BIG DATA ANALYTICS

Financial Risk Analysis

Balamanikandan Gopalakrishnan – Rachan Hegde

Contents

Introduction 2
  Time Series Data 2
Executive Summary 2
  Problem Statement 2
  Data Set 2
  Approach 3
    Load Data 4
    Data Analysis 6
    Data Preprocessing 8
    Prediction 9
    Implementation 9
    Summarization 24
Conclusion 25
References 25

Introduction

In this report, we will look at basics of time-series data and their characteristics. We will consider a business problem of analyzing and predicting the financial risk associated with a portfolio of stocks for a specific period in time. We will also look at the approach taken in solving this business problem using Apache Spark and supporting Big Data technologies.

Time Series Data

The basis of a time series is the repeated measurement of a parameter over time together with the times at which the measurements were made. Time series datasets are typically used in situations in which measurements, once made, are not revised or updated, but rather, where the mass of measurements accumulates, with new data added for each parameter being measured at each new time point. These characteristics of time series limit the demands we put on the technology we use to store time series and thus affect how we design that technology.

There are two major applications from time series data analysis

- **Exploratory Data Analysis** – To understand the historical trend and behavior of subject under analysis
- **Prediction and Forecasting** – To design a mathematical model based on the historical data to predict behavior of process in future

Executive Summary

Problem Statement

Identify the VAR (Value at risk) associated with a portfolio of stocks for a specific time period using the historical time series data of portfolio stocks and the external factors that influence these stocks. Also provide analysis on the top gaining and loosing stocks for the specific period and their relationship with the external factors.

Note: Value at risk is a measure of investment risk that tries to provide an estimate of maximum probable loss in value of an investment portfolio over a particular time period.

Data Set

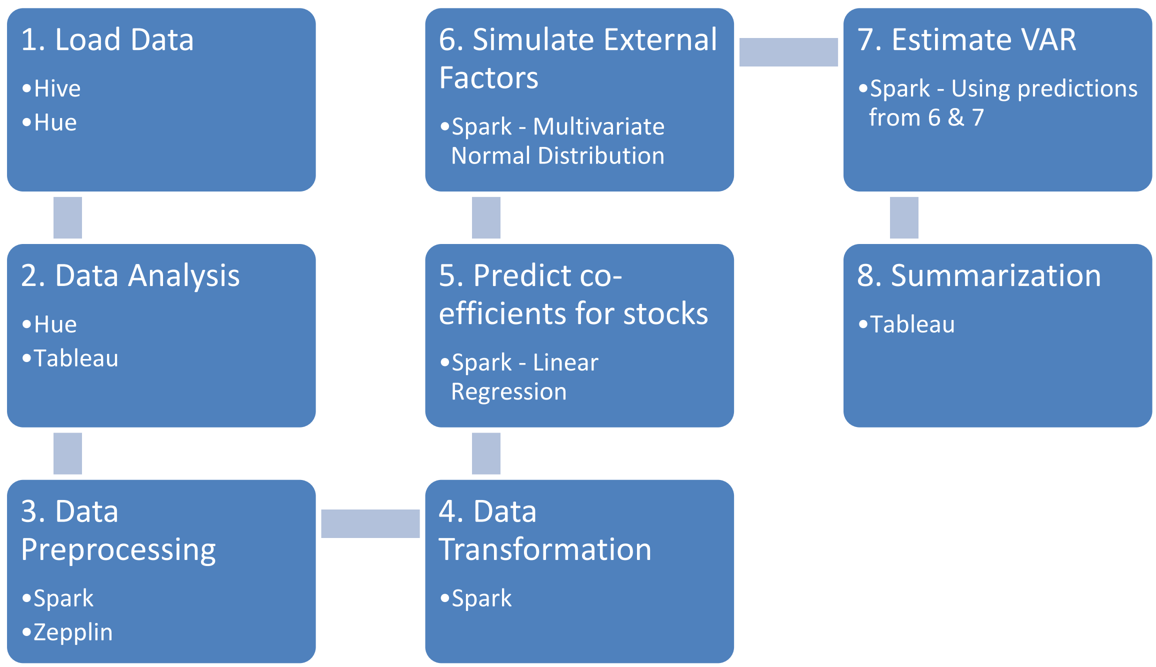
The data set consists of observations for a portfolio of ~3000 stocks from year 2000 to 2015. The observations are available on a daily basis and consists of ~9 Million records. The data set also consists of observations made on a daily basis for external factors that influence these stocks from year 2000 to 2015. Each observation consists of the following attributes

- Open, Close, High, Low, Volume and Adj. Close

Approach

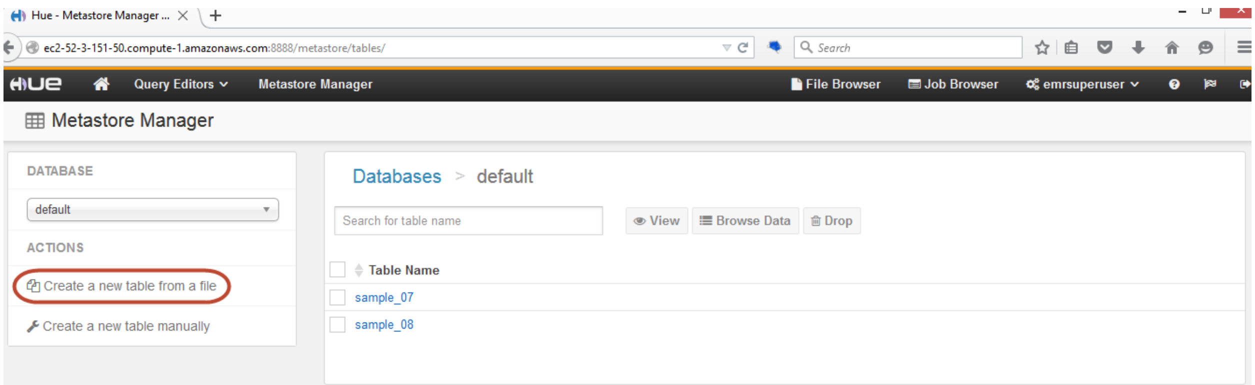
An amazon EMR instance of version 3.8 was set up along with Hive, Hue and Spark.

The following image shows the high-level overview of the approach and technologies chosen to estimate the financial risk associated with the portfolio of stocks from the data set



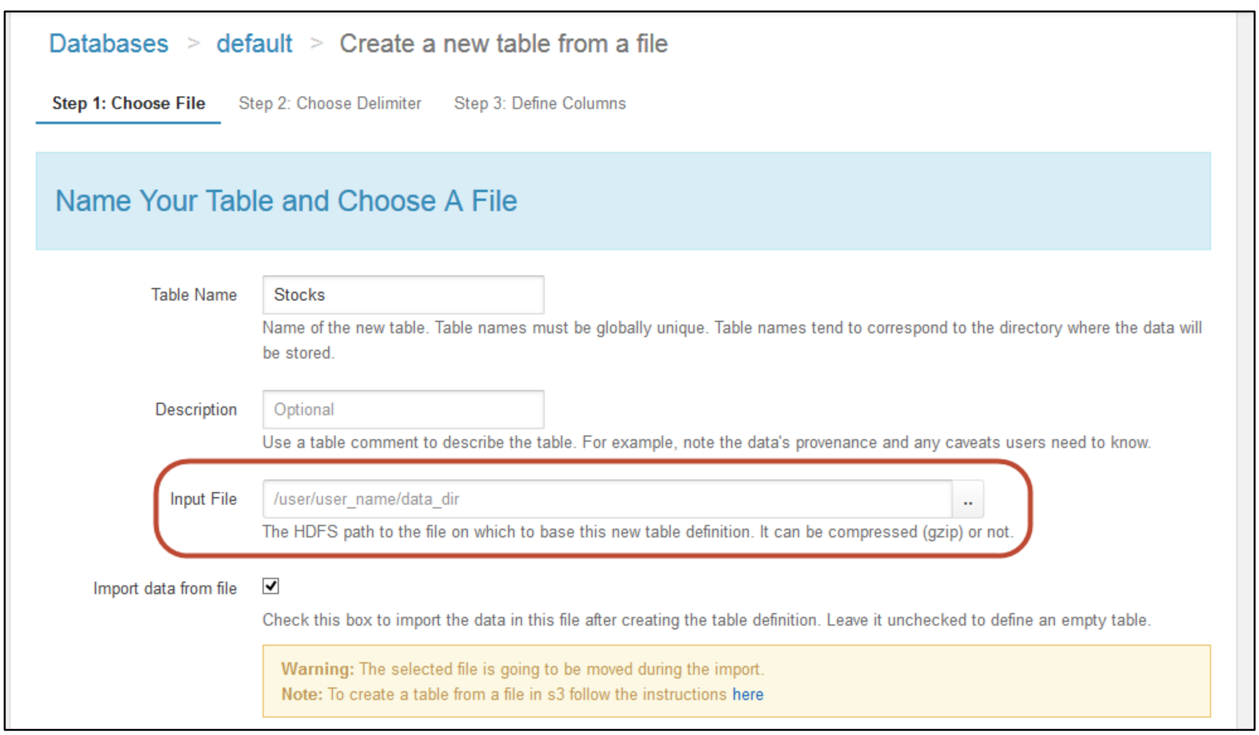
Load Data

The historical stock performance and the external factors related data was loaded into hive tables from files using Hue. A proxy management tool was set up to connect to Hue. The instructions to set up proxy management tools are provided in 'Enable web connection' section of EMR. On connecting to Hue, the users can create new table from files by using the 'Meta Manager' option

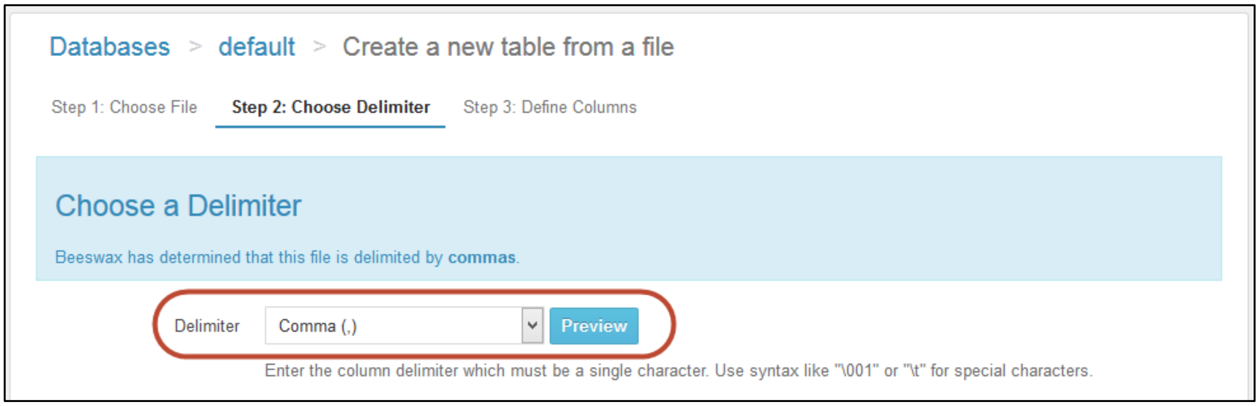


Loading data into Hive using Hue is a three step process involving

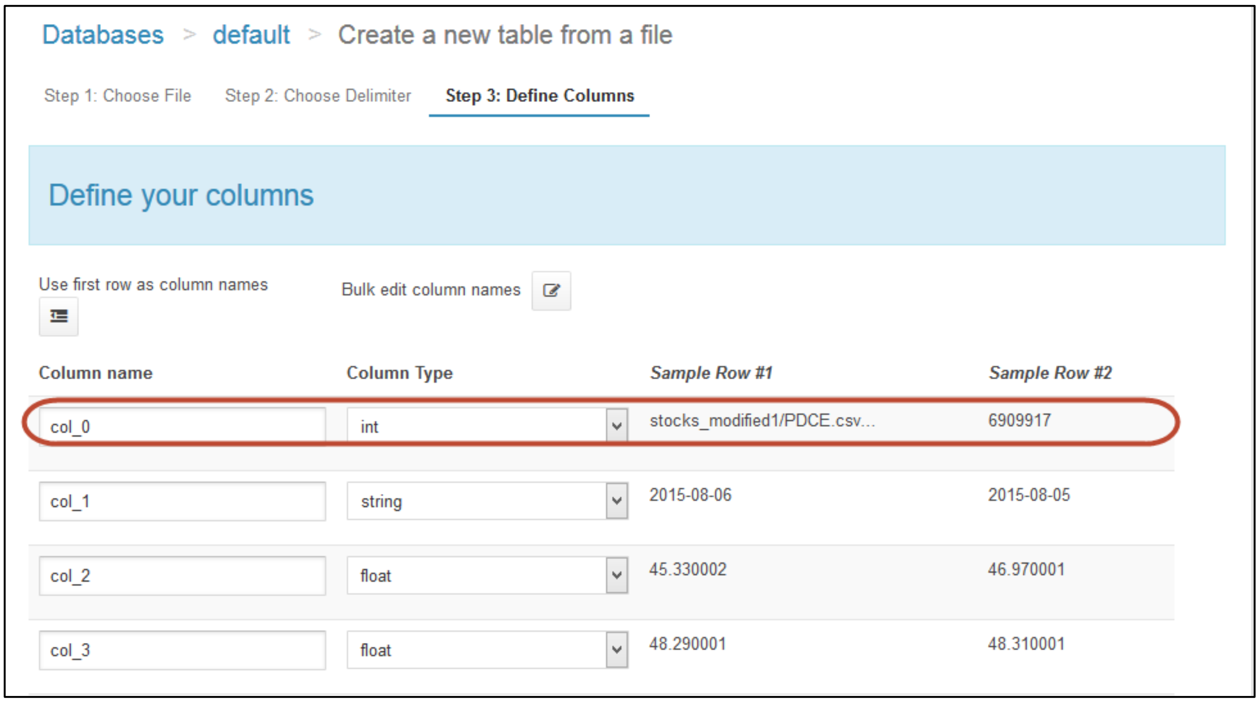
- File Selection
- Choose Delimiter
- Column Definition



Specify the input file

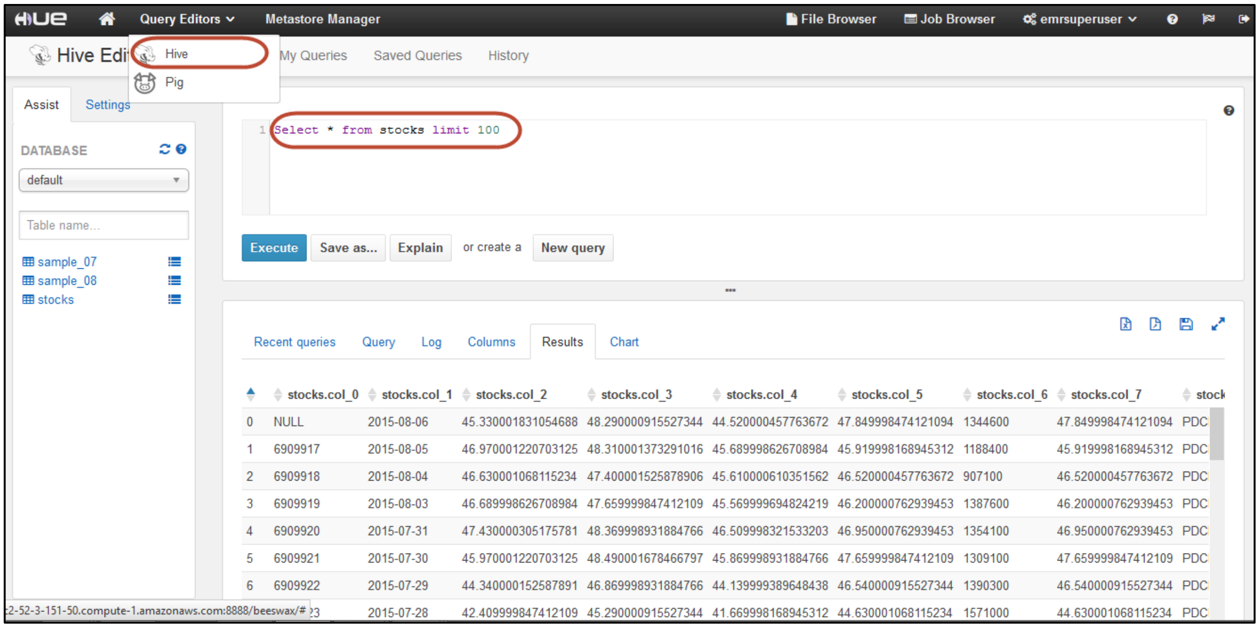


Define Delimiter



Define Column names and data types

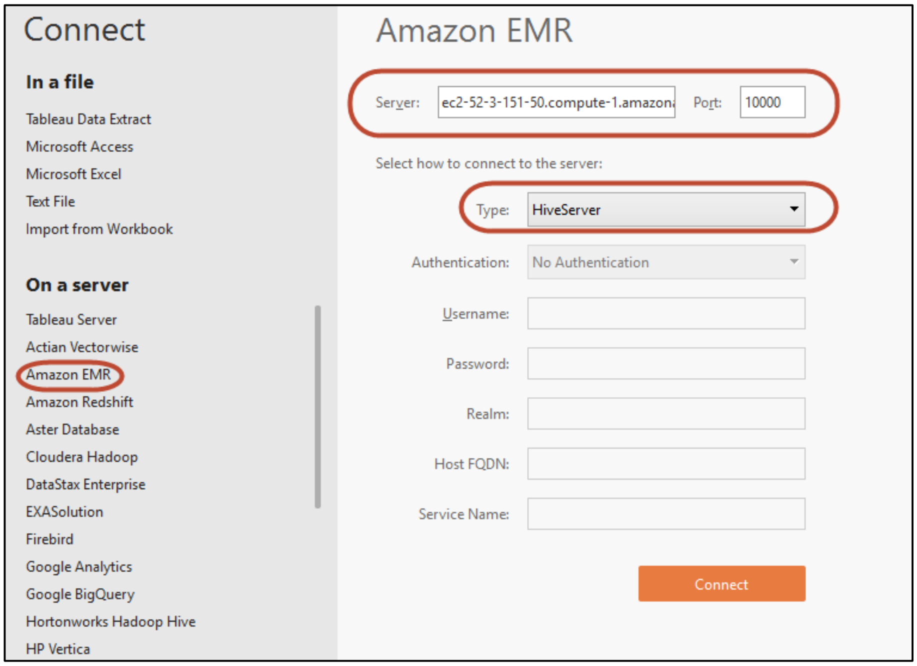
Following these steps loads the data into hive and data analysis was performed using hive editor as shown below



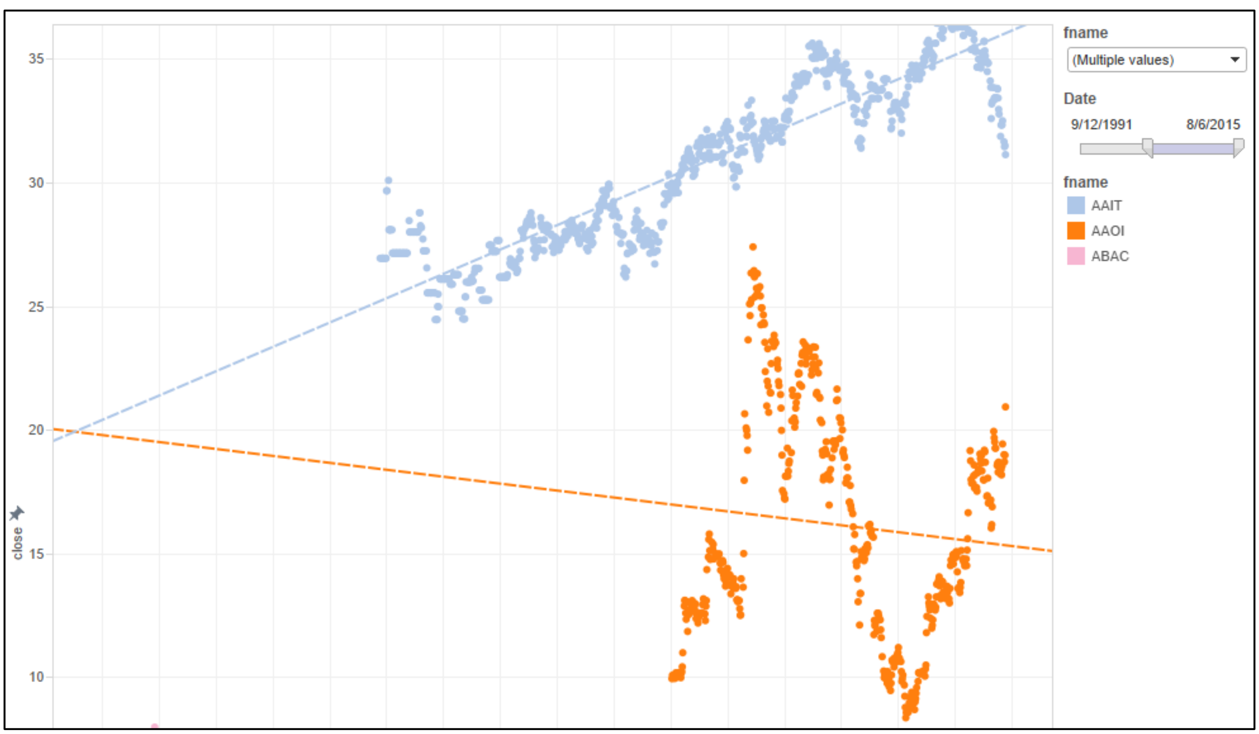
Data Analysis

After loading the data, we visualized the given time series to gain insights about the data that can be later used for prediction or forecasting or an anomalies to be considered before building our model for prediction.

Tableau supports data connections to EMR and the following image shows sample configuration screen. The port number displayed here should be configured in EC2 inbound security group settings to allow external connections



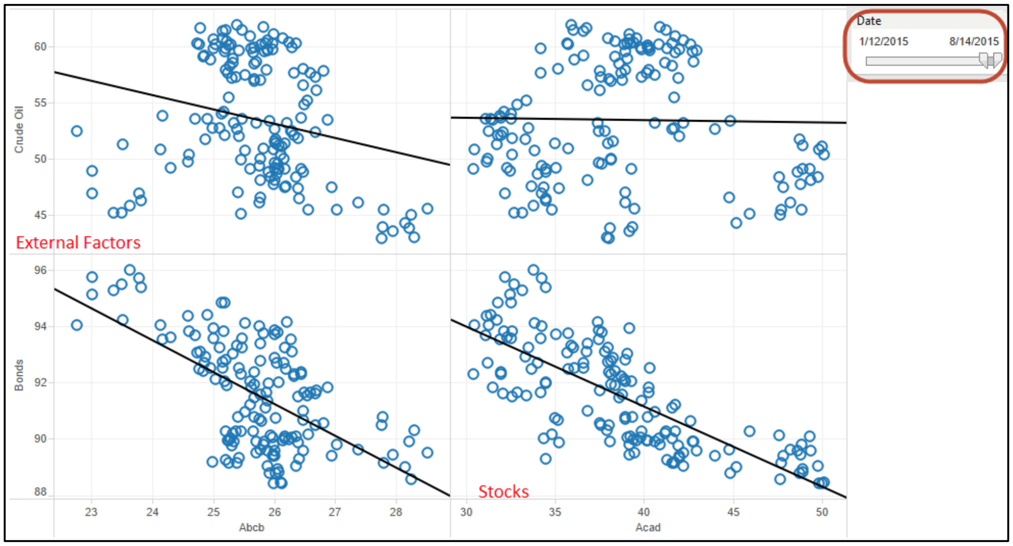
Insights Obtained



It was observed that the time series of observations available for stocks were not consistent. Also some of the some observations were missing for some of the stocks.

Analysis was also performed to identify relationship between stocks and external factors if any. It was observed that there was no direct relationship between stocks and external factors as they vary across time. However the relationship between stocks and external factors in the recent past can be used in predictions

The following images show the variations in relationship between stocks and external factors.



Data Preprocessing

Data preprocessing was performed to fit all the stocks and external factors into a similar date range for prediction. The missing data was filled with observations from nearest neighbor as latest records are more relevant in time series data that using mean.

Prediction

After processing of data, we calculate returns. A stock return is a change in the market value of stock over a particular time. A factor return is change in value of external factor over a period of time. We'll derive a set of features from transformation of factor returns.

For each instrument, we'll train a model that assigns weight to each feature. The return of each instrument is calculated as the sum of returns of market factors multiplied by the weights for that instrument. Using linear regression, we calculate the factor weights associated with each stock based on the historical data.

To model the fact that instrument returns are non-linear functions of factor returns, we include some additional features from non-linear transformations of factor returns by adding additional features for each return (Square and its square root)

With models that map factor returns to stocks, we now need a procedure to simulate market conditions by generating random factor returns. We use multivariate normal distribution to achieve the same. The Multivariate normal distribution also takes correlation information between factors into account during sampling.

We have per-stock model and a procedure using multivariate normal distribution for sampling factor returns. We now need to run trials using spark and parallelize them. In each trial, we sample factor returns, use them to predict the return of each instrument and sum all those to find the full trial loss. We run a number of these trials to achieve representative distribution.

To calculate the 5% VaR, we need to find a return such that we expect to do worse that it 5% of the time and better than it 95% of the time. With our distribution, we can accomplish this pulling the worst performing trials in the distribution. The best trial in that distribution forms the 5% VaR.

Environment Used

- Amazon EMR v3.8
    - 1 Master mx3 large instances
    - 4 Slaves mx3 large instances
- Spark 1.3.1
- Hue 0.17
- Hive
- Apache Zeppelin Bootstrapped
- Tableau

Implementation

//Imports

```
import java.io.File
import java.text.SimpleDateFormat
import scala.collection.mutable.ArrayBuffer
```

import scala.io.Source
import com.github.nscala_time.time.Imports._

//spark related imports

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
import org.apache.spark.mllib.feature.{StandardScaler,Normalizer,ChiSqSelector}
import org.apache.spark.mllib.evaluation.{MulticlassMetrics, BinaryClassificationMetrics}
import org.apache.spark.sql.{Row, SQLContext}
import sqlContext.implicits._
import org.apache.spark.rdd.PairRDDFunctions
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types.StringType
import org.apache.spark.sql.{SQLContext, DataFrame}
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.optimization.{L1Updater,SquaredL2Updater}
import
org.apache.spark.mllib.regression.{LinearRegressionWithSGD,RidgeRegressionWithSGD,LassoWithSGD}
import org.apache.commons.math3.distribution.ChiSquaredDistribution
import org.apache.commons.math3.distribution.MultivariateNormalDistribution
import org.apache.commons.math3.random.MersenneTwister
import org.apache.commons.math3.stat.correlation.Covariance
import scala.util

//Re useable Scala methods

```
def toInt(s: String): Option[Int] = {
  try {
    Some(s.toInt)
  } catch {
    case e: Exception => None
  }
}
```

// Aligns the dataset to the appropriate time period

```
def trimToRegion(history: Array[(String,DateTime, Double)], start: DateTime, end: DateTime)
  : Array[(String,DateTime, Double)] = {
    var trimmed = history.dropWhile(_._2 < start).takeWhile(_._2 <= end)
    if (trimmed.head._2 != start) {
      trimmed = Array((trimmed.head._1,start, trimmed.head._3)) ++ trimmed
    }
    if (trimmed.last._2 != end) {
      trimmed = trimmed ++ Array((trimmed.last._1,end, trimmed.last._3))
    }
    trimmed
  }
```

// Fills missing data with values from nearest neighbor's

```
def fillInHistory(history: Array[(String,DateTime, Double)], start: DateTime, end: DateTime)
  : Array[(String,DateTime, Double)] = {
    var cur = history
    val filled = new ArrayBuffer[(String,DateTime, Double)]()
    var curDate = start
    while (curDate < end) {
      if (cur.tail.nonEmpty && cur.tail.head._2 == curDate) {
        cur = cur.tail
      }

      filled += ((cur.head._1,curDate, cur.head._3))

      curDate += 1.days
      // Skip weekends
      if (curDate.dayOfWeek().get > 5) curDate += 2.days
    }
    filled.toArray
  }
```

/**
*** maps a Stock name to ID**
***/**

```
  def readStockNameMap(file: File) = {
    val lines = Source.fromFile(file).getLines().toSeq
    lines.tail.map(line => {
      val cols = line.split(',')
```

```
    val value = cols(0).toDouble
    (cols(1), value)
  }).toMap
}

/**
 * maps a ID to Stock name
 */
def readIdtoStockNameMap(file: File) = {
  val lines = Source.fromFile(file).getLines().toSeq
  lines.tail.map(line => {
    val cols = line.split(',')
    val value = cols(0).toDouble
    (value, cols(1))
  }).toMap
}

/**
 * Reads a Stock history from each file
 */
def readStockHistory(file: File): Array[(String ,DateTime, Double)] = {
  val format = new SimpleDateFormat("yyyy-MM-dd")
  val lines = Source.fromFile(file).getLines().toSeq
  val name = file.getName().split('.')
  lines.tail.map(line => {
    val cols = line.split(',')
    val date = new DateTime(format.parse(cols(0)))
    val value = cols(1).toDouble
    (name(0) ,date, value)
  }).reverse.toArray
}

/**
 * Reads a Factor history from each file
 */
def readFactorHistory(file: File): Array[(DateTime, Double)] = {
  val format = new SimpleDateFormat("yyyy-MM-dd")
  val lines = Source.fromFile(file).getLines().toSeq
  lines.tail.map(line => {
    val cols = line.split(',')
    val date = new DateTime(format.parse(cols(0)))
    val value = cols(1).toDouble
```

```
    (date, value)
  }).reverse.toArray
 }

/**
  * Reads a all the Stock histories
 */
def readAllStockHistories(dir: File): Seq[Array[(String ,DateTime, Double)]] = {
  val files = dir.listFiles()
  files.flatMap(file => {
    try {
      Some(readStockHistory(file))
    } catch {
      case e: Exception => None
    }
  })
 }

// Estimates the value return for each stock based on sliding window

def valueReturns(history: Array[(String,DateTime, Double)]): Array[(String,Double)] = {
  history.sliding(slidingWindow.getOrElse(0)).map { window =>
    val next = window.last._3
    val prev = window.head._3
    (window.head._1,((next - prev) / prev))
  }.toArray
 }

// Merges all the factor returns into an array
def factorMatrix(histories: Seq[Array[Double]]): Array[Array[Double]] = {
  val mat = new Array[Array[Double]](histories.head.length)
  for (i <- 0 until histories.head.length) {
    mat(i) = histories.map(_(i)).toArray
  }
  mat
 }


// Introduces nonlinear parameter for each factor
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
 }
```

// Removes stock name tagged to each stock return

```
def dataMatrix(histories: Array[(String, Double)]): Array[Double] = {
  val mat = new Array[Double](histories.length)
  //test = test + 1
 //val fileID = test
  var i = 0
  for(y <- histories){
     if (i < histories.length){
        mat(i) = y._2.toDouble
        i= (i + 1)
     }
   }
  mat:+stockNameMap.get(histories(0)._1).get
 }
```

//combines each instrument returns to the factor returns

```
def dataModel(instrument: Array[Double], factorMatrix: Array[Array[Double]])= {
   val mat = new Array[Array[Double]](instrument.length-1)
 for (i <- 0 until instrument.length-1 ) {
     if (i < instrument.length-1){
        mat(i) = factorMatrix(i):+instrument(i):+instrument(instrument.length-1)
     }

  }
  mat
 }
```

//computing instrument models

```
def computeFactorWeights(dataset:Array[Array[Double]])= {
  val stockID = dataset(0)(dataset(0).length-1)
   val data2 = dataset.map{ line =>
  LabeledPoint(line(line.length-2), Vectors.dense(line.dropRight(2)))
  }

  val data2RDD = sc.parallelize(data2)
```

// ALGORITHMS

```
        if (algoNAme == "LinearRegressionWithSGD_L0"){
```

```
  // LR default L0
   var lrAlg = new LinearRegressionWithSGD()
     lrAlg.optimizer.setNumIterations(100).setStepSize(0.001)
     lrAlg.setIntercept(true)
      val model = lrAlg.run(data2RDD)
 var factorWeights = new Array[Double](model.weights.toArray.length)
 var i = 0
 for (wts <- model.weights.toArray){

    if (i < model.weights.toArray.length){
       factorWeights(i) = wts
       i = i + 1
       //println(wts)
       //println("######################")
    }

 }

 factorWeights = factorWeights:+model.intercept
 factorWeights:+stockID
 }
 if (algoNAme == "LinearRegressionWithSGD_L1"){

  // LR default L1
var  lrAlg = new LinearRegressionWithSGD()
   lrAlg.optimizer.setNumIterations(100).setUpdater(new L1Updater).setStepSize(0.001)
   lrAlg.setIntercept(true)
    val model = lrAlg.run(data2RDD)
 var factorWeights = new Array[Double](model.weights.toArray.length)
 var i = 0
 for (wts <- model.weights.toArray){

    if (i < model.weights.toArray.length){
       factorWeights(i) = wts
       i = i + 1
       //println(wts)
       //println("######################")
    }

 }

 factorWeights = factorWeights:+model.intercept
```

```
factorWeights:+stockID
}
if (algoNAme == "LinearRegressionWithSGD_L2"){
        // LR default L2
     var lrAlg = new LinearRegressionWithSGD()
       lrAlg.optimizer.setNumIterations(100).setUpdater(new
SquaredL2Updater).setStepSize(0.001)
 lrAlg.setIntercept(true)

  val model = lrAlg.run(data2RDD)

var factorWeights = new Array[Double](model.weights.toArray.length)

var i = 0
for (wts <- model.weights.toArray){

  if (i < model.weights.toArray.length){
    factorWeights(i) = wts
    i = i + 1
    //println(wts)
    //println("######################")
  }

}
factorWeights = factorWeights:+model.intercept
factorWeights:+stockID
}


if (algoNAme == "RidgeRegressionWithSGD"){
 // RidgeRegressionWithSGD default L2
var lrAlg = new RidgeRegressionWithSGD()
  lrAlg.optimizer.setNumIterations(100).setStepSize(0.001)
  lrAlg.setIntercept(true)
   val model = lrAlg.run(data2RDD)

var factorWeights = new Array[Double](model.weights.toArray.length)

var i = 0
for (wts <- model.weights.toArray){

  if (i < model.weights.toArray.length){
```

```
                factorWeights(i) = wts
                i = i + 1
                //println(wts)
                //println("######################")
              }
          }
          factorWeights = factorWeights:+model.intercept
          factorWeights:+stockID
          }

          if (algoNAme == "LassoWithSGD"){
            // LR default L2
          var  lrAlg = new LassoWithSGD()
                  lrAlg.optimizer.setNumIterations(100).setStepSize(0.001)
            lrAlg.setIntercept(true)
            val model = lrAlg.run(data2RDD)
          var factorWeights = new Array[Double](model.weights.toArray.length)
          var i = 0
          for (wts <- model.weights.toArray){

            if (i < model.weights.toArray.length){
                factorWeights(i) = wts
                i = i + 1
                //println(wts)
                //println("######################")
            }
          }
          factorWeights = factorWeights:+model.intercept
          factorWeights:+stockID
          }
  }

/**
  * Calculate the return of a particular instrument under particular trial conditions.
  */
  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
 }

 /**
  * Calculate the full return of the portfolio under particular trial conditions.
  */
 def trialReturn(trial: Array[Double], instruments: Seq[Array[Double]]) = {
  var totalReturn = 0.0
 // var totalR = new Array[(Double,Array[(Double,Double)])] (1)
  val individualTrialReturns = new Array[(Double,Double)](instruments.length)
  var i = 0
  for (instrument <- instruments) {
   val instrumentReturn = instrumentTrialReturn(instrument.dropRight(1), trial)
    totalReturn = totalReturn + instrumentReturn
   if (i < instruments.length ){
      individualTrialReturns(i) = (instrument(instrument.length-1),instrumentReturn)
      i= i+1
   }
  }
  ((totalReturn / instruments.size),individualTrialReturns)
 }

// Total Trial run for a particular parallel segment
def trialReturns(
    seed: Long,
    numTrials: Int,
    instruments: Seq[Array[Double]],
    factorMeans: Array[Double],
    factorCovariances: Array[Array[Double]]): Array[(Double, Array[(Double, Double)])] = {
  val rand = new MersenneTwister(seed)
  val multivariateNormal = new MultivariateNormalDistribution(rand, factorMeans,
    factorCovariances)

  val trialReturns = new Array[(Double,Array[(Double,Double)])](numTrials)

  for (i <- 0 until numTrials) {

    val trialFactorReturns = multivariateNormal.sample()
    val trialFeatures = featurize(trialFactorReturns)
    trialReturns(i) = trialReturn(trialFeatures, instruments)
  }
```

```
  trialReturns

 }
```

//Calculate VAR

```
def PercentVaR(trials: RDD[Double]): Double = {
  val percentParam = 100 / financialRiskAnalysisParam.getOrElse(0)

  val topLosses = trials.takeOrdered(math.max(trials.count().toInt / percentParam, 1))
  topLosses.last
 }
```

//Calculate ES

```
 def PercentES(trials: RDD[Double]): Double = {
   val percentParam = 100 / financialRiskAnalysisParam.getOrElse(0)
  val topLosses = trials.takeOrdered(math.max(trials.count().toInt / percentParam, 1))
  topLosses.sum / topLosses.length
 }
```

//Obtain confidence level for VAR and ES

```
 def confidenceInterval(
    trials: RDD[Double],
    computeStatistic: RDD[Double] => Double,
    numResamples: Int,
    pValue: Double): (Double, Double) = {
  val stats = (0 until numResamples).map { i =>
    val resample = trials.sample(true, 1.0)
    computeStatistic(resample)
  }.sorted
  val lowerIndex = (numResamples * pValue / 2 - 1).toInt
  val upperIndex = math.ceil(numResamples * (1 - pValue / 2)).toInt
  (stats(lowerIndex), stats(upperIndex))
 }
```

//Main Method

```
val filePrefix = "/home/hadoop/data/StockNamesMap.csv"
val stockNameMap = readStockNameMap(new File(filePrefix))

val start = new DateTime(2000, 2, 13, 0, 0)
val end = new DateTime(2015, 8, 7, 0, 0)
val rawStocks = readAllStockHistories(new File("/home/hadoop/data/stocks/")).filter(_.size >=
260*5+10)
```

```
val rawStocksrdd = sc.parallelize(rawStocks)

val stocks = rawStocksrdd.map(trimToRegion(_,start, end)).map(fillInHistory(_,start, end))

val stocksReturns = stocks.map(valueReturns)

val factorsPrefix = "/home/hadoop/data/factors/"
val factors1 = Array("NDX.csv","SNP.csv","CrudeOil.csv","Bonds.csv").
   map(x => new File(factorsPrefix + x)).
   map(readStockHistory)

val fctors = factors1.
   map(trimToRegion(_, start, end)).
   map(fillInHistory(_, start, end))

val factors = sc.parallelize(fctors)

val factorsReturns = factors.map(valueReturns)

val temp1 = factorsReturns.flatMap { arrayElement =>
  arrayElement filter {
    case (x: String, y:Double) => x == "NDX"
  }
}
val temp2 = factorsReturns.flatMap { arrayElement =>
  arrayElement filter {
    case (x: String, y:Double) => x == "SNP"
  }
}
val temp3 = factorsReturns.flatMap { arrayElement =>
  arrayElement filter {
    case (x: String, y:Double) => x == "CrudeOil"
  }
}
val temp4 = factorsReturns.flatMap { arrayElement =>
  arrayElement filter {
    case (x: String, y:Double) => x == "Bonds"
  }
}

val factor1 = temp1.map(x => x._2).toArray
val factor2 = temp2.map(x => x._2).toArray
```

```
val factor3 = temp3.map(x => x._2).toArray
val factor4 = temp4.map(x => x._2).toArray

val factorMat = factorMatrix(Seq(factor1,factor2))

val factorMatrdd = sc.parallelize(factorMat)

val factorFeatures = factorMatrdd.map(featurize)

val temp = stocksReturns.toArray
val labels = temp.map(dataMatrix)

val temp = labels.map(x => dataModel(x,factorFeatures.toArray))

val factorWts =  temp.map(computeFactorWeights)
```

// Covariance Calculation

```
val factorCov = new Covariance(factorMat).getCovarianceMatrix().getData()

val factorMeans = Seq(factor1,factor2).map(factor => factor.sum /factor.size).toArray

//factorFeatures

val broadCastFactorWts = sc.broadcast(factorWts)

val numTrials = nTrials.getOrElse(0)
val parallelism = numTrials / 10
val baseSeed = 1001L
```

// Generate different seeds so that our simulations don't all end up with the same results

```
val seeds = (baseSeed until baseSeed + parallelism)
val seedRdd = sc.parallelize(seeds, parallelism)

// create an empty map
var stocksTrialsReturns = scala.collection.mutable.Map[String, Double]()

val trialsrdd = seedRdd.flatMap(
    trialReturns(_, numTrials / parallelism, broadCastFactorWts.value, factorMeans, factorCov))

val trials = trialsrdd.map(line => line._1)
```

```
val stockreturns = trialsrdd.map(line => line._2)

var j = scala.collection.mutable.Map[Double, Double]()
val xyz = stockreturns.collect.flatten

//Mapping each stock's aggregated trial returns
val stocksSummations = xyz.foreach(i => if(j.get(i._1) == None){
                                j+= (i._1->i._2)}
                                else {
                                   j+= (i._1 -> (j.get(i._1).get+ i._2))
                                })

//Calculate Financial Risk Parameters
val valueAtRisk = PercentVaR(trials)
val conditionalValueAtRisk = PercentCVaR(trials)
println("VaR : " + valueAtRisk)
println("ES : " + conditionalValueAtRisk)
```

//Saving the aggregated instrument losses is written to S3

```
fileLoc = "s3://advancedbigdataanalytics/outputReturns"+rnd.nextInt()
sc.parallelize(j.toSeq).saveAsTextFile(fileLoc)
```

// Back testing

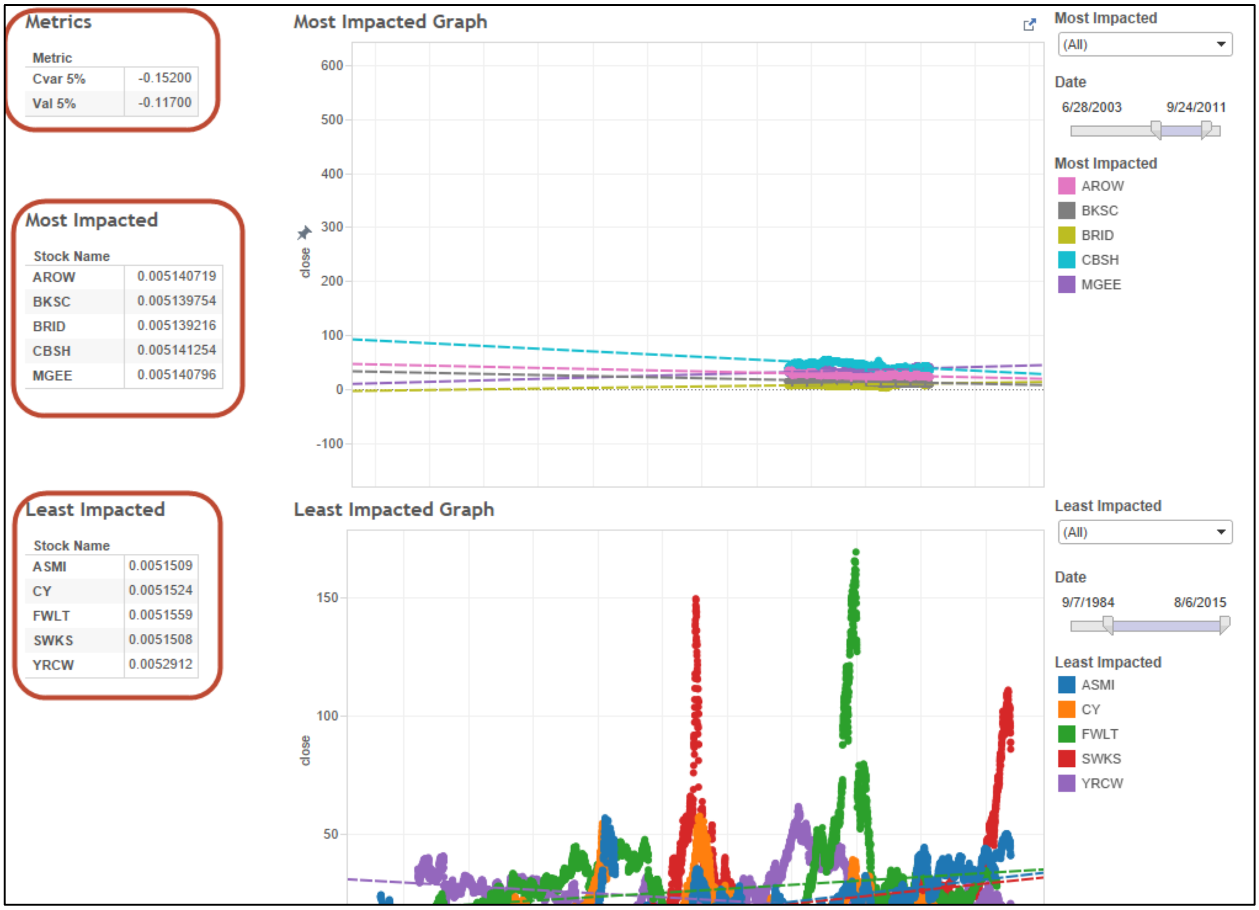
```
def failureCount(stocksReturns: Seq[Array[Double]], valueAtRisk: Double): Int = {
    var failures = 0
    for (i <- 0 until stocksReturns(0).size) {
      val loss = stocksReturns.map(_(i)).sum
      if (loss < valueAtRisk) {
        failures += 1
      }
    }
    failures
  }

  def kupiecStatistic(total: Int, failures: Int, confidenceLevel: Double): Double = {
    val failureRatio = failures.toDouble / total
    val logNumer = (total - failures) * math.log1p(-confidenceLevel) +
      failures * math.log(confidenceLevel)
    val logDenom = (total - failures) * math.log1p(-failureRatio) +
```

```
    failures * math.log(failureRatio)
   -2 * (logNumer - logDenom)
  }

def kupiecPValue(
stocksReturns: Seq[Array[Double]],
valueAtRisk: Double,
confidenceLevel: Double): Double = {
val failures = failureCount(stocksReturns, valueAtRisk)
val total = stocksReturns(0).size
val testStatistic = kupiecStatistic(total, failures, confidenceLevel)
   1 - new ChiSquaredDistribution(1.0).cumulativeProbability(testStatistic)
  }

val varConfidenceInterval = confidenceInterval(trials, PercentVaR, 100, .05)
val esConfidenceInterval = confidenceInterval(trials, PercentES, 100, .05)
println("VaR confidence interval: " + varConfidenceInterval)
println("ES confidence interval: " + esConfidenceInterval)
println("Kupiec test p-value: " + kupiecPValue(labels.toSeq, valueAtRisk, 0.05))
```

Summarization

The prediction results were uploaded to Hive tables. A financial risk summary dashboard was created by connecting tableau to hive data store.

The following are some of the visualizations that are available for financial risk analysis



This dashboard provides the user with a list of top 5 most impacted stocks and top 5 least impacted stocks. This dashboard also provides insights on the % risk associated with these stocks over the specified period in time.

The frames on right side of the dashboard also displays the historical performance of these stocks under consideration.

Following were the observations on performing this financial risk analysis

- ***Value @ Risk with 95 % Confidence Interval:*** *- 0.11*
- ***Expected Shortfall with 95% Confidence Interval:*** *- 0.152*

A back testing of value at risk and expected shortfall was performed by computing the confidence intervals using Kupiec testing and the numbers were acceptable

- ***VaR Confidence Interval:*** (-0.121, -0.07)
- ***ES Confidence Interval:*** (-0.1649, -0.080)

Conclusion

Here we looked into the given stocks on an exploratory basis to gain insights and the relationship between external factors. For each stock, we computed factor weights with the help of linear models using historical data. We also looked into simulating external factors using multivariate normal distributions. With a number of random trials, we computed the stocks losses and full trial losses computed 5% VaR. We also performed an in-depth analysis on the stocks that were most/ least impacted during the specified time period which helps in customizing the portfolio to minimize the losses.

References

- O'Reilly - Advanced Analytics with Spark
- https://spark.apache.org/docs/1.3.1/api/scala/index.html#org.apache.spark.package
- https://spark.apache.org/news/spark-1-3-1-released.html
- http://gethue.com/hadoop-tutorial-hive-query-editor-with-hiveserver2-and/
- http://kb.tableau.com/articles/knowledgebase/hadoop-hive-connection
- https://gist.github.com/andershammar/224e1077021d0ea376dd
- http://www.value-at-risk.net/backtesting-coverage-tests/