David P. Stonehouse

University of Maryland University College

DATA 650 – Big Data Analytics

Fall 2016, Section 9040, Professors Gortcheva and Woo

Assignment 4: Spark Analysis of HVAC systems

## HVAC Model Results

The analysis for this question is based on heating, ventilation and air conditioning (HVAC) data. Details of the data are at Mehrotra (2016). The variables in the data are Date, Time, TargetTemp, ActualTemp, System, SystemAge, and BuildingID. The goal of analyzing the data is to develop a model that will predict whether the system will have a temperature reading that is colder or hotter than the target temperature. The appendix shows a listing of the first five rows of the data.

Using the IBM Bluemix Spark application, the HVAC CSV data was loaded into a data storage object. Using Spark “lazy loading”, several map steps and model creation were organized into a pipeline. The date and time columns were dropped. The System and SystemAge values were read in and put into a term frequency hash. That data was used to fit a logistic regression model using the Spark ML library.

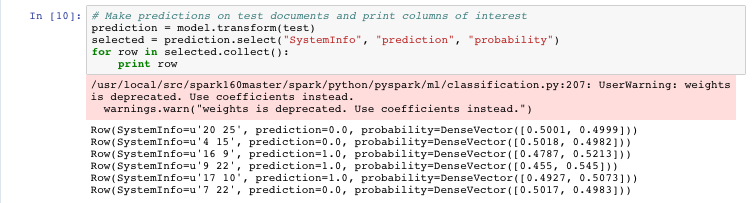


Figure . Listing and output for predicting first set of data

The input data frame for this test had 6 rows. Running the model produced a data frame with 6 rows. The input values to be scored are in the SystemInfo column. Those values are the System and SystemAge, stored together in a string. Similar to the process of producing the model, the SystemInfo values are passed to a Tokenizer to split them into two words. These words are passed to the HashTF function to create hashed values for the input values. These hashed values are then passed to the fitted logistic regression model.

The *prediction* value indicates the prediction from the logistic model, where a value of 1.0 means that the model predicts that the system will be hotter than the target temperature for the system. A value of 0.0 means the system is predicted to be cooler than the target. The *probability* values indicate the probability that the measured value will be cooler or hotter than the target temperature. If the first value in the DenseVector is above .5, then a zero is predicted. If the second value in the DenseVector is above .5, then a one is predicted.

For this set of observations, half of the systems were predicted to be hotter than the target. So, for the first row, the SystemInfo value of “20 25” means the System number is 20 and the SystemAge is 25. The *prediction* is zero, so that system is predicted to be cooler than the target. However, the probability of being cooler is only 50.01%, so really just a coin toss. In contrast, the fourth row, for System 9, 22 years old, is predicted to be hotter with 54.5% probability.

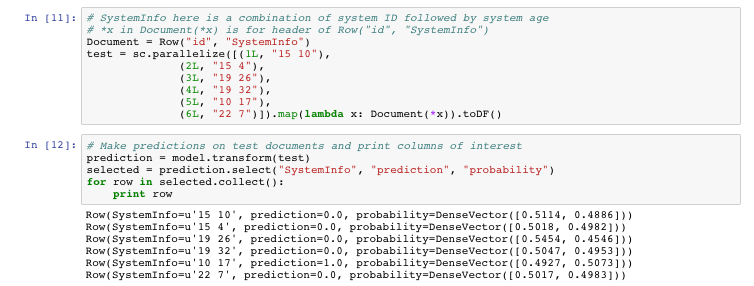


Figure . Listing and output for predicting second set of data

In the second set of results, only one of the observations

From just these two samples, it would seem that the System value has a bigger influence on the prediction. For example, the second set has two pairs of observations for two Systems with a big range of SystemAge values: “15 10”, “15 4”, “19 26” and “19 32”. Also, the first group has two observations with different System values and the same SystemAge value, but the predictions are different: “7 22” and “9 22”.

Logistic regression is suitable for this case. Per Statistics Solutions (2016), logistic regression assumes a number of things. One is that, for binary logistic regression, the target variable is binary. It assumes that P(y=1) is the probability of the event happening. In our case, the target variable is binary, and does represent the label of 1 to mean the event is happening. Only meaningful variables should be included and all meaningful variables should be included. Also, logistic regression requires a large number of samples relative to the number of independent variables. The minimum is 10 cases per independent variable, but often the recommendation is more like 30. The data in this case meets all of these criteria. There are plenty of records (8000) relative to the number of independent variables (2).

An alternative algorithm appropriate for this case is support vector machines (SVMs). The concept of SVM is to divide the observations by a hyperplane, creating the maximum distance between the two groups. SVM can make use of a linear or non-linear (e.g., Gaussian) kernel. SVM with a linear kernel is very similar to logistic regression. Kumar (2015) describes when it is appropriate to use logistic regression, drawing from Andrew Ng’s Machine Learning course. For the HVAC data set, SVM with a Gaussian kernel is appropriate. Logistic regression and SVM with a linear kernel are looking for a linear hyperplane separating two groups, as illustrated in Joglekar (2015). In contrast, SVM with a Gaussian kernel looks to find a non-linear hyperplane between two groups.

Another machine learning algorithm appropriate for this case is decision tree. An advantage of decision tree is that the results are easy to explain. It can also point out interactions between variables that particular indicate one group vs. the other.

## In-memory processing

There are some applications for which in-memory solutions like Spark can replace MapReduce. The first key application is for iterative algorithms. An example is logistic regression using maximum likelihood. Logistic Regression is often done with many iterations through the data, each time MapReduce requires that in between each iteration, data is written to disk. An in-memory solution can speed up this process by only reading data into memory from disk on the first pass. After that, data is read from memory. The difference between the two approaches is illustrated in Figure 3 and Figure 4 from TutorialsPoint, 2015.

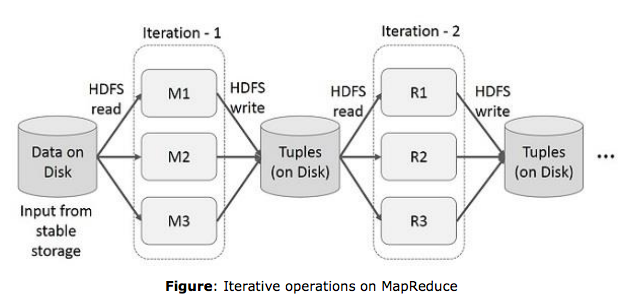


Figure . Data sharing between stages - MapReduce

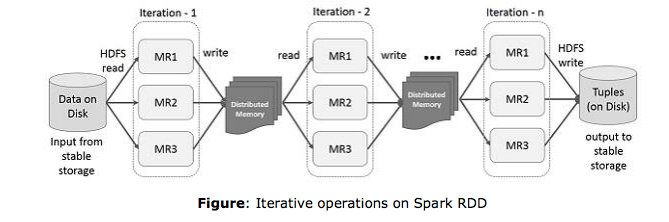


Figure . Data sharing between stages - Spark

Edureka (2015) provides an example of the performance gains between running MapReduce and Spark in-memory (see Figure 5).

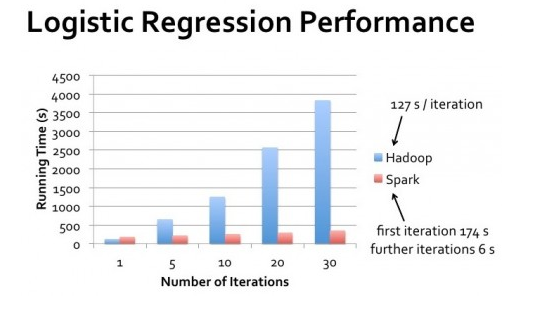


Figure . Performance improvements for an interactive problem using an in-memory solutions

Another key place where in-memory solutions are an improvement over MapReduce is for interactive applications, like queries against big data. For MapReduce, if there are several queries done in a row, each one requires that the data be read from disk. An in-memory solution, when possible, will load the data into memory once, and run the queries against all of the data cached into memory.

McCabe and Wang (2014) describe a relatively new feature in the Hadoop Distributed File System (HDFS), HDFS Read Caching. This feature seems to offer some abilities for Spark users to do more sophisticated caching in memory. This could provide some significant MapReduce speed improvements, especially for interactive use cases.

## Spark, HDFS and Object Storage

**Comparing HDFS and Object Storage**

Both Spark and MapReduce need a way to share data across parallel jobs from one stage to the next. For MapReduce, this is done through HDFS. HDFS suffers because “data sharing is slow in MapReduce due to replication, serialization, and disk IO” (TutorialPoint, 2015). The default mode in HDFS makes two copies of the data. When sending the data, it must be serialized. Serialization means converting from a container (object) to a byte stream. Every time an application needs to share from one stage to next, it involves not only the time to write the block to disk, but also network communication and serialization.

In Spark, sharing from one stage to the next is done through the RDDs and has a lot of flexibility for how it is done. The RDD is configured for sharing through the StorageLevel setting. From the Spark Programming Guide (Apache, 2016), there are numerous options to give flexibility for how to pass data from stage to stage. The MEMORY\_ONLY option means that data is never written to disk. It either fits in memory or is recomputed as needed. The MEMORY\_AND\_DISK option means the data is stored in memory if it fits, but otherwise it is “spilled over” to disk. The DISK\_ONLY option means the RDD / DataFrame will always be written to disk. The MEMORY\_ONLY\_SER and MEMORY\_AND\_DISK\_SER options work like their counterparts, but they use a space efficient format. There is a tradeoff because there is overhead in serializing / deserializing the data. The MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2 and DISK\_ONLY\_2 options are like their counterparts, except a copy of data is stored on a second node. The DISK\_2 option has similar performance characteristics to MapReduce if there is very limited memory on the systems, since Spark will have delays for networking and disk writes.

**Spark and MapReduce**

The MapReduce programming model did not originate with the Hadoop project. It is a concept from functional programming that predates Hadoop. The Message Passing Interface has had a reduce operation in it since the 1990s. As explained by Lu, here are the steps in the programming model:

* Prepare data for the mapper function
* Run user defined mapper code
* Shuffle the mapper results and hand to the reducers
* Run the user-provided reducer code
* Produce the final results

Miner and Shook (2014) provide a useful picture of the flow of MapReduce jobs (Figure 6). The

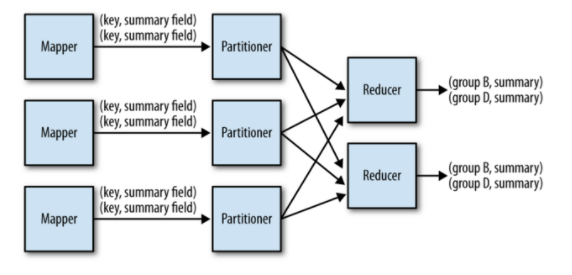


Figure . Flow for Mapper and Reducer jobs

Both Spark and Hadoop MapReduce implement the concepts of that model (see Owen, 2014). The Spark API has a map function which can be used to do filtering and other typical functions done in the Mapper. Both MapReduce and Spark do a shuffle step when there is a need to do a reduce against the keys from the mapping step. An important note is that Spark always writes to disk during the shuffle step (Grishchenko, 2015). This is an important consideration when planning jobs for Spark – they should be done in such a way to minimize the number of times that shuffle will happen.

The API for Spark and MapReduce have different semantics that could cause a problem for Hadoop MapReduce users just starting with Spark. For example, from Owen, 2014, the map() function in Spark is not the same as the functionality in the Mapper. Instead, the mapPartition() will produce a set of results from running a map() transformation on the rows of data for one partition.

Also, Reducers in MapReduce by default will reduce down to values for each key, while the default reduce function for Spark will reduce all rows down to one value. The more analogous call is groupByKey.

**Spark, HDFS and Alluxio**

Spark does not have its own storage mechanism. Distributed storage is accomplished by interacting with storage systems like HDFS and Amazon S3. Originally, Spark only supported HDFS as its distributed storage system. Similarly, HDFS originally only had MapReduce as its key computing framework user, but now supports Spark and others. The situation has become more complex as shown in Figure 7 (Simsa, 2016). Users creating applications can find themselves having to address a large storage options. Similarly, storage providers can find themselves scrambling to keep up with features for all of the different computing frameworks.

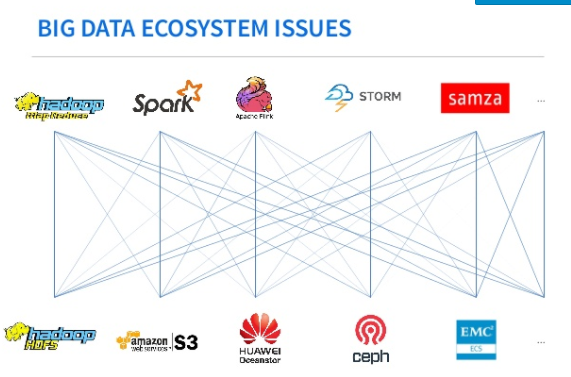


Figure . Big Data Ecosystem issues.

To address this situation of complexity, Alluxio (formerly called Tachyon) was introduced as a memory speed virtual distributed storage system. The intent is to remove dependency between computing frameworks like MapReduce and Spark and the underlying storage systems like HDFS and solutions from cloud providers like Amazon S3 (see Figure 7 from Alluxio, 2016). It also provides value-added services to “facilitate data sharing and locality between jobs” (Alluxio, 2016).

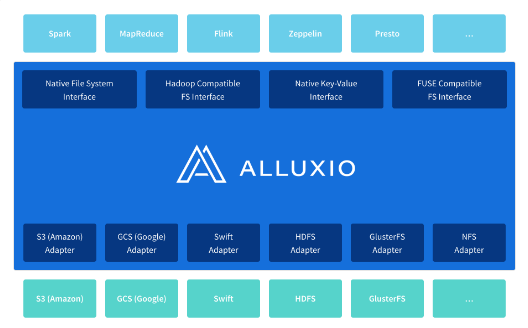


Figure . Alluxio Architecture

## Programming Languages for Spark

A key design goal for the team that supports Spark is that all of the supported languages are treated as “first class citizens” as far as keeping all of them current with the latest features. There is often a small lag between introducing a feature with the Scala API followed by the other APIs, but that lag is getting smaller and smaller. Having said that, users should take into consideration some differences between the languages.

**Convenience in coding**

Scala is unique in that it combines function-oriented programming with object oriented programming. Phatak (2015) highlights some important features of the Scala programming language. A functional programming language is designed to encourage functional programming, where programs are divided into functions that return one value. The name Scala comes from the fact that it was designed to be “scalable”. As part of its design as a functional programming language, a key benefit is support for data that is immutable. Other languages support immutability, but it is a central concept in Scala. Immutability is important for parallelism, because when you have data that can’t be changed, you don’t have to worry about keeping in sync from task to task. You also don’t have to worry about recreating the cache because data was changed underneath you. In contrast, programs that rely on standard database technologies have to suffer overhead for tracking whether the data has changed underneath them.

Other advantages of Scala relate to the fact that it does static typing. Static typing means that the types of variables are known at compile time. It is often more convenient to find errors at compile time instead of runtime. Also, integrated development environments can do more to help programmers with all of the information known from static types.

A benefit of both Scala and Java is that Scala has good integration with Java, since both of them use the Java Virtual Machine. That means that you can “plug and play” libraries built with Java and Scala together in the same process.

For most organizations, there is going to be a convenience among developers who don’t have to do the extra work to learn a new language. That would put things greatly in favor of Java, with Python second, because they are two of the most popular programming languages in use today. New Relic (2016) lists Java as the most requested language in job posts. The same article mentions Scala as the language fastest growing in popularity: job postings are up 50% since 2015. Python job postings are up 13% and Java postings are down 4%.

Gangele (2016) mentions a key drawback when using Java to develop with Spark – “Java does not support [a] REPL(Read-Evaluate-Print Loop) interactive shell”. This type of shell can be valuable when doing some data exploration using Spark. It also can make it easier to try new commands.

Sampathkumar(2016) raises some points in favor of coding with Python. In particular, the power and popularity of the libraries numpy, pandas, matplotlib, seaborn and scikit-learn. There is significant value in using Python to run algorithms on a cluster, then doing visualization or post-processing using these libraries.

**Code size**

Studying a sample of Spark ML classification algorithms in (Apache, 2016b), gives a strong trend in code size between Scala, Java and Python. Python programs are consistently shorter, followed by Scala and then Java. Hicks (2014) gives a series of examples comparing Scala and Java. He gives examples of how Java code is significantly larger than Scala in many cases. The comments section for that blog includes counterexamples using Java 8 where the code size in the two languages seem to be more comparable.

**Performance**

Regarding performance, Xin (2015) shared a few results from benchmarks as of the first part of 2015. Regarding Python, Xin says “When it comes to performance, Python programs historically lag behind their JVM counterparts due to the more dynamic nature of the language.” The gap is closing for Python because pyspark can now run on PyPy, which is a faster version of the Python interpreter/compiler. Starting in 1.3, Spark introduced data frames as an alternative to resilient distributed data sets (RDDs) for working with structured data. Figure 7 shows the results of a performance test with RDDs and data frames using Python and Scala. A similar graphic shown in Spark Summit (2016) shows that the performance using data frames across the languages (Scala, Java, R, and Python) was identica.

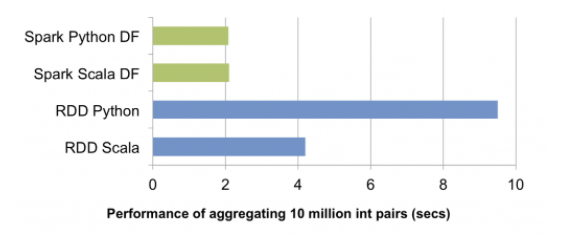


Figure . Performance comparison between RDDs and data frames for Python and Scala

.

## References

Alluxio, 2016, “Alluxio Overview”, retrieved from <http://www.alluxio.org/docs/master/en/index.html>

Apache, 2016a, Spark Progrramming Guide 2.0.2, retrieved from <http://spark.apache.org/docs/latest/programming-guide.html>

Apache, 2016b, “Classification and regression - Spark 2.0.2 Documentation”, retrieved from <https://spark.apache.org/docs/latest/ml-classification-regression.html>

Chambers, B., 2015, “Spark MLLib - Predict Store Sales with ML Pipelines”, retrieved from <http://www.sparktutorials.net/Spark+MLLib+-+Predict+Store+Sales+with+ML+Pipelines>

Damji, J., 2016, “A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets”, retrieved from <https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>

Databaseline, 2015, “An Overview of File and Serialization Formats in Hadoop”, retrieved from <https://databaseline.wordpress.com/2015/12/07/an-overview-of-file-and-serialization-formats-in-hadoop/>

Edureka, 2015, “Apache spark vs. Hadoop map reduce”, retrieved from <http://www.edureka.co/blog/apache-spark-vs-hadoop-mapreduce>

Ganesan, K., 2014. “All About Stop Words for Text Mining and Information Retrieval”. Retrieved from <http://text-analytics101.rxnlp.com/2014/10/all-about-stop-words-for-text-mining.html>

Gangele, J., 2016, “Java vs Scala vs python for big data apache spark project”, retrieved from <https://www.linkedin.com/pulse/java-vs-scala-python-big-data-apache-spark-project-jeetendra-gangele>

Grishchenko, A., 2015, “Spark Architecture: Shuffle | Distributed Systems Architecture”, retrieved from <https://0x0fff.com/spark-architecture-shuffle/>

Hicks, M., 2014, “Java vs. Scala: Why Should I Learn Scala?”, retrieved from <https://www.toptal.com/scala/why-should-i-learn-scala>

Joglekar, S., 2015, “Logistic Regression (for dummies)”, retrieved from <https://codesachin.wordpress.com/2015/08/16/logistic-regression-for-dummies/>

Kumar, A., 2015, “Machine Learning - When to Use Logistic Regression vs. SVM”, retrieved from <http://vitalflux.com/machine-learning-use-logistic-regression-vs-svm/>

Lu, J., “MapReduce Introduction”, retrieved from <http://www.cs.uml.edu/~jlu1/doc/source/report/MapReduce.html>

McCabe, C., and Wang, A., 2014, “New in CDH 5.1: HDFS Read Caching”, retrieved from <http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/>

Mehrotra, N., 2016. “Use Apache Spark to build machine learning applications on HDInsight”, retrieved from <https://azure.microsoft.com/en-us/documentation/articles/hdinsight-apache-spark-ipython-notebook-machine-learning/>

Miner, D. and Shook, A., 2012, MapReduce Design Patterns, O’Reilly Media.

Owen, S., 2014, “How-to: Translate from MapReduce to Apache Spark”, retrieved from <http://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/>

Phatak, M., 2015, “Introduction to Spark Architecture”, retrieved from <https://www.youtube.com/watch?v=65aV15uDKgA>

Sampathkumar, R., 2016, “Is Scala a better choice than Python for Apache Spark?”, <https://www.quora.com/Is-Scala-a-better-choice-than-Python-for-Apache-Spark/answer/Rajesh-Sampathkumar>

Simsa, J., 2016, “Spark Summit EU talk by Jiri Simsa” retrieved from <http://www.slideshare.net/SparkSummit/spark-summit-eu-talk-by-jiri-simsa>

Statistics Solutions, 2016, “Assumptions of Logistic Regression”, retrieved from <http://www.statisticssolutions.com/assumptions-of-logistic-regression/>

TutorialsPoint, 2015, retrieved from <https://www.tutorialspoint.com/apache_spark/apache_spark_tutorial.pdf>

Widman, J., 2016, “The Most Popular Programming Languages of 2016”, retrieved from <https://blog.newrelic.com/2016/08/18/popular-programming-languages-2016-go/>

Xin, R., 2015, “Recent performance improvements in Apache Spark: SQL, Python, DataFrames, and More”, retrieved from <https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html>

Zotti, R., 2013, “How does a Support Vector Machine (SVM) work?” retrieved from <http://stats.stackexchange.com/questions/23391/how-does-a-support-vector-machine-svm-work>

Spark Summit 2016, "Training Apache Spark Essentials”, retrieved from <https://www.youtube.com/watch?v=OheiUl_uXwo&feature=youtu.be>

## Appendix: Supporting Information

**First Rows from the HVAC Data:**

data = sc.textFile("swift://notebooks.spark/HVAC.csv")

data.take(5)

[u'Date,Time,TargetTemp,ActualTemp,System,SystemAge,BuildingID',

 u'6/1/13,0:00:01,66,58,13,20,4',

 u'6/2/13,1:00:01,69,68,3,20,17',

 u'6/3/13,2:00:01,70,73,17,20,18',

 u'6/4/13,3:00:01,67,63,2,23,15']

**R source code:**

Now try with tf-idf

kfit <- DoKMeans(m.tf.idf.transpose2,8)