CPSC597 MS Computer Science Masters Project

"Finding Alpha with Spark"

5th May 2015

Participants

Investigator: Patrick Haren, pharen@my.bridgeport.edu
Advisor: Julius Dichter, dichter@bridgeport.edu

Introduction

Quantitative active portfolio management is the science of applying rigorous analysis and a rigorous process to try to beat market benchmarks. The process involves researching ideas, forecasting returns, implementing portfolios, and observing and refining their performance. Calculations are in the areas of statistics, regression and optimization. In particular, for research and forecasting, historical market and economic data is used – such as stock volumes and prices. Various external factor relationships are identified for use in hedging and for identifying arbitrage opportunities. Trading strategies are "backtested" to simulate trigger events and overall performance.

Apache Spark is a cluster computing system, built on Scala, for general purpose processing of large datasets. It can run on Hadoop clusters and access Hadoop data sources. In addition to the MapReduce programming model, other types of computations are supported, including higher-level collection functions, interactive queries and stream processing.

In this project we have investigated the appropriateness and benefits of using Scala and Apache Spark to construct applications for active portfolio management. A prototype has been built that implements data management, calculations, quantitative research functions and strategy testing functions. Components such as these are typical for use in hedging, arbitrage and trading strategy development. We are particularly interested in problems requiring large amounts of computation and/or larger datasets. Thus, we have chosen data processing scenarios that could be a good fit for Spark – namely fine-grained pricing data (minute-by-minute), as well as analysis functions that are performed over thousands of datasets (corresponding to data for individual market equities).

Spark includes other language bindings, particularly for Python, which might lend itself to use by data scientists that are already familiar with numerical analysis frameworks such as NumPy, SciPy, pandas and the PyData[*] stack. However, since the Scala language forms the

[*] http://pydata.org/, http://scipy.org/

[†] We actually started the project using eclipse, but switched to IntelliJ when eclipse ran into

foundation for Spark and since the language provides object-oriented and functional programming capabilities, we are also interested in how well Scala can work as a structured, expressive language for quantitative analysis. Can these programming paradigms help simplify some of the complexities found in quantitative analysis code, such as in functions related to array and data structure traversal?

The prototype system described in this report is used to evaluate these questions. We propose a software system architecture that could form the basis for some typical use-cases applicable to quantitative analysis applications. For each component, we use example code to demonstrate the concepts of functional programming and structured design using object-oriented programming (courtesy of Scala) as well as structuring computations for parallelism (courtesy of Spark).

The technologies

Scala

Scala is a language that compiles to java bytecode and, thus, runs on standard JVMs. It is a highly expressive language providing both object-oriented and functional-programming capabilities. It is statically typed and compiled, but includes the ability to perform type inference at build time, so we don't (usually) have to define the explicit types. It shares many of the functions of Java and interoperates with Java.
The language comes with an interactive command shell (the REPL), but also has IDE support in eclipse and IntelliJ IDEA[†].
Here are some simple examples of language concepts (see [6] and [7] for more):

Variables (mutable) and values (constant):

```
var i : Int = 10  // an integer
var j = 11        // also an integer
val k = "foo"     // a read-only string value
```

Regular functions:

```
def az(s : String) : String = "yall " + s
def mn(s : String) = {
  s + ", you betcha"
}
```

Functions are first class objects (closures):

```
(x: Int) => x * 3   // triple function
x => x * 3          // triple , but type is inferred
_ * 3               // triple, shorthand (_ is the argument)

x => {              // just a bigger block of code
  val three = 3
  x * three
}

def triple(x : Int): Int => x * 3 // named the function!
```

Functional methods applied to collections:

```
val myList = List(2, 4, 8)
myList.foreach(x => println(x)) // prints 2, 4, 8
myList.foreach(println)         // ditto
```

[†] We actually started the project using eclipse, but switched to IntelliJ when eclipse ran into compilation issues with dependencies within the breeze libraries.

```
myList.map(x => x * 3)          // returns a new List(6, 12, 24)
myList.map(_ * 3)               // ditto
myList.map(triple)              // ditto

myList.filter(x => x>2)         // new List(4, 8)
myList.filter(_>2)              // ditto

myList.reduce((x, y) => x + y)  // 14
myList.reduce(_ + _)            // ditto
myList.sum                      // ditto
```

Method on Seq[T]	Explanation
map(f: T => U): Seq[U]	Each element is result of f
flatMap(f: T => Seq[U]): Seq[U]	One to many map
filter(f: T => Boolean): Seq[T]	Keep elements passing f
exists(f: T => Boolean): Boolean	True if one element passes f
forall(f: T => Boolean): Boolean	True if all elements pass
reduce(f: (T, T) => T): T	Merge elements using f
groupBy(f: T => K): Map[K, List[T]]	Group elements by f
sortBy(f: T => K): Seq[T]	Sort elements
…..	

Spark

Spark is an open source framework for cluster computing, originating from UC Berkeley AMPLab. Spark maintains MapReduce's linear scalability and fault tolerance, but extends it. Rather than relying on a rigid map-then-reduce format, its engine can execute a more general directed acyclic graph (DAG) of operators. However, it can also run on YARN and interact with Hadoop components, such as HDFS and InputSplit classes.
(From [4]) Spark programming involves the use of datasets, usually residing in some form of distributed, persistent storage like HDFS. Writing a Spark program typically consists of a few related things:

- Defining a set of transformations on input datasets.
- Invoking actions that output the transformed datasets to persistent storage or return results to the driver's local memory.
- Running local computations that operate on the results computed in a distributed fashion. These can help decide what transformations and actions to undertake next.

The solution solves for the intersection between the two sets of abstractions the framework offers: storage and execution. Spark pairs these abstractions in an elegant way that essentially allows any intermediate step in a data processing pipeline to be cached in memory for later use.

For our system implementation, we have leveraged the core concepts of Spark – Resilient Distributed Datasets (RDDs).

The problem domain

Quantitative Analysis

Quantitative Analysis is a body of methods and theory applied to finance and investing. There are many aspects to this area, but some key concepts are CAPM, APT and the APM fundamental law, described briefly here. Refer to references [1], [2] and [8], as well as sites such as Investopedia, for more detail.

Active Portfolio Management (APM) [1] is about "beating the market". It relies on statistical methods to measure and manage risk and to "find alpha" (i.e. the elusive "risk free" better-than-market returns). The phrase "alpha" comes from the Capital Markets Model (CAPM), which defines the relationship of an asset to the overall market as:

$$r_a = r_f + \beta_a(r_m - r_f) + \alpha$$

where r_a is the return of an asset/equity/portfolio/instrument
r_f is the risk-free rate (close to 0% for past 8 years).
r_m is the expected return of the "market"
β_a is the asset's correlation to the market
α is the excess return of the asset (zero in an efficient market scenario)

The Arbitrage Pricing Theory (APT)[1] (Created in 1976 by Stephen Ross) is another closely related (and more general) asset pricing model. It states that an asset's returns can be predicted using the relationship between the asset and many common risk factors. The relationship can be expressed as:

$$r_a = r_f + \sum_{i=1}^{n} b_{a,i} f_i$$

where f_i is a risk factor
$b_{a,i}$ is asset's correlation to the risk factor

Risk factors can be from many independent macro-economic variables, but if you reduce the factors to just "the market", the model becomes CAPM. Arbitrage is the opportunity presented due to asset mispricing. A mispriced security will have a price that differs from the theoretical price predicted by the model. By "going short" on an overpriced security, while concurrently "going long" on the portfolio the APT calculations were based on, the arbitrageur is in a position to make a risk-free profit (in theory). This pair-trading strategy is simulated in the project.

The APM Fundamental Law [1] defines the relationship between (arbitrage) information and the number of times such opportunities can be created/found:

$$IR = IC.\sqrt{breadth}$$

where IR = the fund manager's overall opportunity (information ratio)
IC = the manager's information coefficient (measure of skill)
breadth = the number of independent forecasts of exceptional return we make per year

Thus, the effectiveness of a strategy can be thought of as the combination of accuracy and number of opportunities it presents.

Generally, quantitative methods are based on determining the exposure of assets to risk factors and/or determining the theoretical price (and, thus, looking for arbitrage opportunities). Reference text [2] provides a sampling of some of these methods, such as NNR (Neural Network Regression), Co-Integration, Ridge Regression, LASSO (and other bias-variance trade-off concepts). The text provides fun Sunday reading for Stats enthusiasts.

A proposed system architecture

Typical quantitative research and analysis activities could be supported via an environment that provides the necessary background data in a structured online format, for programmatic access, but also in a format that supports scaling out of the amount of data and the amount of processing. We'll call this the "structured data" component.
We propose that individual research scenarios will typically take on the form of individual computations on a set of data that each fit in memory on a typical worker node, but where these computations are repeated across a large set of equities. These individual computations will be performed in the "analysis sessions" component. A researcher will likely interact with the system via a workstation, providing visualization of the results, either via a web-application or via connections and imports to MS Excel. We refer to this component as "visualization".
Since research sessions will be interactive, the structured data, required to support the analysis sessions and visualization, should be available without reprocessing from the raw input data. Thus, the structured data component should remain online/hosted, such as in-memory and/or in close storage to each worker node. A component that we call "data preparation" should manage the intake, conversion and structuring of raw data. It should refresh the data based on latest information, from information sources. The refresh cycle may be overnight (for end-of-day market data), but more frequently updated sources could be supported via stream processing on Spark.

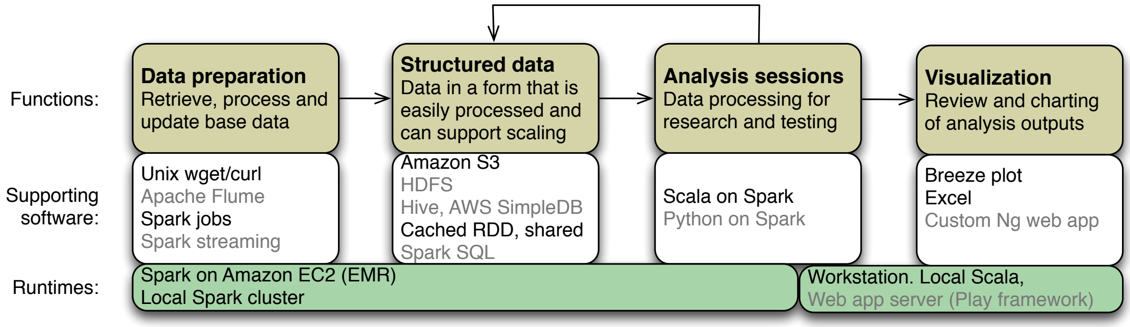

fig. 1 - system components

The relationships between these core functional components are depicted in fig. 1. Below each component, the software and services that can support the functions are shown. For our prototype system, the highlighted software and services have been used.

We now describe the design concepts supporting each component, using the prototype system as example. Alternatives and further research and development areas are also discussed at the end of each component section.

Data preparation

For our use-cases, data preparation is an important and non-trivial component. The issues include:

i) Inexpensive, readily-available minute-by-minute raw trading data is limited. For example, our google data-source only goes back 14 days at most.
ii) Data is not immediately usable – i.e. is typically in formatted, delimited text with additional header rows and different formats for each field
iii) Some stocks suffer from survivorship bias and other changes. For example, Allergan (AGN) was acquired in March 2015, resulting in the symbol being retired
iv) Stock dividends and splits should affect the adjusted close price of collected data
v) Not all stocks are actively traded every minute, leading to apparent data gaps
vi) Not all data services use the same naming conventions for stock symbols. e.g. BIO-B vs. BIO.B

Through research and design, these issues were addressed for the prototype.

For 60-second resolution trading data, we are using an "undocumented" google API. If you google "undocumented google finance API" [9], you can find a description. Each equity is a separate resource, accessed via an http client. Query parameters specify the data granularity (60 seconds), number of days of history (max 14) and equity name.

After the header rows, the text resource returned contains rows with date/time, closing price, high price (over the 60-second period), low price, opening price (for the period) and volume. The first row for a new day contains an 'a' followed by seconds since the epoch. An example snippet (spaces added for readability), is shown:

```
a1428932100, 32.15, 32.27, 32.1,  32.27, 1938
          5, 32.1,  32.15, 32.1,  32.15, 400
          6, 32.03, 32.11, 32.01, 32.11, 600
```

Thus, interpreting the resource requires that we split each row along commas and convert the first row unix date field to a date and carry this value as a base date-time for other rows within the same day. To build-up historical data, we started to collect this data straight away and appended to it as time progressed. Basic retrieval is via a unix curl script (similar to wget), retrieving from the resource (param $1 = symbol name):

```
curl … http://www.google.com/finance/getprices?i=60&p=14d&f=d,o,h,l,c,v&df=cpct&q=$1
```

For more historical daily-level data, we retrieve resources from Yahoo! Finance:

```
sym=`echo $1 | sed -e 's/\./-/g'`

curl … http://real-chart.finance.yahoo.com/table.csv?

       s=$sym&a=00&b=2&c=2000&d=$mth&e=$day&f=$yr&g=d&ignore=.csv
```

The first line substitutes dots in the symbol name with dashes, to follow the Yahoo naming scheme. The retrieval scripts are driven by a master list of stock symbols, based on the active symbol list maintained by nasdaq.com. We added a pause in-between each retrieval, so as to not overload the resource sites. After an initial run with the full list, stocks that have no active trading data were eliminated from the master list. The retrieved raw data is saved to local file system, for local-mode testing, and S3[10] for running on a cluster. Another simple option would be to store on HDFS, if running on a persistent Hadoop cluster. For the prototype, we have not used HDFS, since local mode multi-core performance will be similar from the direct file-system and HDFS storage on AWS[10] requires the use of persistent EBS[10] storage for each node, which was beyond the budget of the project.

The second phase of data preparation is to process the raw pricing data and prepare it as online, structured data for processing.

The daily data includes an "adjusted close" column, which we use to determine the adjustment ratio that should be applied for each day. Thus, to build a full picture of an equity's historical and minute-by-minute data, we process the Yahoo data first, then append the adjustment ratio to the minute-by-minute data as a step during google data processing.

To match resolution of data, we have removed stocks that are not traded daily and have implemented an interpolation function to fill in missing minute data. We treat the gap period as having the same price as last trade, but set the volume to zero.

To complete the picture of each equity, we compile the interpreted market data into fields of an EquityData object and have also added convenience data, such as mean and variance. Within the PriceData class, representing each period, we defined a convenience adjClose field (implemented as a function) that provides the adjusted closing price. This hides the additional complexity of applying the adjustment multiplier to our price data every time it is used.

The full Scala code is in the separately attached source code tarball.

Opportunities for more R&D: For the data preparation component area, opportunities for additional functionality include adding data loaders for commercial data-source providers. Loading to HDFS could also be implemented – with specific focus on making sure that data is partitioned along the boundary of individual equities and that each node is aware of the data locality. We also experimented with running the fetch scripts on workers nodes, called from Spark. However, since this data is prepared at the end of a trading day, retrieving more data faster (and overloading the resource sites) was not needed. Future projects could identify some more real-time data sources and implement Spark streaming to ingest and structure such data more rapidly.

Structured data

In the second phase of data preparation, the data is transformed into structured data that can be easily used within the analysis sessions. We have designed some simple Scala case classes that store each of the pricing data fields, as well as some summary information. Within the PriceData class, we also implemented a few convenience methods to give the objects the appearance of having additional fields (but which are really transformations of the underlying data).

Each PriceData element (representing a trading period – e.g. a day or a minute), is structured as follows:

```scala
/* US equity pricing data */
case class PriceData(epochTime : Long, open : Double, close : Double, high : Double, low: Double, volume:
Long, adjuster: Double)
{
  def tradingTime(tz: String) = Instant.ofEpochSecond(epochTime).atZone(ZoneId.of(tz)).toLocalTime
  def tradingDate(tz: String) = Instant.ofEpochSecond(epochTime).atZone(ZoneId.of(tz)).toLocalDate
  def nyTradingDate = tradingDate("America/New_York")
  def nyTradingTime = tradingTime("America/New_York")
  def adjClose = close * adjuster
}
```

The packaged equity data is structured to include minute-by-minute, as well as longer-term daily historical data, using the EquityData class:

```scala
case class EquityData(name: String, dailyPrices : Array[PriceData], minutePrices : Array[PriceData])
{
  val tradingDaysPerYr = 252
  val totalDays = dailyPrices.length
  val totalMins = minutePrices.length

  val yrOfDaily = dailyPrices.slice(totalDays-tradingDaysPerYr , totalDays-1)
  val yrOfDailyClose = yrOfDaily.map(_.adjClose)
  val allMinClose = minutePrices.map(_.adjClose)

  def yrAvgDailyVolume = yrOfDaily.map(_.volume).sum/tradingDaysPerYr   // reduceLeft(_ +
_)/tradingDaysPerYr
  def yrStdDev = stddev(yrOfDailyClose)
  def yrMean = mean(yrOfDailyClose)
}
```

Note that instead of calling reduceLeft to sum neighbouring elements, Scala has provided a sum method, applicable for collections of numeric data.

On Spark, the data used for processing can be thought of as either:

i) elements within RDDs
ii) elements passed in function calls
iii) elements in broadcasted reference data

In all cases, we use our structured equity and pricing data class structures.

For (i) [elements within RDDs], these elements are loaded directly within the worker nodes, by processing the raw data files (from the data preparation step). We 'persist'/cache the RDDs, so that distributed data is available throughout the analysis sessions. The data preparation is defined in a function called equitiesOnSpark:

```
def equitiesOnSpark(firstTradeDate: LocalDate, rawMinBasePath: String, rawDailyBasePath : String, sc:
SparkContext)
```

This function provides an RDD of EquityData elements. To produce this RDD, the function first produces an RDD of daily data for each stock, from the raw data on S3:

```
val dayFiles = sc.wholeTextFiles(rawDailyBasePath).partitionBy(symbolPartitioner)
val symbolDailies = dayFiles.mapPartitions(fileNamesToSymbols,true)   // content of each daily file, keyed
by symbol
val symbolDayData = symbolDailies.mapValues(contents =>   toPerDayStockSeries(contents.lines)).persist()
```

Note that we have defined a custom partitioner class (SymbolPartitioner) so that we can keep data for the same symbol on the same partition. This provides two performance boosts – 1. we can use the mapPartitions method knowing that all symbols are covered and, 2. Spark is aware when the partitioner is the same for this RDD and our later RDD for minute data, so that the RDD join operation can also take place separately on each worker node without the need for an expensive network shuffle.

We next produce an RDD of raw minute data (from the google text files). We have to produce this first, so that we can then transform it into structured minute data, once we have the daily data (that includes the all-important dividends & splits price adjuster):

```
val minuteFiles = sc.wholeTextFiles(rawMinBasePath).partitionBy(symbolPartitioner)
val symbolMinutePieces = minuteFiles.mapPartitions(fileNamesToSymbols,true)
val symbolMinutes = symbolMinutePieces.reduceByKey(_ + _)   // content of each per min, keyed by symbol
```

The final step is to join the two RDDs and complete the build of our EquityData structures for each symbol. It is at this point that the per-minute raw data can be processed into structured PriceData elements for inclusion in the full EquityData objects:

```
// This should be an inexpensive join, since spark will not need to shuffle
val symbolPackets = symbolMinutes.join(symbolDayData)   // key, (minuteText, dayData)
// RDD to return, should be an RDD with value = equity case instance (of minute data and day-data)
symbolPackets.mapPartitions(_.map(kp => buildEquity(kp._1, kp._2)),true)
```

Note: the buildEquity function takes the raw minute text data, plus the existing daily data, builds the structured minute data and then combines into the final EquityData structure. While building the minute data, we also interpolate over any gaps in the data.

For (iii) [elements in broadcasted reference data], these elements are built from a subset of the prepared data and are, thus, collected from the worker nodes and redistributed as reference data. This allows the shared data to be available regardless of which partition in the overall dataset the node is running on.

```
val basket = equities.filter(e => basketSymbols.contains(e.name)).collect()
// share the full basket index, so that it is available to each worker w/o a full SeDe each time
val commonBasket = sc.broadcast(basket)
```

Quantitative analysis involves various forms of statistical analysis, such as Neural Network Regression (NNR)[2] and co-integration with ridge regression[2]. These regressions are often performed using market factor data and market indexes. Since this common factor data is used across different calculations, we perform this data distribution once in advance (using the broadcast facility in Spark). The data can then be used as reference on each worker node. This avoids serializing, transmitting and deserializing (SeDe) the same data multiple times.

The net result of these steps is that we have structured data (in EquityData objects) available for analysis sessions, through a Spark RDD of the full set of equities plus a reference set of index/market data locally available on each worker node, also in structured EquityData objects. Analysis functions can be performed on this data by submitting the functions to element-wise methods of the RDD, as we'll see in the next section.

The full Scala code is in the separately attached source code tarball.

Opportunities for more R&D: For the structured data component, we have opportunities to build out the system some more. We could add additional reference data, such as external trend data, like economic indicators, oil, gold, commodities futures, etc. Depending on how frequently the various data sources are used, we could store some data in structured online key-value datastores, such as SimpleDB, DynamoDB, Cassandra or Hive.

Analysis sessions

Analysis sessions are where the content/input of quantitative analysts would come in. The analysts would define analysis functions to research various hedging and trading opportunities. They would define functions that run against equities and leverage the reference data. Research against the, approximately 7400, equities can be performed in parallel by Spark. Analysts would write simple functions that take an equity as input and the reference data as input and produce a structured result set as output. These functions would be submitted as DAG‡s to the RDD. More sophisticated cross-RDD analysis could also be performed by writing driver code. This code could, for example, zip an RDD of result data from one analysis with an RDD of related result data from another analysis.

The diagram in fig. 2 shows the typical interaction in analysis sessions.

‡ Directed Acyclic Graph – in this case, the graph represents a set of delayed operations (to be performed on the cluster).

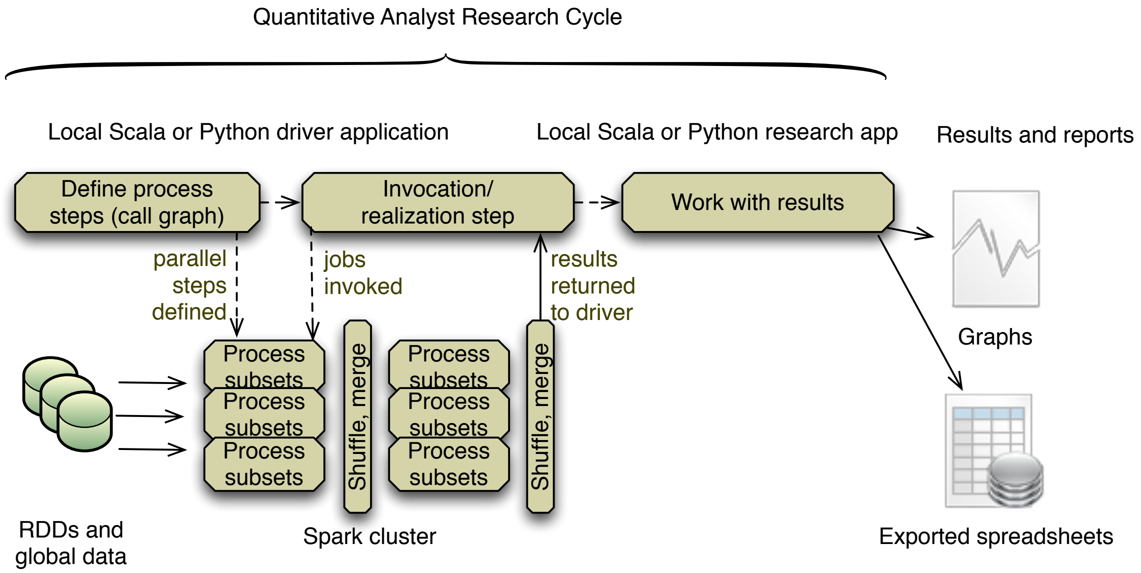

fig. 2 - research sessions on Spark

Local extracts or an online web-site could be used to produce visualizations of the result data.

There are literally thousands of different quantitative methods in this field. We have taken a representative example from the reference texts to demonstrate how the methods would be implemented using our Spark framework. The example is taken from Applied Quantitative Methods for Trading and Investment[2], chapter 2 – co-integration.

With co-integration we perform a regression of an equity's price data against a collection of equities. What we are looking for is a representative weighting of each equity towards the price data of the target equity. The weightings represent each contributing equity's "influence" on the target equity, as if the contributing equities were themselves market factors. If we have a diverse set of contributing equities, the technique should provide an implicit correlation of the target equity to its underlying market and economic influences. In the simple case of co-integrating the target equity with a single market index (e.g. S&P 500), the weighting produced is the beta of the equity in CAPM. By co-integrating with a "basket" of equities, we will have a list of betas. The value/price of the target will be represented by a synthetic fund, whose price at period t is given by:

$$S(t) = \sum_{n=1}^{N} \beta_n . E_n(t) + \epsilon$$

Where β_n represents the factor weights for each contributing equity E_n. The synthetic fund could be a portfolio of equities used as a hedge against the unknown implied market factors. Alternatively, the weighted basket of equities could be used for "statistical arbitrage" trading opportunities, whereby the basket and the target equity are pair-traded. We have included an example of back-testing this trading strategy, using the 60-second price data to identify the trading events.

The first step of the analysis is to perform co-integration of each potential equity against the basket of potential equities (our "index"). Our basket is made up of the S&P500 list of stocks, plus an ETF that represents the overall S&P. As recommended in the text, the co-integration should include a ridge factor λ. The betas are determined using a number of matrix operations, as follows:

$$\beta = (C^T C + \lambda\sigma I)^{-1} Ct$$

Where β is the list/vector of factor weights, C is the matrix of price data for each contributing equity and Ct is the matrix multiplication of the target price data against this overall matrix. We also include a fictitious "unity" stock, of constant value 1, within the matrix, which allows to include an offset bias beta factor in the co-integration.

Linear algebra functions are implemented in Scala, in the breeze.linalg package. Using this package, the full implementation of this beta list calculation is straightforward:

```
def cointegrated(testEquity: EquityData, basket: Array[EquityData], learningDays: Int, validatingDays:
Int) =
{
  val lamda = 0.01

  val totalDataDays = learningDays+validatingDays
  val targetLearningStartPos = testEquity.dailyPrices.length-totalDataDays

  /* Value of a cell in the 'C' matrix (closing prices over learning days[row], for each stock in basket
[col])
   * row = values on a particular trade period
   * col = stock# (or constant/unity, if col = 0)
   */
  def CValue(row: Int, col: Int) : Double =
  {
      if (col == 0) 1.0 // col 0 is the offset bias
      else {            // other cols are for each stock in basket
        val learningStartPos = basket(col-1).dailyPrices.length - totalDataDays
        basket(col - 1).dailyPrices(learningStartPos + row).adjClose
      }
  }

  /* co-integration formula:
   * inverse(C.transposed*C + lamda*sigma*I)*(C.transposed*t)
   * where C = pricing of each stock, over training days
   * t = pricing of target stock, over same period
   * lamda = ridge coefficient [regularization tuning factor - to balance bias vs. variance]
   * {source: Applied quantitative methods for trading and investment, Ch. 2 (Burgess)}
   */
  val C = DenseMatrix.tabulate(learningDays, basket.length+1)(CValue) // C - the matrix of price at a time
(row) per stock (col)
  val target = DenseVector(testEquity.dailyPrices.slice(targetLearningStartPos,
targetLearningStartPos+learningDays).map(_.adjClose))
  val Ctransposed = C.t
  val Ct : DenseVector[Double] = Ctransposed * target
  val corr : DenseMatrix[Double] = Ctransposed * C
  for (i <- 0 to basket.length) corr(i,i) = (1+lamda) * corr(i,i)    // apply the ridge factor
(lamda*sigma*I)
  val invCorr = inv(corr)

  val betas : DenseVector[Double] = invCorr * Ct

  betas.toArray
}
```

That's it (Scala is concise)!

We use the tabulate method to populate the value of each cell within the matrix. To apply the ridge factor, we take a short-cut of traversing the matrix diagonally and applying the factor directly (which would be the same as adding the weighted identity matrix). Typically, the Scala compiler can infer the desired return type, but in a few places we had to include the type – e.g. DenseVector[Double] – since multiple types were possible.

The paper also describes how to measure the quality of the fit. We ideally want to have a synthetic fund that, when compared to the target fund, is closely aligned and where the residual (difference between the two) tends to be a random series around the mean – i.e. that has a mean reversion and does not have any trending factor. This can be measured using a variance ratio. The ratio is a measure of the series movement after so many days, compared to series movement earlier. If the series is mean reverting, the variance ratio should be trending lower than 1.0. We use 30 days as the test for variance ratio score. To calculate the variance within the test window, there is no convenient built-in function for this. So we quickly define one:

```
def windowVariance(prices: Array[Double], w1Start: Int, w2Start: Int, windowSize: Int) =
  (0 to windowSize-1).map(day=> sqr(prices(w1Start+day)-prices(w2Start+day))).sum/(daysInSlider-1)
```

We are basically computing the variance from first principles and comparing the residual series over different time periods (windowSize days starting at w1Start vs. w2Start). The residual series itself is briefly defined as:

```
val residuals = synthStock.zip(targetStock).map(pair => pair._1 - pair._2)
```

Here we are using the handy zip function to join elements of two collections into a collection of pairs. We then produce the difference by passing the “difference function” into an element-wise map on the pair collection.

In our hypothetical analysis scenario, we want to identify a smaller basket of the factor contributors (so that pair-trading transaction costs are smaller). The optimalSmallBasket function performs this by:

1. running a full co-integration (all index equities included)
2. picking the highest 5 betas
3. re-running the co-integration with the 5 highest correlating equities
4. producing a summary of the fit and quality measurement, including the beta terms

This function becomes the primary analysis that we repeat on each equity, searching for good correlation scenarios (as determined by the variance ratio criteria):

```
val summaries = equities.map(target => StockCoIntegration.optimalSmallBasket(target, commonBasket.value,
1000, 200, 5)).collect()
// looking for lowest variance ratios - if significantly lower than 1.0, it probably represents a mean-
reverting residual
val sortedSummaries = summaries.sortBy(_.day30VR)
```

Equities is our RDD of EquityData, containing 7400 stocks across the cluster. The collect() method forces the Spark job to run (since we are asking for the result data, it can no longer be lazy!).

Opportunities for more R&D: There are many forms of analysis possible, based on papers published on quantitative analysis. Further examples could identify additional libraries, additional data sources and/or additional types of processing that we would want to implement on Shark. For example, there are scenarios that would use a Neural Network Perceptron library. There may be other scenarios where we could leverage the machine learning libraries – to perform clustering or classification. Depending on the size of data, this might need to use the distributed MLib libraries, instead of packages such as breeze.

Visualization

Continuing with the analysis sessions section, our prototype shows two ways in which to support visualization of the analysis progress. The first is to use the breeze.plot package. This package uses the java jfreechart under the covers. The advantage of this package is that it provides an instant display, after the execution. It does, however, include a lot of external library dependencies, putting an extra burden on setup of the visualization workstation. A simple plotter for reviewing price data is as follows:

```
object Plotter {
  def doPlot(series: Array[PriceData]): Figure =
  {
    val min = 0
    val max = series.length

    val rng = Range.Double(min, max, 1).toArray
    val prices = series.map(_.adjClose).toArray

    val f = Figure()
    val p = f.subplot(0)
    p += plot(rng, prices)
    p.xlabel = "Period"
    p.ylabel = "Price ($)"
    f
  }
}
```

Another approach is to produce MS-Excel readable outputs. Every quantitative analyst runs Excel and it provides a lot of charting options. The output in fig. 3 shows an example chart for our co-integration scenario. The variance ratio for this regression was 0.203, which indicated that it had a strong tendency for our desired mean-reversion behavior. The chart verifies this result.

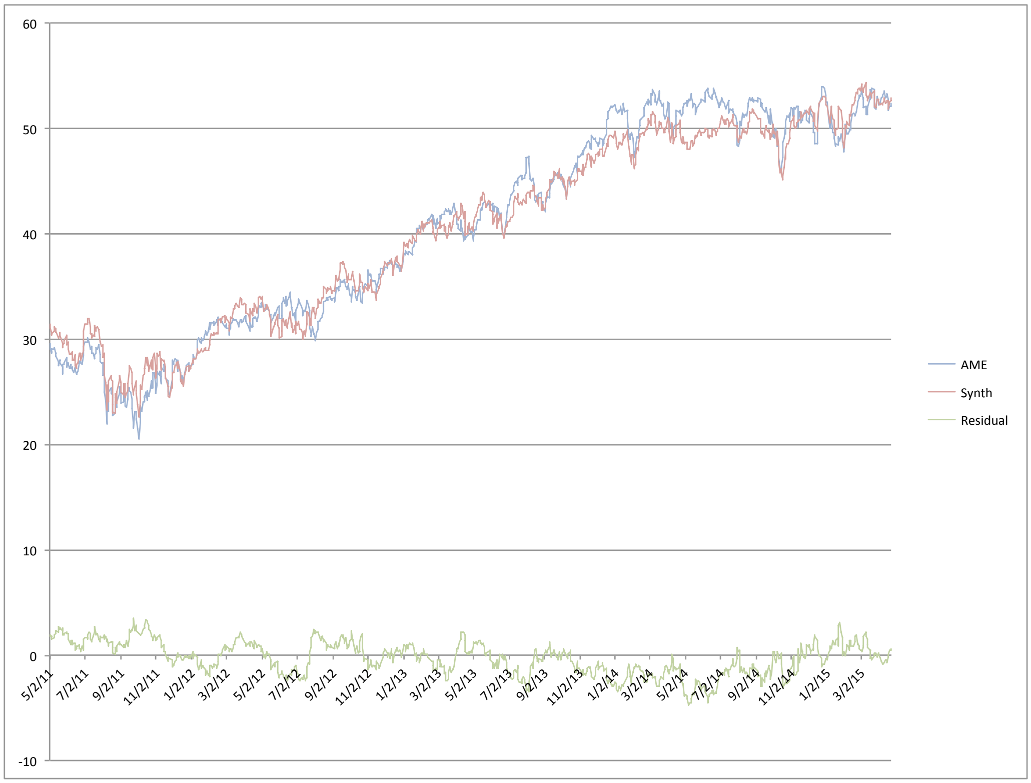

fig. 3 - excel chart example

There are many commercial trading systems – e.g. WealthLab[§] – which provide charting capabilities, including the ability to include technical indicators. It may make sense to integrate our analysis sessions with one of these tools, rather than 'reinventing the wheel'. However, some potential development opportunities are described.

Opportunities for more R&D: Since the analysis sessions should be interactive (keeping the structured data online/available), a web-based interface would be an interesting project. This might be implemented on the Scala Play[**] framework, in conjunction with interactive html5/Angular.js[††] charting component. This type of implementation would also allow us to more easily share the system across a team of quantitative researchers.

Back-testing

Part of our analysis sessions would be to perform trading strategy simulations. The project includes an initial back-testing implementation, showing the pair-trading scenario. Each strategy would be defined by extending the TradingStrategy abstract base class:

```
abstract class TradingStrategy
{
  def getStocksOfInterest : List[String] = ???
  def setStockHistories(minuteData : Map[String, Array[PriceData]]) : Unit = ???
  def evaluate(startAmt: Double) : StrategyResults = ???
}
```

[§] http://www.wealth-lab.com/

[**] http://www.playframework.com/

[††] http://angularjs.org/

This allows the strategy simulation to be implemented without having to know where to get simulation data (i.e. the inversion-of-control pattern[‡‡]). The basics of our trading simulation are shown here:

```
// run through the minute-by-minute price data, detecting position signals and performing trades
for (period <- 0 to tradePeriods - 1) {
  val targetAmt = targetHistory(period).adjClose
  val residual = pairResidual(period)

  if (haveOpenPosition) {
    // looking for when to exit position (think about this one first!)
    if (timeToSell(residual) || timeToStopLoss(residual))
      closePosition(residual)
  }
  else // looking for the start signal
    if (timeToBuy(residual))
      openPosition(residual, targetAmt)
}
StrategyResults(coIntInfo.target, cash + (gain(pairResidual(targetHistory.length-
1))+priceAtBuy)*positionShares, events, winners)
```

The StrategyResults class provides a summary of the simulation – i.e. value of portfolio, number of signals and number of winning trades. For the AME pair trading example the $1,000,000 initial balance became $1,064,985 after the 48 trading days. This is approximately 6.5%, or 36% annually. Over the same period, an index-linked (S&P) fund returned 2.4%, or 12% annually. It should also be noted that the return of AME itself was 0.3% during the same period, showing the potential power of a pairs-trading strategy to provide statistical arbitrage while hedging against (implicit) market factors. Note, however, that there were only 5 trading signals, which is very likely to be sub-optimal in general. Our hypothetical analyst would need to spend more time on the strategy, perhaps implementing a Spark job to iterate over trading parameters.

Opportunities for more R&D: A more robust back-testing framework should include portfolio optimization functions. It may also be worth defining a testing framework that takes the trading algorithm as input (i.e. a function) and executes it without the algorithm having access to actual data or having to traverse the historical data within the algorithm itself (i.e. a sort of "blind taste test" that also avoids coding errors related to interpreting the historical data).

Conclusions

By review of research and practices related to quantitative analysis and through the implementation of a prototype system to support these scenarios, we have shown that Scala and Spark provide an interesting technology suite on which to implement an active portfolio management system. Scala's mix of functional programming and object-oriented concepts allow us to perform our analysis in a concise manner and can help avoid the cryptic use of low-level semantics (such as in matrix manipulation and compounded formulas).

Spark provides an environment in which to run non-trivial analysis scenarios at a higher level of abstraction than base MapReduce. Knowledge of how to break up the problem, so

[‡‡] http://martinfowler.com/bliki/InversionOfControl.html

that it can be run within the distributed cluster on partitioned data, is still required, but the algorithms can be expressed more cohesively without having to explicitly relate them back to mappers and reducers.

Even though Spark provides elegant high-level abstractions, we were still able to provide performance hints, when needed – such as via the use of a custom partitioner and defining operations on local partitions (e.g. mapPartition vs. map).

The project has highlighted one of the points that is emphasized in [4] – "Data is messy, and cleaning, munging, fusing, mushing, and many other verbs are prerequisites to doing anything useful with it." Developing the data preparation component within our prototype was a testament to this point.

The nature of the quantitative analysis topic offers fun, challenging opportunities for further research and for further development of the framework. We have outlined some of these potential follow-on projects within each component area of the system. These include extending the library of functions and classes related to analysis, adding additional data sources and leveraging additional technologies within the Spark solution, such as Streaming and the SparkSQL mapping framework.

References

1. Grinold, Richard C., and Ronald N. Kahn. "Active portfolio management." McGraw-Hill, 2000.
2. Dunis, Christian L., Jason Laws, and Patrick Naïm, eds. "Applied quantitative methods for trading and investment." John Wiley & Sons, 2004.
3. Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on hot topics in cloud computing. 2010.
4. Ryza, Sandy, Laserson, Uri, Owen, Sean and Josh Wills. "Advanced Analytics with Spark." O'Reilly 2015.
5. Karau, Holden, Konwinski, Andy, Wendell, Patrick and Matei Zaharia. "Learning Spark: Lightning-Fast Big Data Analysis." O'Reilly 2015.
6. Odersky, Martin, Lex Spoon, and Bill Venners. "Programming in Scala." Artima Inc, 2008.
7. Odersky, Martin. "Functional Programming Principles in Scala – Coursera." - http://www.coursera.org/course/progfun
8. Balch, Tucker. "Computational Investing, Part I – Coursera." - http://www.coursera.org/course/compinvesting1
9. NetworkError (an alias). "Google's Undocumented Finance API" - http://www.networkerror.org/component/content/article/44-googles-undocumented-finance-api.html , 2013
10. "Amazon Elastic MapReduce" - http://aws.amazon.com/elasticmapreduce/