

# *Market prediction with Big Data Tools and Technologies*

Team 11: Robin Liu, Frank Li, Daniel Rodriguez

## **Introduction:**

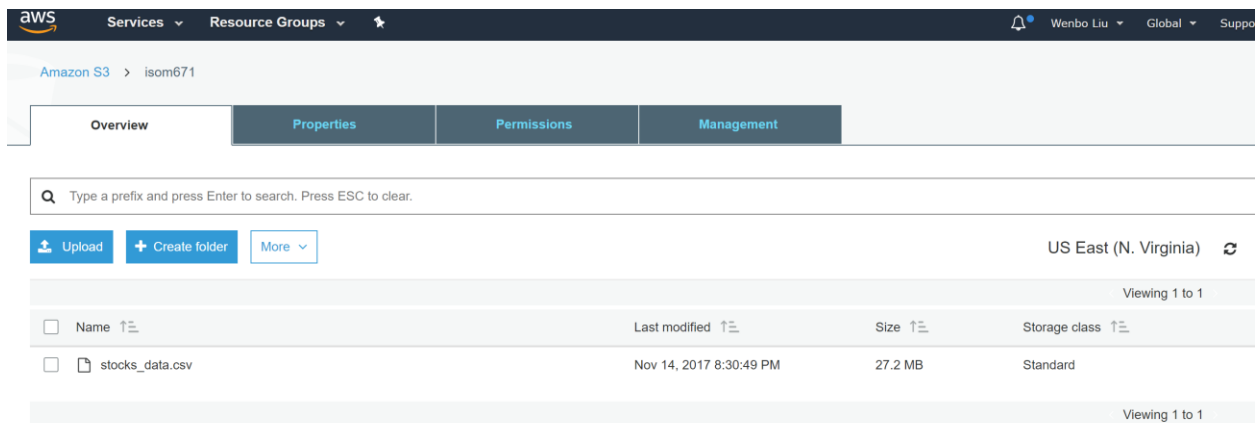
The Big Data Ecosystem, or Big Data Zoo, contains a plethora of tools to collect, store, process, and analyze large amounts of data in a relatively short amount of time. For our Big Data project we wanted to explore a subset of this ecosystem and apply it to real world data in a way that would allow us to utilize a combination of big data tools.

After much deliberation, we ultimately decided on using stock market data as our example data set. Due to the small number of columns and large amount of tuples, this data set was ideal for dipping our toes into new big data tools. Our initial goal was to simply process the dataset using a tool like Hive and compare it to SQL, but we grew more ambitious. Instead, we decided to use R to pull the data from Yahoo's API, use Amazon's S3 service to store the dataset, use Hive to process the data into a table, and load that table onto Spark with Zeppelin to prime it for any analysis one might want to do.

## **Methods:**

1: Pull data Through R package

2: Log on to AWS and upload the file to a S3 bucket



Using putty SSH to connect to the master node and execute

```
aws s3 cp s3://isom671/stocks_data.csv ./
```

Now that we have our data in the terminal, let's create a hql script to create a database called 'stock\_info' and a table called 'stock'. At this point, a lot of skills and experiences we learned in lectures and labs become very handy:

```
[hadoop@ip-172-31-16-161 ~]$ ll
total 27872
-rw-rw-r-- 1 hadoop hadoop      290 Nov 14 03:44 create_table.hql
-rw-rw-r-- 1 hadoop hadoop 28535496 Nov 14 03:21 stocks_data.csv
```

```
[hadoop@ip-172-31-16-161 ~]$ cat create_table.hql
DROP DATABASE IF EXISTS stock_info CASCADE;
CREATE DATABASE stock_info;
USE stock_info;
DROP TABLE IF EXISTS stock;
CREATE EXTERNAL TABLE stock (
open_date STRING,
open DOUBLE,
high DOUBLE,
low DOUBLE,
close DOUBLE,
volume INT,
name STRING)
ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','
```

Using hive -f to run a hiveql script

```
[hadoop@ip-172-31-16-161 ~]$ hive -f create_table.hql
Logging initialized using configuration in file:/etc/hive/conf/hive-default.properties Async: false
OK
Time taken: 1.414 seconds
OK
Time taken: 0.041 seconds
OK
Time taken: 0.027 seconds
OK
Time taken: 0.109 seconds
OK
Time taken: 0.535 seconds
```

And let's have a look at our newly created databases

```
hive> show databases;
OK
default
stock_info
Time taken: 0.014 seconds, Fetched: 2 row(s)
hive> USE stock_info;
OK
Time taken: 0.015 seconds
hive> describe stock;
OK
open_date          string
open               double
high              double
low               double
close             double
volume            int
name              string
Time taken: 0.043 seconds, Fetched: 7 row(s)
```

After we created this database, let's load this database. First of all use

```
hadoop fs -mkdir final_project
```

to create a folder to hold our stock data. Then:

```
[hadoop@ip-172-31-16-161 ~]$ hadoop fs -put stocks_data.csv final_project
[hadoop@ip-172-31-16-161 ~]$ hadoop fs -ls final_project
Found 1 items
-rw-r--r--  1 hadoop hadoop    28535496 2017-11-15 01:42 final_project/stocks_data.csv
```

Call hive again, this time tell it to load our database.


```
hive> USE stock_info;
OK
Time taken: 0.01 seconds
hive> LOAD DATA INPATH 'final_project/stocks_data.csv' INTO TABLE stock;
Loading data to table stock_info.stock
OK
Time taken: 0.732 seconds
```

Try to see what's in our hive system:

```
hive> SELECT * FROM stock_info.stock LIMIT 5;
OK
8/13/2012      92.29  92.59  91.74  92.4    2075391 MMM
8/14/2012      92.36  92.5   92.01  92.3    1843476 MMM
8/15/2012      92.0   92.74  91.94  92.54   1983395 MMM
8/16/2012      92.75  93.87  92.21  93.74   3395145 MMM
8/17/2012      93.93  94.3   93.59  94.24   3069513 MMM
Time taken: 1.59 seconds, Fetched: 5 row(s)
```

Seems like we've successfully loaded our database. Let's move on to setting up zeppelin and spark.

Cluster: My cluster-1 Waiting Cluster ready after last step completed.

Summary	Application history	Monitoring	Hardware	Events	Steps	Configurations	Bootstrap actions
<b>Connections:</b> <a href="#">Enable Web Connection</a> – Hue, Zeppelin, Spark History Server, Resource Manager ... (View All)							
<b>Master public DNS:</b> ec2-54-89-144-232.compute-1.amazonaws.com <a href="#">SSH</a>							
<b>Tags:</b> -- <a href="#">View All / Edit</a>							
<b>Summary</b>				<b>Configuration details</b>			
ID: j-2A9VF0LPQXRHM				Release label: emr-5.9.0			
Creation date: 2017-11-13 13:41 (UTC-5)				Hadoop distribution: Amazon 2.7.3			
Elapsed time: 1 day, 7 hours				Applications: Hive 2.3.0, Pig 0.17.0, Hue 4.0.1, Spark 2.2.0, Sqoop 1.4.6, Zeppelin 0.7.2			
Auto-terminate: No				Log URI: s3://aws-logs-238816346879-us-east-1/elasticmapreduce/ 			
Termination protection: Off <a href="#">Change</a>				EMRFS consistent view: Disabled			
				Custom AMI ID: --			
<b>Network and hardware</b>				<b>Security and access</b>			
Availability zone: us-east-1d				Key name: Robin_N.Virginia			
Subnet ID: <a href="#">subnet-30a69378</a>				EC2 instance profile: EMR_EC2_DefaultRole			
Master: <span style="color: green;">Running</span> 1 m3.xlarge				EMR role: EMR_DefaultRole			
Core: <span style="color: green;">Running</span> 2 m3.xlarge				Auto Scaling role: EMR_AutoScaling_DefaultRole			
Task: --				Visible to all users: All <a href="#">Change</a>			
				Security groups for <a href="#">sg-99a564e9</a> (ElasticMapReduce-Master: master)			
				Security groups for <a href="#">sg-07a06177</a> (ElasticMapReduce-Core & Task: slave)			

As we see, Zeppelin has been added when we created the cluster. We just need to access its interface. There are two ways to do this: establishing an SSH tunnel with the master node using either local port forwarding or dynamic port forwarding. If we choose to do the latter option, we would also need to configure a proxy management tool. Therefore, let's use a local port forwarding method.

**In bash, write down these commands:**

```
ssh -i C:/Users/liuwe/Desktop/Robin_NVirginia.pem -N -L 8157:ec2-54-89-144-232.compute-1.amazonaws.com:8890 hadoop@ec2-54-89-144-232.compute-1.amazonaws.com
```

where:

- -i tells it to read a file as public key
- -N tells it not to do not execute anything
- -L Forwards local port to remote address

what it does is that it establishes a connection between a local port(in our case local port 8157 ) and a port on our master node(in our case 8890, on which Zeppelin interface can be accessed.)

```
MINGW64:/c/Users/liuwe/Google Drive/Emory MSBA/Manage Big Data/Fina...
liuwe@DESKTOP-PI0U5KV MINGW64 ~/Google Drive/Emory MSBA/Manage Big Data/Final Project
$ ssh -i C:/Users/liuwe/Desktop/Robin_NVirginia.pem -N -L 8157:ec2-54-89-144-232.compute-1.amazonaws.com:8890 hadoop@ec2-54-89-144-232.compute-1.amazonaws.com
```

We choose to access port 8890 because that's where our Zeppelin interface resides; there are different ports for other applications if you included them when you set up your cluster. For example, changing '8157' to '8826'(which we randomly selected and is not used by any other application) and '8890' to '8088' will take you to YARN interface.



## All Applications

Logged in as: dr:who

Cluster

About Nodes

Node Labels

Applications

NEW SUBMITTED

NEW SAVING

ACCEPTED

RUNNING

FINISHED

FAILED

KILLED

Scheduler

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
36	0	1	35	1	896 MB	22.50 GB	0 B	1	16	0	2	0	0	0	0	0

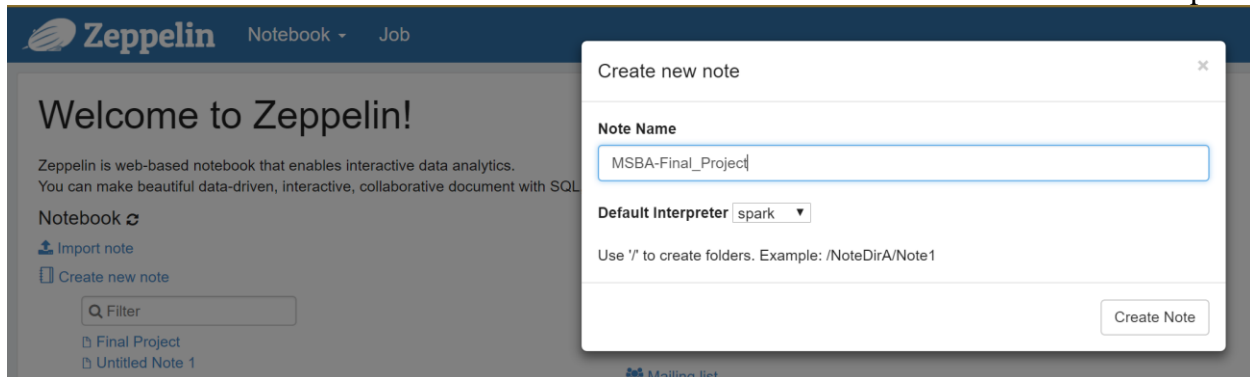
Scheduler Metrics

Scheduler Type		Scheduling Resource Type		Minimum Allocation				Maximum Allocation							
Capacity Scheduler		[MEMORY]		<memory:32, vCores:1>				<memory:11520, vCores:8>							
Show 20 entries															
Search:															
ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes				
application_1510598648567_0040	hadoop	HIVE-d6397792-ba0a-4255-b289-7360be798a2	TEZ	default	Tue Nov 14 20:40:30 -0500 2017	Tue Nov 14 20:45:41 -0500 2017	FINISHED	SUCCEEDED		History	N/A				
application_1510598648567_0039	hadoop	HIVE-3578377d-730d-4afd-7360be798a2	TEZ	default	Tue Nov 14 20:38:22 -0500 2017	Tue Nov 14 20:42:30 -0500 2017	FINISHED	SUCCEEDED		History	N/A				

A complete list of application and their ports can be found on aws website.

Name of interface	URI
YARN ResourceManager	http:// <i>master-public-dns-name</i> :8088/
YARN NodeManager	http:// <i>slave-public-dns-name</i> :8042/
Hadoop HDFS NameNode	http:// <i>master-public-dns-name</i> :50070/
Hadoop HDFS DataNode	http:// <i>slave-public-dns-name</i> :50075/
Spark HistoryServer	http:// <i>master-public-dns-name</i> :18080/
Zeppelin	http:// <i>master-public-dns-name</i> :8890/
Hue	http:// <i>master-public-dns-name</i> :8888/
Ganglia	http:// <i>master-public-dns-name</i> /ganglia/
HBase UI	http:// <i>master-public-dns-name</i> :16010/

Now that we have the access to Zeppelin, let's create a notebook and choose default interpreter to be spark.

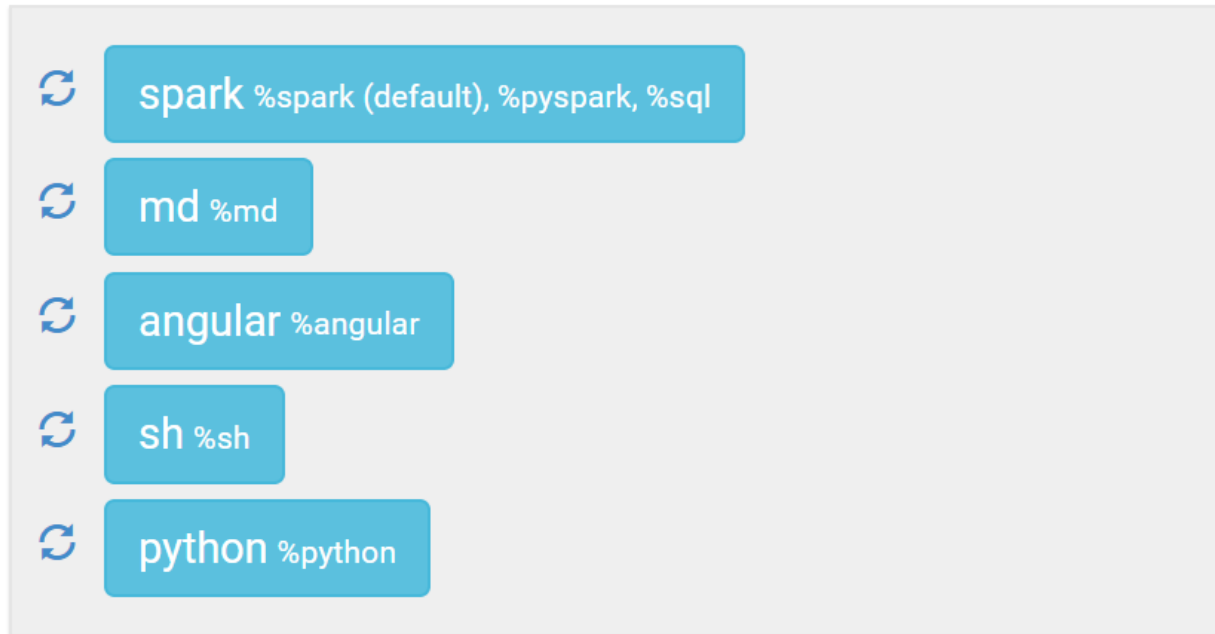


If we want to know what interpreters are being provided by zeppelin:

## Settings

### Interpreter binding

Bind interpreter for this note. Click to Bind/Unbind interpreter. Drag and drop to reorder. The first interpreter on the list becomes default. To create/remove interpreter



As we can see, spark provides a sql shell as well as a python interface, pyspark, which provides python api. One thing EMR does very nicely is that it integrated Hive with Spark. Therefore, when you query in Spark's sql shell, it creates a sqlContext object as well as a HiveContext object which queries Hive for you. This is very important because right now when you want to run your Spark application and do analysis on files, you can translate your text file into Hive (or when you have a database, you can use tools such as sqoop to transform data into hive) and directly run query through Spark SQL as you go. This adds great flexibility into project development, and it reduces cost: all the data can be stored in S3, which provides cheap storage option. Whenever you need to run analysis on the data, you can simply acquire the portion you need and put it into Hive.



The Zeppelin Notebook interface shows a job titled "Final Project". The SQL query executed is:

```
%sql
SHOW DATABASES
```

The results show the following databases:

```
databaseName
default
stock_info
```

The second query is:

```
%sql
SELECT * FROM stock_info.stock LIMIT 9
```

The results are displayed in a table:

open_date	open	high	low	close	volume	name
8/13/2012	92.29	92.59	91.74	92.4	2075391	MMM
8/14/2012	92.36	92.5	92.01	92.3	1843476	MMM
8/15/2012	92.0	92.74	91.94	92.54	1983395	MMM
8/16/2012	92.75	93.87	92.21	93.74	3395145	MMM
8/17/2012	93.93	94.3	93.59	94.24	3069513	MMM

As we see, when we run query in Zeppelin through Spark SQL, our data stored in Hive showed up.

And from this Spark History Server we can see that every time we run Zeppelin, it creates a new SparkContext object with a different AppID.

The Spark History Server interface shows a list of applications. The event log directory is `hdfs:///var/log/spark/apps`. The last updated time is 11/14/2017, 11:38:37 PM.

Search:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
application_1510598648567_0043	Zeppelin	2017-11-15 04:07:17	2017-11-15 04:07:44	27 s	zeppelin	2017-11-15 04:07:44	<a href="#">Download</a>
application_1510598648567_0042	Zeppelin	2017-11-15 04:04:34	2017-11-15 04:06:25	1.8 min	zeppelin	2017-11-15 04:06:25	<a href="#">Download</a>
application_1510598648567_0041	Zeppelin	2017-11-15 04:03:55	2017-11-15 04:04:18	24 s	zeppelin	2017-11-15 04:04:18	<a href="#">Download</a>
application_1510598648567_0019	Zeppelin	2017-11-13 22:57:54	2017-11-15 03:52:56	28.9 h	zeppelin	2017-11-15 03:52:57	<a href="#">Download</a>

Showing 1 to 4 of 4 entries

[Show incomplete applications](#)

And our analysis:

1258

Finished

▶

⌕

⌕

⌕

⌕

▶

⌕

⌕

⌕

⌕

%pyspark

DFAAMZN = spark.sql("SELECT close,volume FROM stock\_info.stock WHERE name == 'AMZN'")

DFAAMZN.count()

1258

Finished

▶

⌕

⌕

⌕

⌕

▶

⌕

⌕

⌕

⌕

Took 9 sec. Last updated by anonymous at November 15 2017, 3:11:45 AM.

%pyspark

from pyspark.ml.regression import LinearRegression

from pyspark.ml.linalg import VectorUDT

from pyspark.sql.functions import udf

from pyspark.mllib.regression import LabeledPoint

### Make this a label/feature vector

data = DFAAMZN.rdd.map(lambda r: LabeledPoint(r[1],[r[0]])).toDF()

#data.show()

### convert from org.apache.spark.mllib.linalg.VectorUDT to ml.linalg.VectorUDT

as\_ml = udf(lambda v: v.asML() if v is not None else None, VectorUDT())

data = data.withColumn("features", as\_ml("features"))

lm = LinearRegression()

model = lm.fit(data,[lr.regParam:50.0])

prediction = model.transform(data)

#prediction.count()

prediction.show()

[ [233.19]|2750998.0| 3558151.657380377|

[ [237.42]|3266819.0|3560323.9976442503|

[ [241.55]|4312712.0|3562444.9823463303|

[ [241.17]|3085900.0| 3562249.831211514|

[ [240.35]|1889992.0| 3561828.715604806|

[ [239.45]|2571911.0|3561366.5155486623|

[ [243.1]|2473886.0|3563240.9935541325|

[ [241.2]|2454445.0| 3562265.237880052|

1259

Finished

▶

⌕

⌕

⌕

⌕

▶

⌕

⌕

⌕

⌕

Took 7 sec. Last updated by anonymous at November 15 2017, 3:14:30 AM. (outdated)

Nowadays, Financial market depends on the ability to process huge volume of data and unearth information from it. Despite we wanted to implement investment strategies mentioned in our proposal, we really wanted to play around with the machine learning library that pyspark provides.

Therefore, we ran a little demo: using volume to predict closing price. Of course, what we did today is quite simple, and this might not be the best predictive model in the financial industry. However, the big data technology set up for more challenging problems could be similar, and we can definitely apply what we learned from this project to run more complex algorithms and analysis in the future.

We also did our original goal: to test our investment strategy. The code below will be on our presentation tomorrow and meant to be flawed: it will have had ran for several hours. The reason it will be very slow is that it is querying Hive many times, and querying in Hive is very time-consuming since it's based on HDFS. The correct way to do the exact same task would query Hive only once and do the analysis in-memory, something spark is excel at, and it would only take seconds. Eventually, out strategy beat the average S&P market return by 2%, though this doesn't take into account any extra costs such as commission fees.

```
%pyspark
change_date = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))-1))
new_date,open,name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))='1')")
change_date.show()
```

-----+-----+  
|new\_date| open |name|  
-----+-----+  
24154	92.9	P000
24155	87.94	P000
24158	101.44	P000
24159	103.39	P000
24160	106.0	P000
24161	104.78	P000
24163	108.37	P000
24164	118.38	P000
24166	119.69	P000
24167	126.76	P000
24172	135.88	P000
24173	139.5	P000
24175	143.41	P000
24176	140.15	P000
24177	141.15	P000
-----+-----+  
Took 0 sec. Last updated by anonymous at November 15 2017, 6:05:23 AM.

```
%pyspark
best_overall = 0.0
worst_overall = 0.0
for i in range(1, len(value)):
    best = np.zeros(10)
    worst = np.ones(10)
    for c in set(cp):
        current = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd')))) AS
new_date, open, name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd'))='1' where new_date= '"+str(value[i])+ "' AND name = '"+c+"'") AS
past = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd')))) AS
new_date, open, name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date, 'MM/dd/yyyy'), 'yyyy-MM-dd'))='1' where new_date= '"+str(value[i])+ "' AND name = '"+c+"'") AS
current_open = sc.parallelize(current.select("open").collect()).map(lambda d: d[0]).collect()
past_open = sc.parallelize(past.select("open").collect()).map(lambda d: d[0]).collect()
if(len(current_open)==0 or len(past_open)==0):
    pass
else:
    current_open = float(current_open[0])
    past_open = float(past_open[0])
    ratio = (current_open-past_open)/past_open
    best_min_index = np.argmin(best)
    worst_max_index = np.argmax(worst)
    if(best[best_min_index]<ratio): best[best_min_index]=ratio
    elif(worst[worst_max_index]>ratio): worst[worst_max_index]=ratio
ave_best = np.mean(best)
ave_worst = np.mean(worst)
best_overall = (best_overall+ave_best)/2
worst_overall=(worst_overall+ave_worst)/2
```

RUNNING 0% 

Speaking of Future:



With these two technologies, we can automate most of our work: Using spark stream real-time market data through API and run analysis, and dump everything to S3 bucket once we finish with our analysis.

Last but not least, THANK YOU so much for an amazing class, we enjoyed your class, we learned a lot of exciting things, and we are looking forward to seeing you next semester's machine learning II!

# THE END