For assignment C we decided that the initial data ingestion should start with RDD vs Spark’s DataFrames. RDD is as close to Python’s basic data structure as possible. Tweet data is not simple key/values, it has a simple but deep hierarchical layout that is not suited for storage in a DataFrame. Hence, for ingestion, we utilized Spark textFile function to ingest tweet data as strings and to take advantage of RDD’s distributed nature. The ingestion process was slow as Spark had to decompression the files and read the values. To ensure that each executor had enough data to work on at a time we repartitioned the data so that each executor had a reasonably sized chunk. In our case the partition size was selected to be # of executors times 4 (3 – 4 was commonly suggested)

Each bz2 compressed file contained roughly 2,000 JSON documents in a line-delimited format. Each of these JSON documents are pushed through Python’s built-in JSON module to convert it to a dictionary of values using a map function

For cleaning, we ran some preliminary filter functions. Several JSON documents seemed to indicate that it was deleted, hence we assumed that documents with the delete key were to be removed. Another filter function specifically looks for the lang key. Several JSON documents didn’t have this tag and was simply removed. After those two filters, we finally filtered for English tweets (based on the lang key)

This was developed on a Hortonwork’s virtual machine using Spark 2.1.0 and Anaconda’s Python 3.5.3, with 7 cores and 12GB of RAM. Ingestion was a painfully slow process. Therefore, to minimize wait times (ingestion took 4 -5 hours on a 45GB set of compressed JSON files) we ensured we persisted the data after the RDD ingestion. Running count, we retrieved ~37 million JSON documents from archive 09 (our archive 10 was corrupted and we couldn’t read it)

A separate feature generator function was used to create features suitable for DataFrame. This included term frequency counts, mention and hashtag counts, various punctuation counts, and even mentions of money

Due to time constraints and constant crashes, we were only able to test a fit on a Linear Regression. The goal was to associate the number of frequency counts to closing price, and then use the model throughout a trading day to predict closing prices. In our model, we used total term frequency count for a day which would not be suitable for prediction if we had only a partial count. Ideally, we’d normalize the data into a sliding window (e.g. term frequency for the past 1 hour or so) but did not do this as we only had closing prices for a day