Everyone Plays

Collaborative Data Science Using Apache Zeppelin
Preamble

• This presentation is meant to be given in Notebook form inside Zeppelin.

• The actual notebooks are available at
  - [https://github.com/rawkintrevo/apachecon](https://github.com/rawkintrevo/apachecon)
  - Along with instructions for importing them into Zeppelin.

• It’s good to have some slides of screen shots though for if/when my Notebook goes off the rails due to something out of anyone’s control.

• SO.....
Everyone Plays: Collaborative Data Science with Apache Zeppelin

trevor grant, aka @rawkintrevo

Overview
Chapter 1. What is Apache Zeppelin?
Chapter 2. Collaboration Among Apache Products
Chapter 3. Intra-Organizational Collaboration
Chapter 4. Questions

Chapter 1. What is Apache Zeppelin?!

- Paragraph: Web-based interpreter for a whole ecosystem of Big Data Products
- Dynamic frameworks and ‘pre-built’ visualizations for data exploration.
- Extensible:
  - Integration with AngularJS for pretty demos (beyond pre-packaged).
  - Connects to non-Apache products via JDBC interpreter (or home-brew your own)
- Resource pooling: allows for variable sharing between interpreters. More on this in Chapter 2
- Version control and cron jobs built in.
- And more...
### Zeppelin Pivot Tables and Dynamic Forms

Zeppelin's `Stable` interface provides a pivot-table-esque interface for viewing various types of charts.

Dynamic Forms allow for user interactivity.

By default, tables are limited to 1000 results. For better interactivity, we can combine the two.

---

#### Zeppelin Pivot Table

```sql
select * from bank
```

<table>
<thead>
<tr>
<th>age</th>
<th>job</th>
<th>marital</th>
<th>education</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>unemployed</td>
<td>married</td>
<td>primary</td>
<td>1,787</td>
</tr>
<tr>
<td>33</td>
<td>services</td>
<td>married</td>
<td>secondary</td>
<td>4,789</td>
</tr>
<tr>
<td>35</td>
<td>management</td>
<td>single</td>
<td>tertiary</td>
<td>1,350</td>
</tr>
<tr>
<td>30</td>
<td>management</td>
<td>married</td>
<td>tertiary</td>
<td>1,476</td>
</tr>
<tr>
<td>59</td>
<td>blue-collar</td>
<td>married</td>
<td>secondary</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>management</td>
<td>single</td>
<td>tertiary</td>
<td>747</td>
</tr>
<tr>
<td>36</td>
<td>self-employed</td>
<td>married</td>
<td>secondary</td>
<td>307</td>
</tr>
<tr>
<td>30</td>
<td>technician</td>
<td>married</td>
<td>secondary</td>
<td>147</td>
</tr>
</tbody>
</table>
### Zeppelin Pivot Table

```sql
select * from bank
```

<table>
<thead>
<tr>
<th>age</th>
<th>job</th>
<th>marital</th>
<th>education</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>technician</td>
<td>married</td>
<td>secondary</td>
<td>4,073</td>
</tr>
<tr>
<td>37</td>
<td>admin.</td>
<td>single</td>
<td>tertiary</td>
<td>2,317</td>
</tr>
<tr>
<td>25</td>
<td>blue-collar</td>
<td>single</td>
<td>primary</td>
<td>-221</td>
</tr>
<tr>
<td>31</td>
<td>services</td>
<td>married</td>
<td>secondary</td>
<td>132</td>
</tr>
<tr>
<td>38</td>
<td>management</td>
<td>divorced</td>
<td>unknown</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>management</td>
<td>divorced</td>
<td>tertiary</td>
<td>16</td>
</tr>
<tr>
<td>44</td>
<td>services</td>
<td>single</td>
<td>secondary</td>
<td>106</td>
</tr>
<tr>
<td>44</td>
<td>entrepreneur</td>
<td>married</td>
<td>secondary</td>
<td>90</td>
</tr>
<tr>
<td>26</td>
<td>housemaid</td>
<td>married</td>
<td>tertiary</td>
<td>543</td>
</tr>
</tbody>
</table>

Results are limited by 1000.
Time: 0 seconds

### Dynamic Pivot Table

```sql
select $(groupBy),
$(func)$(metric) as $(metric),
count(job) as job,
count(marital) as marital,
count(education) as education
from bank
$((filterString))
group by $(groupBy)
order by $(metric) $(sort)
```
Dynamic Pivot Table

```
-- make sure you've run zeppelinTutorial first!

select $groupBy1, $rollup($metric) as $metric, count($label) as $label, count($label) as marital, count($label) as education
from table
{$filterString}
group by $groupBy1
order by $metric

-- $groupBy1 = job, marital, education
-- $metric = balance, sum(balance)
-- $rollup = sum, avg
-- $filterString = where marital = "divorced"
-- $sort = asc, desc
```

<table>
<thead>
<tr>
<th>FilterString</th>
<th>metric</th>
<th>groupBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>where marital = &quot;divorced&quot;</td>
<td>balance</td>
<td>job</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sort</th>
<th>balcony</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>count</td>
</tr>
</tbody>
</table>

All fields:
- job
- balance
- marital
- education

Keys
- job

Groups
- marital

Values
- balance SUM
### Filter String
```
where marital = "divorced"
```

### Metric
- `balance`

### Group By
- `job`

### Sort
- `asc`

### All Fields
- `job`
- `balance`
- `job`
- `mental`
- `education`

### Keys
- `job`

### Values
- `balance` (SUM)

### Graph
- Bar chart showing the distribution of job categories with their counts.

<table>
<thead>
<tr>
<th>Job Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>management</td>
<td></td>
</tr>
<tr>
<td>blue-collar</td>
<td></td>
</tr>
<tr>
<td>technician</td>
<td></td>
</tr>
<tr>
<td>admin.</td>
<td></td>
</tr>
<tr>
<td>services</td>
<td></td>
</tr>
<tr>
<td>retired</td>
<td></td>
</tr>
<tr>
<td>self-employed</td>
<td></td>
</tr>
<tr>
<td>entrepreneur</td>
<td></td>
</tr>
<tr>
<td>unemployed</td>
<td></td>
</tr>
<tr>
<td>housemaid</td>
<td></td>
</tr>
<tr>
<td>student</td>
<td></td>
</tr>
<tr>
<td>unknown</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1. What is Apache Zeppelin?!

- Paragraph-Web based interpreter for a whole galaxy of Big Data Products
- Dynamic forms and 'pre built' visualizations for data exploration.
- Extendable!
  - Integration with AngularJS for pretty demos (beyond prepackaged).
  - Connects to non-Apache products via JDBC interpreter (or home-brew your own)
- ResourcePool, allows for variable sharing between interpreters. More on this in Chapter 2.
- Version control and cron jobs built in.
- And more...
The Variety of Apache Experiences

The importance of contra-Resume-Driven-Design

Good Data Scientist

The 'Good' Data Scientist. In another life this person might have been (more appropriately) called an Analyst, or possibly a fresh undergraduate with a degree in something STEM-esque (read Economics or Quantitative Business). The Good Data Scientist has a basic understanding of the principals of analytics and charting, possibly even one advanced skillset that sets them apart such as PowerPivots, or knowing a little R.

Magical Allegory

The Good Data Scientist has a Magic 8 Ball that is always right. Given the problem of winning the lottery they ask questions like:

- Will I win the lottery?
- Will I win the lottery this week?
- Should I buy a lottery ticket?

An experienced Good Data Scientist may even cleverly ask:

- Will I win the Illinois Pick 3 this week?
- Is the first winning number 0?

The Paradox of Big Data

The Good Data Scientist is a good person to ask for advice. When it comes to winning the lottery however, the time it will take her to figure out the winning lottery numbers, exceeds the time between lotteries. The Good Data Scientist lacks the tools/knowledge to tackle Big Data Problems.

Better Data Scientist

The 'Better' Data Scientist. This Data Scientist might be fresh out of a mathematics/computer science graduate program or came from another STEM discipline but an auto-diabetic who learned the ways of the Bayesian on her own. She is strong in at least one of the major languages, and well versed in the machine learning libraries associated. She also has some exposure to a big data platform or two.

Magical Allegory

The Better Data Scientist has used a Magic 8 Ball in the past and still will employ it where appropriate. She also has Magic 8 Oke, can read tea leaves.

- She asks her Magic 8 Ball: "Will I win the lottery?"
- She reads her tea leaves to know the shape of the municipality where she should play the lottery.
- She asks, "how many numbers?", and rolls her magic dice.
- She says, "the first number is", and rolls again.

The Right Tool for the Right Job

The Better Data Scientist is closer to the true meaning of the title. The reason for wildly varying salaries of Data Scientists is directly proportional to the various flavors of witchcraft they know. The more tools at their disposal the more effective they are. The Better Data Scientist has a bag of tricks from which they draw and apply the correct tools for the problem at hand.

Data Shaman

The Data Shaman. Oh you haven’t heard of us? We’re very rare and don’t want our titles to get diluted, please don’t call your intern one. Advanced Degrees in Mathematics/Statistics/Computer science, and extensive study across all disciplines. The Data Shaman can speak intelligently about the advantages and drawbacks to many different machines learning libraries and Big Data platforms. Often consults source code before docs to discover hidden magic—she will implement own algorithms in best suited environment if not already available.

Magical Allegory

The Data Shaman might be a headmaster at Hogwarts with a particular specialty but very experienced in multiple disciplines of magic and potion making.

- She reflects on advantages and drawbacks of different approaches.
- She consults online documentation and source code.
- Implements prepackaged solution, deep magic, heavy wizardry, or invents new potions as required.

Make it up as you go

The Data Shaman by definition prefers the path of least resistance, and embodies the lazy programmer adage. The Data Shaman needs an environment that not only puts a wide portfolio of useful products at her disposal, but also that is extendable and supports her doing weird things when the situation calls for it.

Todo: Show ResourcePools notebook.
Everyone Plays: Collaborative Data Science with Apache Zeppelin

about the author

- trevor grant, aka @hawkintrevo

Collaboration Among Apache Products

Flink Scala-Shell

**Apache Flink** is an open source platform for distributed stream and batch data processing.

**Flink’s core** is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams.

Flink includes **several APIs** for creating applications that use the Flink engine:
1. **DataStream API** for unbounded streams embedded in Java and Scala, and

Spark Scala, Python, R, and SQL Shelles

**Spark**

Lightning-fast cluster computing

Latest News
Spark Summit (June 6, 2016, San Francisco) agenda posted
(Apr 17, 2016)

Alluxio (Tachyon) Shell

Alluxio, formerly known as Tachyon, is a memory speed virtual distributed storage
Performance is key. Consistency is a must.

 Providing low latency, high concurrency data management solutions since 2002.
 Build high-speed, data-intensive applications.

 Connect to anything with JDBC and a Jar

 Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which client programs can use to access databases. It is part of the Java Standard API from Oracle Corporation. It provides methods to query and update database records. A JDBC bridge enables connections to any ODBC-accessible database.

 ...or hell, just connect to anything.

 Zeppelin Interpreter
 Zeppelin Interpreter is a language backend. For example, to use Scala code in Zeppelin, interpreterGroup is a unit of start/stops interpreter. Interpreters in the example, SparkSQL interpreter can reference Spark interpreter to get SparkContext from SparkSession.
GoogleViz

```
library(googleVis)
bubble <- gvvisBubbleChart(Fruits, idvar="Fruit",
                         xvar="Sales", yvar="Expenses",
                         colorvar="Year", sizevar="Profit",
                         options=list(
                              hAxis=\'{minValue:75, maxValue:125}\')
)
print(bubble, tag = 'chart')
```

GoogleViz

```
library(googleVis)
geo <- gvvisGeoChart(Exports, locationvar = "Country", colorvar="Profit", options=list(Projection = "eckaraysky-v11"))
print(geo, tag = 'chart')
```
Everyone Plays: Collaborative Data Science with Apache Zeppelin

trevor grant, aka @hawkintrevo

Zeppelin ResourcePool for sharing objects between interpreters

The Spark Family of Interpreters (Spark, SQL, PySpark, R) have access to the ResourcePool via the ZeppelinContext e.g. `z.put(varName) / z.get(varName)`. Other interpreters must be more explicit in accessing.

Flink explicitly accesses the ResourcePool

```scala
// For flink we have to declare the resource pool
val resourcePool = InterpreterContext.get().getResourcePool()

// Import the resource pool
import org.apache.zeppelin.interpreter.InterpreterContext
import org.apache.zeppelin.resource.ResourcePool

// Set the resource pool
resourcePool.put("foo", "bar")
```

Flink puts something in the ResourcePool

```
val resourcePool = InterpreterContext.get().getResourcePool()

foo: String = bar
```

Spark accesses via Zeppelin Context

```
Spark

// Spark has some built in wrappers. Let's get "foo" from the resource pool
println("foo is: " + z.get("foo"))

foo is: bar
```

Any serializable object (not just strings) can be placed in the resource pool.
Any serializable object (not just strings) can be placed in the resource pool

Spark puts a List in the ResourcePool

```scala
Kapark
val foo2 = ressourcePool.get("foo2").get().getInstanceOf[List[Int]]()
p = println("foo2 = ", foo2)
val foo3 = foo2.map(x => x + 1)
resourcePool.put("foo3", foo3)
foo2: List[Int] = List(1, 2, 3)
foo2 is List[Int] = true
foo3: List[Int] = List(2, 3, 4)
```

Get variable from ResourcePool, put updated back in pool

```scala
GetLink
val foo2 = ressourcePool.get("foo2").get().getInstanceOf[List[Int]]()
p = println("foo2 = ", foo2)
val foo3 = foo2.map(x => x + 1)
resourcePool.put("foo3", foo3)
foo2: List[Int] = List(1, 2, 3)
foo2 is List[Int] = true
foo3: List[Int] = List(2, 3, 4)
```

pySpark too!

```python
import time
df = spark.createDataFrame([(i*2, i) for i in range(1000)], ("a", "b"))
```

Traceback (most recent call last):
File "file", line 29, in 
ImportError: No module named pyspark
pyspark is not responding
Everyone Plays: Collaborative Data Science with Apache Zeppelin

trevor grant, aka @rakintrevor

ETL (Extract-Transform-Load)

Download Data [Disabled]

Peek at Data [Disabled]

Spark ETL

```java
T111 = load Divey_Trips_2015_Q4.csv
```
Spark ETL

```scala
import java.sql.Timestamp
import java.text.SimpleDateFormat
scoval
val ds = sc.textFile("hdfs://.../divvy_data/*")
scoval
val tripdataRdd = sc.textFile("hdfs://.../divvy_data/*").mapPartitionsRDD[1] at textFile at <console>:31
```

Exploratory Pivot Chart

```sql
SELECT
    FROM_station_id, TO_station_id,
    COUNT(trip_id) as trips,
FROM
    (SELECT trip_id,
    FROM_station_id, TO_station_id, gender, user_type, hour_of_day, day_of_week
    FROM
        divvy_trips
    WHERE
        day_of_week IN ('Sun', 'Sat')
    AND user_type IN ('Subscriber', 'Subscriber')
    GROUP BY gender, user_type, day_of_week, hour_of_day
    ORDER BY trip_id ASC)
```
Business Problem: Overtime Trips

Background: Divvy charges customers an overtime fee if the trip lasts longer than 30 minutes.

Task: We want to be able to predict when a customer is going to rent longer than 30 minutes so that we might send them an alert when they check out the bike reminding them of the 30 minute time window.

Conditions: Because people don’t (shouldn’t) be on their phone while riding their bike, we need to send this message as soon as possible. We are hoping to take advantage of Flink’s lower latency / higher throughput put for our production system, so while exploratory analysis may be done in Spark, we would like a solution utilizing Flink.

Approach

First, we make a pivot chart for Spark to do some exploratory analysis. We’ll use screen shots of interesting findings for sales, and export the entire chart as a ‘working toy’ for the sales reps to use in the pitch.

Second, we’ll ETL data into Flink and run a multi-layer perceptron on it. Our reasoning for doing this is that this entire example is completely contrived anyway and we want to show how dependencies can be loaded in Flink.

Finally, we’ll try to come up with some additional visualization that add questionable business value but looks super sexy to once again support sales guys and because our bonuses are calculated based on our sizzle factor. If I had to guess, I’d say it’s going to be a community-detection thing feeding into a OSI.

Deep Dive Pivot: Overtime trips

```sql
SELECT
FROM station_id, gender, userType, hourOfDay, dayOfWeek, count(trip_id) as trips
FROM
(SELECT trip_id, hourOfDay, dayOfWeek, gender, userType, (hour * 60 + day) as dayOfYear,
    "date_format(starttime, 'EEE')" as dayOfWeek -- only in 1.5
FROM divvy_trips
WHERE dayOfWeek IN ('Wed', 'Thu', 'Fri', 'Sat')
-- and userType IN ('Customer', 'Subscriber', 'Subscriber:resident')
GROUP BY gender, userType, dayOfWeek, hourOfDay, from_station_id, to_station_id
HAVING trips > 1
--order by stopTime asc
```

DayOfWeek
- Sun
- Settings

UserType
- Customer
The image shows a data visualization with a graph and some SQL query text. The graph appears to represent data distribution over time, possibly related to user type and gender, with different color areas indicating varying user categories. The SQL query text is partially visible, showing a structured query language (SQL) query with variables and conditions for data filtering and aggregation.
What we want out of this dataset for our Flink purposes:

- Day of Week of the start time
- Hour of Day of the start time
- From Station
- Member Type
- Gender

Label:
- Trip Duration

Later on for Graph Processing in Gelly, we'll want to reload the data set as edges and the stations as nodes.

Flink ETL

```java
val divvyDataFormat = new SimpleDataFormat("yyyy-MM-dd\thh:mm:ss")

def maleTimestamp(ts: String) = (new java.sql.Timestamp(divvyDataFormat.parse(ts).getTime()))

val rides = new JavaTextFile("file://home/trevor/git/data/divvy_data")
val maleRides = rides.map(line => line.split(",")).filter(line => line(0) != "trip_id")
val ridesLV = maleRides.map(_.length == 12)
  .map1 => LabelVector(1)(0).toInt match {
    case 0 => 1.0
    case _ => 0.0,
    // ride over 30 minutes
    SparseVector(207861) => 1.0,
```
Machine Learning

(The Point: Just because Zeppelin is pretty, doesn’t mean it’s weak. You can do just as much weirdo stuff here as you can at the CLI)

We’re using a custom machine learning jar with some as-of-yet unmerged features such as

- train-test-splitting
- multi-layer perception
- warm-starts
- evaluation framework

Punchline This code won’t work unless you’ve compiled a FlinkML jar with the appropriate pull requests merged in.

Dark Magic

```scala
import org.apache.flink.ml.optimization.SimpleGradientDescent
import org.apache.flink.ml.optimization.LearningRateSchedule

// These are not officially supported features. Use them in production, and may ___ have mercy on your soul.
import org.apache.flink.ml.preprocessing.Splitter
import org.apache.flink.ml.mlbase.RegressionEvaluator

val data: Array[Array[Double]]
val model = new GradientDescentClassifier
val accuracies = model.run(data)
```

Stop and show the folks how easy this is, time-permitting.

Task 2 seconds (updated)
Questions?