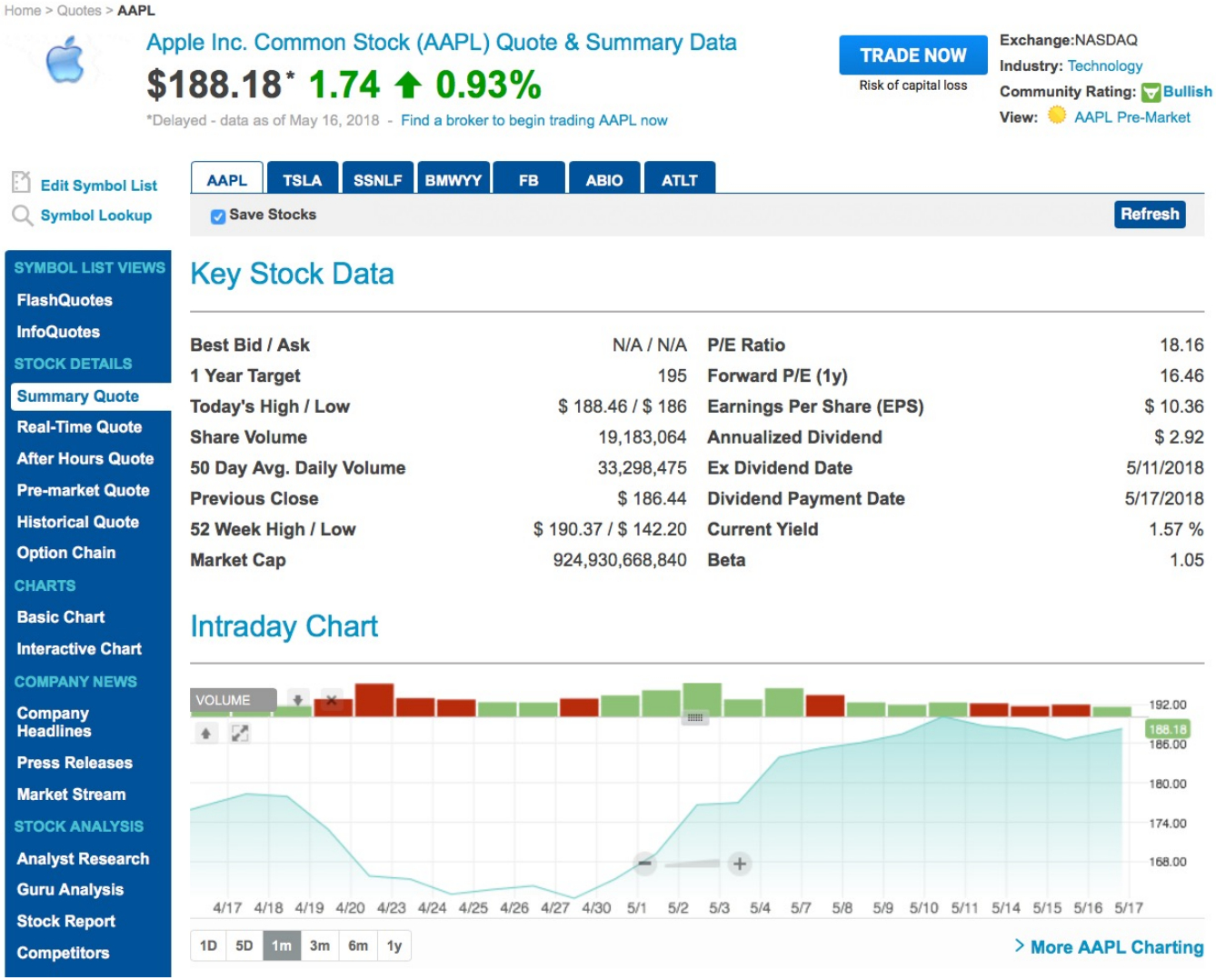
University of Luxembourg

7 Steps to Estimate the "Value-at-Risk" from Market Data

The **Value-at-Risk** is defined as the *5% quantile of worst returns* of a given portfolio (of stocks or other kinds of shares) over a *fixed period of time*.

1. Fix a **time period** (e.g. "2 weeks") over which to measure stock returns and market factors
2. Define **stock returns** r_i of given portfolio over a sliding window of market data periods
3. Similarly define **factor returns** f_j of given market factors over a sliding window of market data periods
4. Train a **linear-regression model** to predict returns r_i with respect to market factors f_j from past market data
5. Refine the linear-regression model by computing **pairwise correlations** between market factors f_j from past market data via a multivariate normal distribution
6. **Sample** returns r_i under all combinations of correlated market factors f_j
7. **VaR** then is the 5% quantile of the resulting distribution of returns r_i

Example: NASDAQ Apple (AAPL) Stocks



Source: https://www.nasdaq.com/symbol/aapl

Stock Markets & Returns of Investments

Consider a typical **stock-market setting**:

- Every **stock price** may in- or decrease over a **given period of time** (e.g., 1 week or 1 month) in reaction to a given set of **market conditions**.
- The **return** r_i of a stock i denotes the relative difference between its closing and opening price under the observed time period:

$$r_i = \frac{closingPrice_i - openingPrice_i}{closingPrice_i}$$

In addition to the stocks, we thus are also interested in a set of market conditions that analysts may have identified as **good indicators** or **influential factors** for the development of the stock prices. Also these will be measured in terms of returns.

Examples of market conditions/factors:

- S&P 500 drops by 5%
- US GDP increases by 0.2%
- Gold price increases by 1%

In the following, we consider only **large stock indexes** as such market conditions…

A Linear Regression Model for Stocks and Factors

Next, we may aim to estimate the **impact of each market condition** (then called "**factor**") on stock i from the developments of the stock under a set of previously observed market conditions:

$$r_i = c_i + \sum_{j=1}^{m} w_{i,j} \, f_j$$

The above formula denotes a simple form of **linear regression**:

- r_i are the **observed return values** (i.e., the "labels") obtained from the previous development of stock i.
- f_j are the so-called **features** (e.g., the observed factor return, squared factor return, etc.) derived from the previous development of the factors.
- c_i and $w_{i,j}$ are the **parameters** of the model which we wish to learn.

Note:

- To be able to compare the return of the stock r_i and the impact of each feature f_j on it, we need to ensure that the observations of both the stocks and the factors are perfectly temporally aligned.

How Can We Estimate "Financial Risk" ?

Large customers (such as banks) usually have an entire **portfolio** (consisting of multiple stocks and other investments) that they manage.

- If one stock goes up another one may go down, such that the portfolio will change its return value according to the sum or average of the returns' changes.

The **Value-at-Risk** (VaR) is a statistic that is often applied in Finance. It reflects *how much aggregated return* a given portfolio of stocks likely loses over a given period of time by considering all possible market conditions.

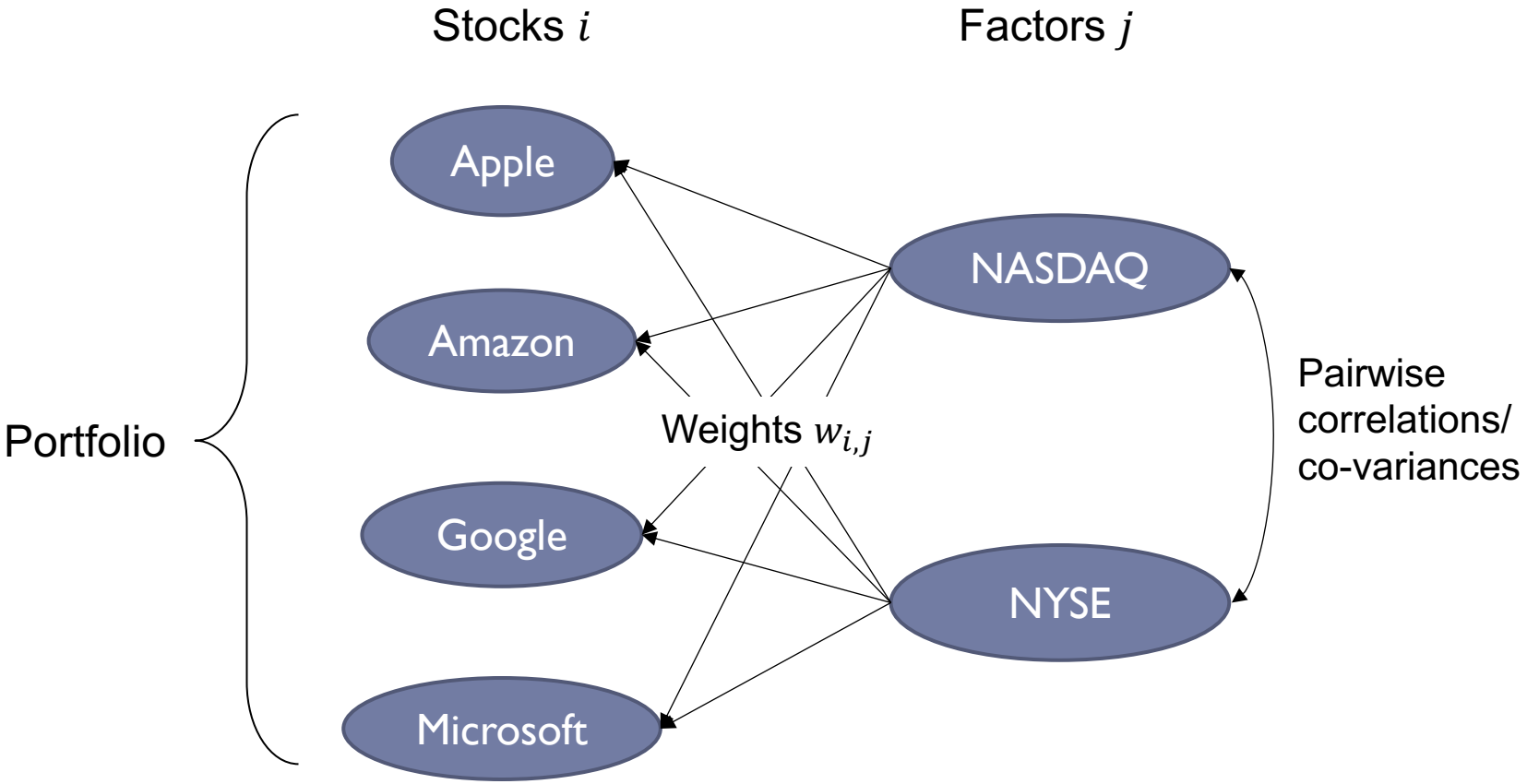
Example:

- A VaR of 1 million dollars at a 5%-quantile of the worst estimated returns over a 2-week period indicates that the entire portfolio has a 5% chance of losing more than 1 million dollars within 2 weeks.

Notes:

- Stocks typically do not react independently to changes in the market conditions, since a change in each market condition may affect multiple stocks.
- Even the market conditions may not develop independently of each other (see all the major crashes in the past).
- The number of combinations of possible market conditions is typically too large to be searched exhaustively to calculate the VaR directly.

Stocks vs. Factors



- A portfolio is a collection of stocks we may wish to invest in.
- Various market factors have an impact on each stock individually.
- Market factors are generally not independent (captured as pairwise correlations).

Time-Series Data for Stocks & Factors

Both stocks and factors are represented as **time-series data** (consisting of at least a *time-stamp* and a *measureable unit*, e.g., the market value of the stock or an entire index) at that time-stamp.

```
Date,       Open,  High,  Low,   Close, Volume
31-Dec-13, 79.17, 80.18, 79.14, 80.15, 55819372
30-Dec-13, 79.64, 80.01, 78.90, 79.22, 63407722
27-Dec-13, 80.55, 80.63, 79.93, 80.01, 56471317
26-Dec-13, 81.16, 81.36, 80.48, 80.56, 51002035
24-Dec-13, 81.41, 81.70, 80.86, 81.10, 41888735
23-Dec-13, 81.14, 81.53, 80.39, 81.44, 125326831
20-Dec-13, 77.91, 78.80, 77.83, 78.43, 109103435
....
```

$$r_i = \frac{80.01 - 78.43}{80.01} = 0.02$$

(for a 1-week period)

(The above CSV format is the one that was used by Google Finance to export portfolios until May 2018; we will extract only the Date and Close fields.)

Monte Carlo Simulations

The previous regression formula serves two purposes:

1. From the historical observations of stocks and factors, we can calculate the returns r_i and the features f_j of the portfolio we are interested in. We can use these to **train the parameters** c_i and $w_{i,j}$ of the linear-regression model.

2. To calculate the actual VaR, we feed the same regression formula (now with the learned parameters) with a large number of samples that **simulate all possible market conditions** to predict the estimated returns under these conditions.
   - From these samples, we obtain a distribution of the aggregated (e.g., using *sum* or *avg*) returns r_i of all stocks in our portfolio. The 5%-quantile of this distribution then is the VaR.
   - This repeated form of sampling is known as a **Monte Carlo Simulation**.

The Multivariate Normal Distribution

In principle, we could simply generate the samples of all market conditions that go into our predictions independently of each other (e.g., "*S&P 500 drops by 5% and Dow Jones increases by 10%*" could be one such sample).

But this would violate the assumptions we made earlier...

To consider correlations among market conditions, we assume that the factors derived from these market conditions follow a **multivariate normal distribution,** and we **sample the factors** according to this distribution.

- That is, we assume the factors $X = (x_1, \ldots, x_k)$ to follow $X \sim N(\mu, \Sigma)$ with **means vector** μ, **pairwise co-variance matrix** Σ and the following **probability density function**:

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

- Both μ and Σ can be estimated from our observations of the previous market conditions as well.

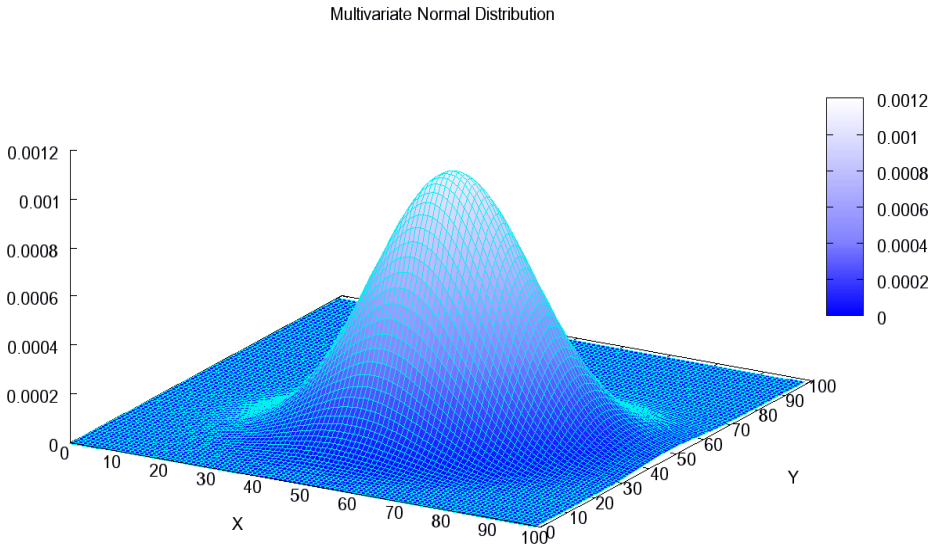


Image source: Wikipedia

Datasets for Stocks and Factors

As mentioned earlier, both stocks and factors are represented as time-series data in the Google Finance format.

- The stock files in Moodle capture the stock prices of **Amazon** (AMZN.csv), **Apple** (AAPL.csv), **Google** (GOOGL.csv) and **Microsoft** (MSFT.csv) between 1-Jan-2000 and 31-Dec-2013.
- The factor files contain the **iShares 20+ Year Treasury Bond ETF** (NASDAQ-TLT), the **iShares U.S. Credit Bond ETF** (NYSEARCA-CRED.csv) and the **SPDR Gold Shares** (NYSEARCA-GLD.csv) from a similar time range.
- Both the stock and factors files together merely consist of about **20,000 lines of CSV data**…

Parsing the Time-Series Files (I)

- We start by defining the usual parsing and cleaning functions in Scala. Note that, since the input files are not too large, we may just start with a regular **file-based API** in Scala rather than by creating an RDD.
- We will treat both stock and factor files exactly analogously in the following steps.

```
import java.io.File
import java.time.LocalDate
import java.time.format.DateTimeFormatter

// reads a stock history in the Google Finance time-series format
def readGoogleHistory(file: File): Array[(LocalDate, Double)] = {
  val formatter = DateTimeFormatter.ofPattern("d-MMM-yy")
  val lines = scala.io.Source.fromFile(file).getLines().toSeq
  lines.tail.map { line =>
    val cols = line.split(',')
    val date = LocalDate.parse(cols(0), formatter)
    val value = cols(4).toDouble
    (date, value)
  }.reverse.toArray
}
```

Parsing the Time-Series Files (II)

- We read the files from the stocks directory one-by-one, thereby catching possible exceptions:

```
val stocksDir = new File("./stocks/")
val files = stocksDir.listFiles()
val allStocks = files.iterator.flatMap { file =>
  try {
    Some(readGoogleHistory(file))
  } catch {
    case e: Exception => None
  }
}
```

- And we finally filter out only time-series of stocks that capture more than 5 years of trading days:

```
val rawStocks = allStocks.filter(_.size >= 260 * 5 + 10)
```

Trimming the Time-Series Data

- Next, the time-series data of each input file is trimmed to also match a given range of dates:

```
def trimToRegion(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate): Array[(LocalDate, Double)] = {
  var trimmed = history.dropWhile(_._1.isBefore(start)).
    takeWhile(x => x._1.isBefore(end) || x._1.isEqual(end))
  if (trimmed.head._1 != start) {
    trimmed = Array((start, trimmed.head._2)) ++ trimmed
  }
  if (trimmed.last._1 != end) {
    trimmed = trimmed ++ Array((end, trimmed.last._2)) }
  trimmed }
```

- And here is the actual call of the function using a `map` transformation:

```
val start = LocalDate.of(2009, 10, 23)
val end = LocalDate.of(2014, 10, 23)
val trimmedStocks = rawStocks.map(trimToRegion(_, start, end))
```

Auto-Completing the Time-Series Data

- To compare the returns of stocks with their factors, both types of time-series have to be perfectly aligned. We achieve this by imputing missing values into the time-series data from the input files:

```
def fillInHistory(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate): Array[(LocalDate, Double)] = {
  var cur = history
  val filled = new ArrayBuffer[(LocalDate, Double)]()
  var curDate = start
  while (curDate.isBefore(end)) {
    if (cur.tail.nonEmpty && cur.tail.head._1 == curDate)
      cur = cur.tail
    filled += ((curDate, cur.head._2))
    curDate = curDate.plusDays(1)
    if (curDate.getDayOfWeek.getValue > 5) // skipping weekends!
      curDate = curDate.plusDays(2) }
  filled.toArray }
```

- And create the final stocks time-series:

```
val stocks = trimmedStocks.map(fillInHistory(_, start, end))
```

Using Sliding Windows to Compute Bi-Weekly Returns

- We will fix our analysis to **bi-weekly returns** for all of the following steps.
- To do so, we implement a **sliding window** that computes (overlapping) bi-weekly returns over each of the stocks' and factors' time-series:

```
def twoWeekReturns(history: Array[(LocalDate, Double)]):
      Array[Double] = {
  var i = 0
  history.sliding(10).map { window =>
    val next = window.last._2
    val prev = window.head._2
    i += 1
    (next - prev) / prev
  }.toArray
}

val stockReturns = stocks.map(twoWeekReturns).toArray.toSeq
val factorReturns = factors.map(twoWeekReturns).toArray.toSeq
```

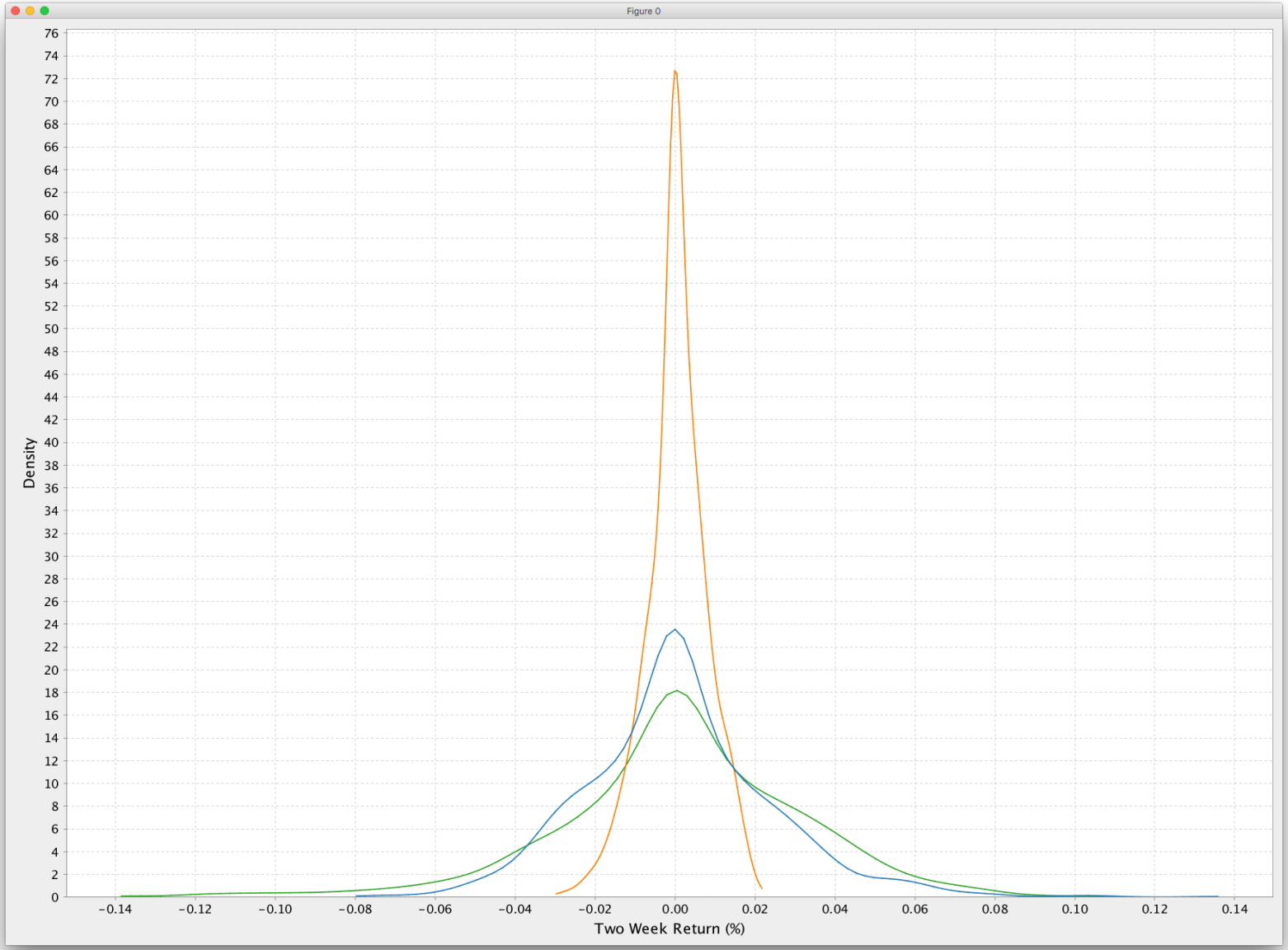
- Note that a "history" here refers to the observed time-series data of a stock or factor file. The `sliding` operator creates an overlapping sliding window over these. Compare: `List(1,2,3,4,5).sliding(3).foreach(println)`

Plotting the Distributions of Factor Returns

- The `breeze` and `jfree` libraries offer a variety of options to draw plots in Scala.
- Note that the `KernelDensity` class of Spark is used here to smooth the curve given by the raw samples of factor returns.

```
def plotDistributions(sampleSets: Seq[Array[Double]]): Figure = {
  val f = Figure()
  val p = f.subplot(0)
  …
  for (samples <- sampleSets) {
    val min = samples.min
    val max = samples.max
    val stddev = new StatCounter(samples).stdev
    val bandwidth = 1.06 * stddev * math.pow(samples.size, -.2)
    val domain = Range.Double(min, max, (max - min) / 100).toList.toArray
    val kd = new KernelDensity().
      setSample(sc.parallelize(samples)).setBandwidth(bandwidth)
    val densities = kd.estimate(domain)
    p += plot(domain, densities) }
  f }
plotDistributions(factorReturns)
```

Distributions of Factor Returns



Computing the Factor Means and Co-Variances

We now pursue in three steps to **fit the parameters** of our multivariate-normal-distribution from the factor returns:

1. We compute the means of the distributions directly from the `factorReturns`.
2. We compute a `factorMatrix` object that captures the three arrays with the factor returns in (transposed) matrix form.
3. From the `factorMatrix`, we compute also the pairwise co-variances of the factor returns.

```
import org.apache.commons.math3.stat.correlation.Covariance

def factorMatrix(histories: Seq[Array[Double]]): Array[Array[Double]] = {
   val mat = new Array[Array[Double]](histories.head.length)
   for (i <- histories.head.indices) {
     mat(i) = histories.map(_(i)).toArray
   }
   mat }

val factorMeans = factorReturns.map(factor => factor.sum / factor.size).toArray
val factorMat = factorMatrix(factorReturns)
val factorCov = new Covariance(factorMat).getCovarianceMatrix().getData()
```

"Featurizing" the Factor Returns

- To turn the factor returns we calculated from the time-series data into the actual features f_j, which we will feed into our linear-regression model, we include also the *squares* and the *square roots* of the factor returns into the final features.

```
// include additional feature functions such as square, square root, etc.
def featurize(factorReturns: Array[Double]): Array[Double] = {
  val squaredReturns = factorReturns.map(x => math.signum(x) * x * x)
  val squareRootedReturns = factorReturns.map(x => math.signum(x) *
     math.sqrt(math.abs(x)))
  squaredReturns ++ squareRootedReturns ++ factorReturns
}
```

- And we invoke the learning step for the linear-regression model over both the observed stock and feature returns:

```
// learn the parameters of the linear-regression model
val factorFeatures = factorMat.map(featurize)
val factorWeights = computeFactorWeights(stockReturns, factorFeatures)
```

Training the Linear-Regression Model

We pursue in a similar manner to also **fit the parameters** of our linear-regression model to later predict the stock returns from the simulated factor returns:

1. We compute the means of the distributions directly from the `factorReturns`.
2. We compute a `factorMatrix` object that captures the three arrays with the factor returns in (transposed) matrix form.
3. From the `factorMatrix`, we compute also the pairwise co-variances of the factor returns.

```
import org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression
def linearModel(instrument: Array[Double], factorMatrix: Array[Array[Double]]):
     OLSMultipleLinearRegression = {
  val regression = new OLSMultipleLinearRegression()
  regression.newSampleData(instrument, factorMatrix)
  regression
}
def computeFactorWeights(
  stocksReturns:  Seq[Array[Double]],
  factorFeatures: Array[Array[Double]]): Array[Array[Double]] = {
  stocksReturns.map(linearModel(_,
     factorFeatures)).map(_.estimateRegressionParameters()).toArray }
```

Parallel Sampling

- To facilitate a form of **parallel sampling in Spark**, we distribute both the stock and factor returns by turning them into an RDD (specifically: a `Dataset`).
- Parallel sampling requires the repeated generation of random numbers by some form of random-number generator (here: a `MersenneTwister`).
- To avoid a repeated sampling of the same values, we initialize a new random-number generator for each partition of the RDD with a different seed value.

```
val numTrials = 1000000
val parallelism = 100
val baseSeed = 1001L

import org.apache.spark.sql.Dataset
import org.apache.spark.sql.SQLContext

val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val seeds = (baseSeed until baseSeed + parallelism)
val seedsDS = seeds.toDS().repartition(parallelism)
```

Spark's Dataset API

- Spark 1.6 introduced the `Dataset` API as an extension of the `DataFrame` API.
- Unlike a `DataFrame`, a `Dataset[T]` is a strongly typed data structure and thereby allows for better compile-time type safety than a `DataFrame` (which, internally, is actually an alias for a `Dataset[Row]`, where the type of `Row` however need not explicitly be specified by the programmer).
- Both are backed up internally by an `RDD` and thus fully support all `RDD` transformations and actions.
- By importing `spark.implicits._` one can cast dynamically among an `RDD`, `Dataset` and `DataFrame` using `toDS()` and `toDF()`, respectively.
- By accessing `Dataset.agg`, one has access to a number of SQL-style aggregation functions.

  Example:

```
samples.agg(functions.min($"value"), functions.max($"value"))
```

- By accessing `Dataset.stat`, one has access to a number of statistical functions.

  Example:

```
samples.approxQuantile("value", Array(0.05), 0.0))
```

Running the Sampler

- The `trialReturns` function computes a number of samples (called "trials") from the assumed multivariate normal distribution which is initialized with the same parameters for all partitions (but each with a different random-number generator).

```
def trialReturns(
     seed:       Long,
     numTrials:  Int,
     instruments:Seq[Array[Double]],
     factorMeans:Array[Double],
     factorCov:  Array[Array[Double]]): Seq[Double] = {
    val rand = new MersenneTwister(seed)
    val mvn = new MultivariateNormalDistribution(rand, factorMeans, factorCov)
    val trialReturns = new Array[Double](numTrials)
    for (i <- 0 until numTrials) {
      val trialFactorReturns = multivariateNormal.sample()
      val trialFeatures = featurize(trialFactorReturns)
      trialReturns(i) = trialReturn(trialFeatures, instruments) }
    trialReturns }

val trialsDS = seedDS.flatMap(trialReturns(_, numTrials / parallelism, factorWeights,
                                           factorMeans, factorCov))
```

Aggregating the Trial Returns

- We are now ready to **predict the average stock return** of each stock in our portfolio for a sampled set of factor returns: $r_{avg} = \frac{1}{n}\sum_{i=1}^{n}(c_i + \sum_{j=1}^{m} w_{i,j}\, f_j)$
- An "instrument" here denotes the array of weights $w_{i,j}$ learned for each stock i.
- A "trial" is the array of features f_j computed from the sampled factor returns; c_i is set to the first weight $w_{i,0}$ of each instrument.

```
def instrumentTrialReturn(instrument: Array[Double], trial: Array[Double]):
     Double = {
    var instrumentTrialReturn = instrument(0)
    var i = 0
    while (i < trial.length) {
      instrumentTrialReturn += trial(i) * instrument(i + 1)
      i += 1 }
  instrumentTrialReturn }

def trialReturn(trial: Array[Double], instruments: Seq[Array[Double]]): Double = {
  var totalReturn = 0.0
  for (instrument <- instruments) {
    totalReturn += instrumentTrialReturn(instrument, trial) }
  totalReturn / instruments.size }
```

Calculating the VaR

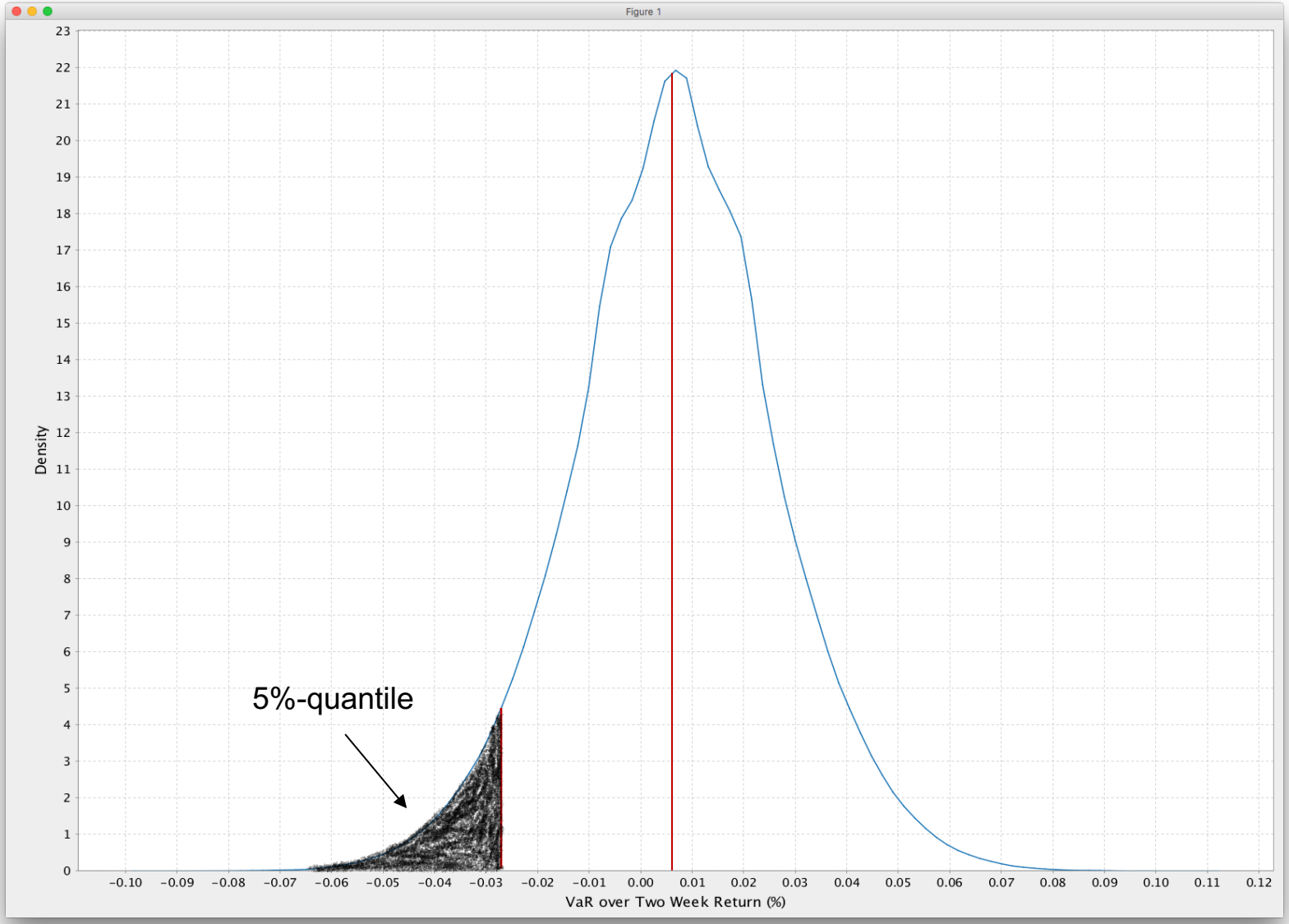
- The VaR now confirms to the **5%-quantile of the distribution of the average stock returns** that we estimated by the trials from the previous step.
- This can also very conveniently be obtained from the `Dataset` API as follows:

```
def fivePercentVaR(trialsDS: Dataset[Double]): Double = {
  val quantiles = trialsDS.stat.approxQuantile("value", Array(0.05), 0.0)
  quantiles.head
}

val valueAtRisk = fivePercentVaR(trialsDS)
```

- About 200 lines of Spark/Scala code and ca. 100,000,000 Monte Carlo samples later…

Plotting the VaR for our Portfolio (AMZN/AAPL/GOOGL/MSFT)



The 5%-quantile is at about -0.026.

Conditional Value-at-Risk

- Similarly, the **Conditional Value-at-Risk** (CVaR) is defined as the *average* over the 5%-quantile of the worst returns:

```
def fivePercentCVaR(trialsDS: Dataset[Double]): Double = {
  val topLosses = trialsDS.orderBy("value").limit(math.max(trialsDS.count().
     toInt / 20, 1))
  topLosses.agg("value" -> "avg").first()(0).asInstanceOf[Double]
}

val conditionalValueAtRisk = fivePercentCVaR(trialsDS)
```

Summary

- VaR and CVaR are only two of the commonly applied statistics to assess financial risk.
- VaR has also been criticized in the following ways:
  - Rare events, such as major crashes, still cannot be predicted by VaR alone.
  - VaR may give overly high confidence in high-risk investments and thus may lead to excessive risk-taking by investors.
- Both of the measures however demonstrate an interesting **combination of machine-learning techniques** (learning factors and weights from historical data) **and more traditional statistical approaches** (computing co-variances and sampling from an assumed distribution) to simulate future events.
- **Distributed sampling** is "naturally" facilitated here by Spark's `RDD` and `Dataset` APIs.


Data
Analytics
Big
Hadoop
Spark
Prediction
Distribution
Matrix
Scalability
Predictive
Sparse
Scala
Trees
GraphX
Rank
GFS
Purity
Systems
JSON
XML
MLlib
Sampling
Variable
Random
GeoJSON
Recommender
Analytics
Science
Ham
Confusion
Clustering
Reduce
Parallelism
Language
RDD
Streaming
Multivariate
AudioScrobbler
Twitter
MeSH
Recall
Broadcast
Precision
HDFS
Visualization
Decision
Validation
Triangles
Statistical
Components
Value
Normal
Classification
Outlier
Combiner
Action
Gradient
ALS
CoverType
Partitioner
GPS
Cosine
Java
Eigenvalue
Cross
Vector
Entropy
Semantic
HITS
Efficiency
Map
Linear
Networks
Testsal
WordCount
Decomposition
Trips
Connected
API
Natural
Forests
Transformation
Cliques
Latent
Processing
Gini
Singular
Medline
Taxi
Medical
Weather
Detection
Strongly
Risk
Financial
Eigenvector
Factorization
PageRank
Anomaly
Probability
Accumulator
Confidence
Dense
DStream
K-Means
Spam
Regression
Graphs
DataFrame
Dataset
Dataflow
Boosted
SQL
Open
NoSQL
GIS