AAS_CH8: Geospatial and Temporal Data Analysis on New York City Taxi Trip Data

Alexander Spivey

Startup/Setup

- Start Spark shell like this:

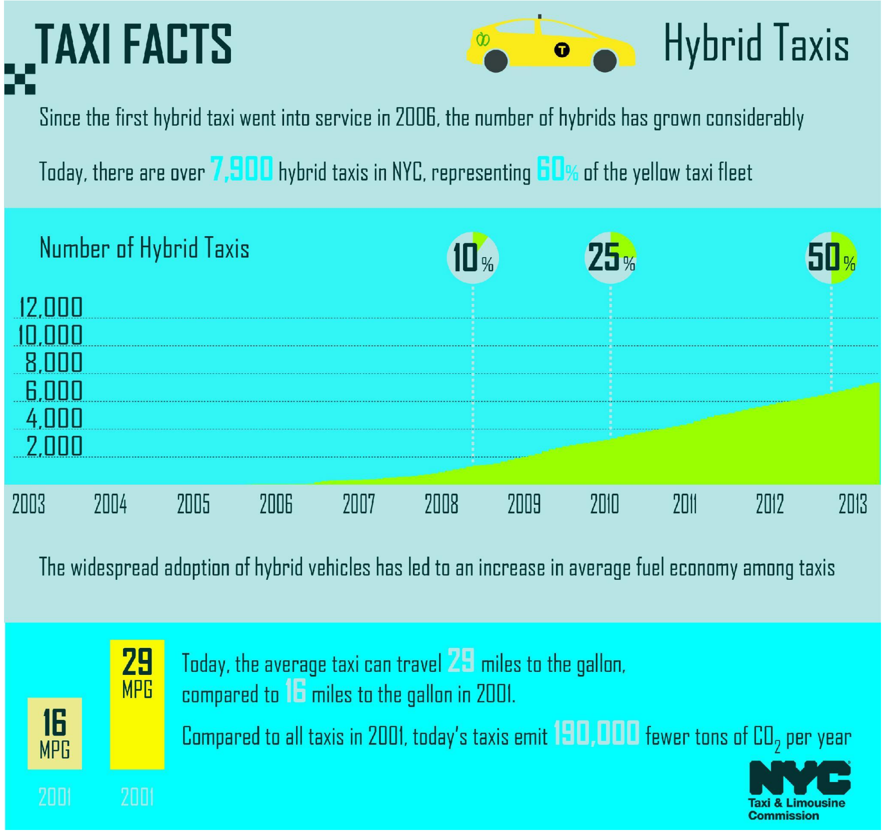
spark-shell --master local[*] --driver-memory 4g --jars /proj/cse398-498/course/aas/ch08-geotime/target/ch08-geotime-2.0.0-jar-with-dependencies.jar

- You'll also need GEOJSON file for NYC boroughs:

/proj/cse398-498/course/AAS_CH8/nyc-boroughs.geojson

The Release of the Dataset

- New York City Taxi and Limousine Commission
  - Shared infographic March 4th 2014
- Chris Wong files FOIL request
  - Freedom of Information Law
- Chris receives 2 500gb drives
  - Releases to public
- Proves riding subway from 4-5pm



Purpose & Dataset

- Important statistic would be utilization
    - Amount of time/duration of usage
- Temporal data, Dates/time, and Geospatial information
- Spark's capability rise, still difficult
    - Java 8 - java.time
    - UDFs
    - Additional libraries

Getting the Data

- Each row represents a single taxi ride
  - hashed version of the medallion number
  - hashed driver hack license
  - temporal information about trip start/end
  - longitude/latitude coordinates of dropoff/pickup

medallion	hack_licen	vendor_id	rate_code	store_and	pickup_datetime	dropoff_datetim	passenger	trip_time_	trip_dista	pickup_lo	pickup_lat	dropoff_lo	dropoff_latitude
89D227B6	BA96DE41	CMT	1	N	1/1/2013 15:11	1/1/2013 15:18	4	382	1	-73.9782	40.75798	-73.9898	40.75117
0BD7C8F5	9FD8F69F	CMT	1	N	1/6/2013 0:18	1/6/2013 0:22	1	259	1.5	-74.0067	40.73178	-73.9945	40.75066
0BD7C8F5	9FD8F69F	CMT	1	N	1/5/2013 18:49	1/5/2013 18:54	1	282	1.1	-74.0047	40.73777	-74.0098	40.726
DFD2202E	51EE87E32	CMT	1	N	1/7/2013 23:54	1/7/2013 23:58	2	244	0.7	-73.9746	40.75995	-73.9847	40.75939
DFD2202E	51EE87E32	CMT	1	N	1/7/2013 23:25	1/7/2013 23:34	1	560	2.1	-73.9763	40.74853	-74.0026	40.74787
20D9ECB2	598CCE5B	CMT	1	N	1/7/2013 15:27	1/7/2013 15:38	1	648	1.7	-73.9667	40.76425	-73.9833	40.74376
496644932	513189AD	CMT	1	N	1/8/2013 11:01	1/8/2013 11:08	1	418	0.8	-73.9958	40.74398	-74.0074	40.74434
0B57B963	CCD4367B	CMT	1	N	1/7/2013 12:39	1/7/2013 13:10	3	1898	10.7	-73.9899	40.75678	-73.8653	40.77063
2C0E91FF	1DA2F654	CMT	1	N	1/7/2013 18:15	1/7/2013 18:20	1	299	0.8	-73.9801	40.74314	-73.9827	40.73534
2D4B95E2	CD2F522E	CMT	1	N	1/7/2013 15:33	1/7/2013 15:49	2	957	2.5	-73.9779	40.78698	-73.9529	40.80637
E12F6AF9	06918214	CMT	1	N	1/8/2013 13:11	1/8/2013 13:19	1	477	1.3	-73.9825	40.77317	-73.9641	40.77382

Working with Third-Party Libraries in Spark

- Foundation of Java - astronomical code available
    - Quality differs drastically
- Must be Serializable Interface (or Kyro)
- Least amount of dependencies
- Don’t use Java-oriented design patterns
- Some reduce boilerplate & increase scalability

hand. For example, in the following Java class representing a pet, almost all the code is boilerplate except for the declarations of Pet, name and owner:

```
public class Pet {
    private PetName name;
    private Person owner;

    public Pet(PetName name, Person owner) {
        this.name = name;
        this.owner = owner;
    }

    public PetName getName() {
        return name;
    }

    public void setName(PetName name) {
        this.name = name;
    }

    public Person getOwner() {
        return owner;
    }

    public void setOwner(Person owner) {
        this.owner = owner;
    }
}
```

Geospatial Data with the Esri Geometry API and Spray

- Geospatial data 2 major kinds: vector and raster.
- longitude/latitude and vector in GeoJSON
    - represents boundaries boroughs within New York
- Library exists for GeoJSON to Java Object
    - Lack library to do spatial analyzation
- Use ESRI Geometry API to parse subset
    - Need to clean data
    - Make new scala function to parse all

Exploring the Esri Geometry API

- Core data: Geometry - shape and geolocation.
- Esri library can compute (within Geometry Engine):
  - are of geometry
  - whether two overlap
  - compute the geometry of overlap
  - Contains operation
- Geometry object represent drop offs and boroughs
  - Is point in a borough?
- The contains operation
  - 3 arguments, two Geometry, one SpatialReference
- SpatialReference used is WKID 4326, coord system for GPS

```
Following the naming convention, we are going to add some helper methods.
*/
import com.esri.core.geometry.{GeometryEngine, SpatialReference, Geometry}
import scala.language.implicitConversions

class RichGeometry(val geometry: Geometry,
    val spatialReference: SpatialReference =
      SpatialReference.create(4326)) {
  def area2D() = geometry.calculateArea2D()

  def contains(other: Geometry): Boolean = {
    GeometryEngine.contains(geometry, other, spatialReference)
  }

  def distance(other: Geometry): Double = {
    GeometryEngine.distance(geometry, other, spatialReference)
  }
}
//make it so that it implicitly converts all Geometry to RichGeometry
object RichGeometry {
  implicit def wrapRichGeo(g: Geometry) = {
    new RichGeometry(g)
  }
}
//import this implicit function
import RichGeometry._
```

Intro to GeoJSON

- Boundaries for boroughs are in GeoJSON format.
  - core object is called a feature
    - geometry instance and properties.
  - A set of features is called a FeatureCollection
- Esri will parse Geometry objects
  - No id or properties
- Use Spray - convert Scala object to JsValue
- Convert string to parseJson then to Scala.
- Create a class to hold GeoJSON features
  - Each JSON object references its own attribute.

```scala
case class Feature(
    val id: Option[JsValue],
    val properties: Map[String, JsValue],
    val geometry: RichGeometry) {
  def apply(property: String) = properties(property)
  def get(property: String) = properties.get(property)
}

//We need to also make a corresponding class for GeoJSON FeatureCollection.
case class FeatureCollection(features: Array[Feature])
    extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length
}

/*
After creating our case classes, we need a way to help Spray convert RichGeometry, Feature,
and FeatureCollection along with a JsValue.
*/
implicit object FeatureJsonFormat extends
    RootJsonFormat[Feature] {
  def write(f: Feature) = {
    val buf = scala.collection.mutable.ArrayBuffer(
      "type" -> JsString("Feature"),
      "properties" -> JsObject(f.properties),
      "geometry" -> f.geometry.toJson)
    f.id.foreach(v => { buf += "id" -> v})
    JsObject(buf.toMap)
  }

  def read(value: JsValue) = {
    val jso = value.asJsObject
    val id = jso.fields.get("id")
    val properties = jso.fields("properties").asJsObject.fields
    val geometry = jso.fields("geometry").convertTo[RichGeometry]
    Feature(id, properties, geometry)
  }
}
```

Preparing the Data

- Don’t use automatic encoder
  - 2 folds, inefficient, wasted when dropped
- Do custom conversion
- If we want to use Dataset
  - Must stick to small data types

```
//--Preparing the New York City Taxi Trip Data--//
val taxiRaw = spark.read.option("header", "true")
    .csv("/proj/cse398-498/course/AAS_CH8/taxidata")
taxiRaw.show()

case class Trip(
    license: String,
    pickupTime: Long,
    dropoffTime: Long,
    pickupX: Double,
    pickupY: Double,
    dropoffX: Double,
    dropoffY: Double
)
//As of now, time is long due to Unix epoch, and x&y will become a Point

//Create a method to parse information if null
class RichRow(row: org.apache.spark.sql.Row) {
    def getAs[T](field: String): Option[T] = {  //returns an Option[T] to
        if (row.isNullAt(row.fieldIndex(field))) {  //explicitly handle nulls
            None
        } else {
            Some(row.getAs[T](field))
        }
    }
}
```

```scala
//Parse string to get time in miliseconds
def parseTaxiTime(rr: RichRow, timeField: String): Long = {
 val formatter = new SimpleDateFormat(
     "yyyy-MM-dd HH:mm:ss", Locale.ENGLISH)
 val optDt = rr.getAs[String](timeField)
 optDt.map(dt => formatter.parse(dt).getTime).getOrElse(0L)
}

//Convert pickup/dropoff locations from string to Doubles using implicit method
def parseTaxiLoc(rr: RichRow, locField: String): Double = {
    rr.getAs[String](locField).map(_.toDouble).getOrElse(0.0) //return 0 if null
}

//Combining all 3 methods into one:
def parse(row: org.apache.spark.sql.Row): Trip = {
val rr = new RichRow(row)
    Trip(
        license = rr.getAs[String]("hack_license").orNull,
        pickupTime = parseTaxiTime(rr, "pickup_datetime"),
        dropoffTime = parseTaxiTime(rr, "dropoff_datetime"),
        pickupX = parseTaxiLoc(rr, "pickup_longitude"),
        pickupY = parseTaxiLoc(rr, "pickup_latitude"),
        dropoffX = parseTaxiLoc(rr, "dropoff_longitude"),
        dropoffY = parseTaxiLoc(rr, "dropoff_latitude")
    )
}
```

Handling Invalid Records at Scale

- Many failures within pipeline: nonstandard data
  - Game of whack-a-mole; one after another
- Can use try-catch block
- In Scala, we can parse invalid entries
- There is 2 possible outcomes, success parsing or failure.
  - Either[L (success), R (failure, a tuple of entry and exception)]

```
def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {
    new Function[S, Either[T, (S, Exception)]] with Serializable {
        def apply(s: S): Either[T, (S, Exception)] = {
            try {
                Left(f(s))
            } catch {
                case e: Exception => Right((s, e))
            }
        }
    }
}

//Apply safe wrapper to parser to prevent parsing issues
val safeParse = safe(parse)
val taxiParsed = taxiRaw.rdd.map(safeParse) //no direct due to Either not in Dataset API
taxiParsed.map(_.isLeft).   //print number correctly parsed
    countByValue().
    foreach(println)
//(true,14776615)

//Since none failed, convert parsed to Dataset
val taxiGood = taxiParsed.map(_.left.get).toDS
taxiGood.cache()
```

```scala
/*
Just because everything parsed properly doesnt mean there are discrpencies within the
data. One of the top of the head, is if dropoff time is earlier than pickup.
*/
//Create a method to convert miliseconds to hours
val hours = (pickup: Long, dropoff: Long) => {
    TimeUnit.HOURS.convert(dropoff - pickup, TimeUnit.MILLISECONDS)
}

//Wrap the hours in a UDF (UserDefinedFunction) to apply to both time columns
import org.apache.spark.sql.functions.udf
val hoursUDF = udf(hours)
taxiGood.
    groupBy(hoursUDF($"pickupTime", $"dropoffTime").as("h")).
    count().
    sort("h").
    show()
    //returns a histogram of time and count (perfect use of DataSetAPI/SparkSQL)
//Analyze odd instance
taxiGood.
    where(hoursUDF($"pickupTime", $"dropoffTime") < 0).
    collect().
    foreach(println)
//Trip(4669D6DB6D5B6739B9194E999D907924,1359155305000,1359125716000,-73.952911,40.748318,-73.952835,40.748287)


//Analyzing histogram shows that most rides are no longer than 3 hours
spark.udf.register("hours", hours)  //registering our hours function as an SparkSql function
val taxiClean = taxiGood.where(
    "hours(pickupTime, dropoffTime) BETWEEN 0 AND 3"
)
//MAIN IDEA: Use Scala's Option[T] to deal with nulls and clean data using Sql
```

```
+---+--------+
|  h|   count|
+---+--------+
| -8|       1|
|  0|14752326|
|  1|   22934|
|  2|     843|
|  3|     197|
|  4|      86|
|  5|      55|
|  6|      42|
|  7|      33|
|  8|      17|
|  9|       9|
| 10|      11|
| 11|      13|
| 12|       7|
| 13|       5|
| 14|       5|
| 15|       3|
| 16|       5|
| 17|       4|
| 19|       3|
+---+--------+
only showing top 20 rows
```

Geospatial Analysis

```
/*
Another instance we clean from the data are checking to see if trips start and end
long/lat are within NY Broughs.
*/
//Read in our GeoJson file using the source class from scala.io
val geojson = scala.io.Source.
    fromFile("/proj/cse398-498/course/aas/ch08-geotime/src/main/resources/nyc-boroughs.geojson").
    mkString

//Using Esri and Spray to parse geojson to FeatureCollection
import com.cloudera.datascience.geotime._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]

//Lets try to test some random point and see where it may be
import com.esri.core.geometry.Point
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
// Some(Feature(Some(72),Map(boroughCode -> 1, borough -> "Manhattan", @id -> ...

/*
To increase time efficency, we are going to take the boroughs that are largest
and move them to the top, that way, statistically, our most common points
will load faster since they are higher up the hierachy.
*/
val areaSortedFeatures = features.sortBy(f => {
  val borough = f("boroughCode").convertTo[Int] //switch boroughs to #1-5
  (borough, -f.geometry.area2D())//based of 2d area
}) //scala auto sorts ascending order
```

```
scala> val features = geojson.parseJson.convertTo[FeatureCollection]
features: com.cloudera.datascience.geotime.FeatureCollection = FeatureCollection(Feature(Some(0),Map(boroughCode -> 5, borough -> "Staten I
sland", @id -> "http://nyc.pediacities.com/Resource/Borough/Staten_Island"),com.cloudera.datascience.geotime.RichGeometry@d1ade54), Feature
(Some(1),Map(boroughCode -> 5, borough -> "Staten Island", @id -> "http://nyc.pediacities.com/Resource/Borough/Staten_Island"),com.cloudera
.datascience.geotime.RichGeometry@4b83786f), Feature(Some(2),Map(boroughCode -> 5, borough -> "Staten Island", @id -> "http://nyc.pediaciti
es.com/Resource/Borough/Staten_Island"),com.cloudera.datascience.geotime.RichGeometry@61f9d53c), Feature(Some(3),Map(boroughCode -> 5, boro
ugh -> "Staten Island", @id -> "http://nyc.pediacities.com/Resource/Borough/Staten_Island"),com.cloude...
```

```
scala> val borough = features.find(f => f.geometry.contains(p))
borough: Option[com.cloudera.datascience.geotime.Feature] = Some(Feature(Some(72),Map(boroughCode -> 1, borough -> "Manhattan", @id -> "htt
p://nyc.pediacities.com/Resource/Borough/Manhattan"),com.cloudera.datascience.geotime.RichGeometry@69f5e3eb))
```

```
//create a histogram of trips to borough
taxiClean.
    groupBy(boroughUDF($"dropoffX", $"dropoffY")).
    count().
    show()
```

```
+--------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+--------------------+--------+
|              Queens|  672192|
|                  NA|  339037|
|            Brooklyn|  715252|
|       Staten Island|    3338|
|           Manhattan|12979047|
|               Bronx|   67434|
+--------------------+--------+
```

```
/*
Most are typically in Manhattan, which is not a suprised, but what is suprising
is the number of NA counts.
*/
```

```
//Filter out all cases where start and stop are 0.0 using SparkSql
val taxiDone = taxiClean.where(
  "dropoffX != 0 and dropoffY != 0 and pickupX != 0 and pickupY != 0"
).cache()

//Rerun histogram to show changes
taxiDone.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
```

```
+--------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+--------------------+--------+
|              Queens|  670912|
|                  NA|   62778|
|            Brooklyn|  714659|
|       Staten Island|    3333|
|           Manhattan|12971314|
|               Bronx|   67333|
+--------------------+--------+
```

Sessionization in Spark

```
val sessions = taxiDone.
    repartition($"license").    //make sure they have same license
    sortWithinPartitions($"license", $"pickupTime") //then sort by pickupTime
sessions.cache()
//When working with large sets like this, it is useful to cache/export out
```

```
def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {
  val b = bLookup(t1.dropoffX, t1.dropoffY)
  val d = (t2.pickupTime - t1.dropoffTime) / 1000
  (b, d)
}

//Instead of using a loop to apply method to all sequential pairs, use sliding
val boroughDurations: DataFrame =
      sessions.mapPartitions(trips => {
        val iter: Iterator[Seq[Trip]] = trips.sliding(2)
        val viter = iter.
            filter(_.size == 2). //ignore if there is only 2 trips
            filter(p => p(0).license == p(1).license) //ignore if license not same
        viter.map(p => boroughDuration(p(0), p(1)))
      }).toDF("borough", "seconds") //returns as DF
boroughDurations.
    where("seconds > 0").
    groupBy("borough").
    agg(avg("seconds"), stddev("seconds")).
    show()
```

```
scala> boroughDurations.
     |   where("seconds > 0 AND seconds < 60*60*4").
     |   groupBy("borough").
     |   agg(avg("seconds"), stddev("seconds")).
     |   show()
+-------------+-----------------+--------------------+
|      borough|     avg(seconds)|stddev_samp(seconds)|
+-------------+-----------------+--------------------+
|       Queens|2380.6603554494727|  2206.6572799118035|
|           NA|  2006.53571169866|  1997.0891370324784|
|     Brooklyn| 1365.394576250576|  1612.9921698951398|
|Staten Island|         2723.5625|  2395.7745475546385|
|    Manhattan| 631.8473780726746|   1042.919915477234|
|        Bronx|1975.9209786770646|   1704.006452085683|
+-------------+-----------------+--------------------+
```

My Extension - SUPER SIMPLIFIED

```
     | show()
+-------------+------------------+--------------------+
|      borough|      avg(seconds)|stddev_samp(seconds)|
+-------------+------------------+--------------------+
|       Queens| 15145.02921535893|   46184.65570022602|
|           NA| 11145.50690421012|   41062.38476837451|
|     Brooklyn|10924.258102953178|   40079.37390372924|
|Staten Island| 17012.34120171674|  41266.189555996105|
|    Manhattan| 3441.172764592876|   22029.98741240281|
|        Bronx|13846.641869522882|   41205.83813202717|
+-------------+------------------+--------------------+
```

```
scala> boroughDurations.
     | where("seconds > 0").
     | groupBy("borough").
     | agg(avg("seconds"), stddev("seconds")).
     | show()
```

← Before

Removed all instances with distances == 0 && (seconds < 60 && distance < 3)

```
sessions: org.apache.spark.sql.Dataset[Trip] = [license: string, pickupTime: bigint ... 5 more fields]
boroughDuration: (t1: Trip, t2: Trip)(String, Long)
boroughDurations: org.apache.spark.sql.DataFrame = [borough: string, seconds: bigint]
+-------------+------------------+--------------------+
|      borough|      avg(seconds)|stddev_samp(seconds)|
+-------------+------------------+--------------------+
|       Queens| 15987.87468371353|   46748.80682717808|
|           NA|12499.481820445773|   44664.82118171125|
|     Brooklyn|11703.938399555638|   41485.08013004814|
|Staten Island|18761.355064844025|  43003.091870988756|
|    Manhattan| 3847.903044809918|   23382.97123334362|
|        Bronx|14974.669308311672|  42677.088182751584|
+-------------+------------------+--------------------+
```