Big Data Analytics

Chapter 7: Geospatial, Temporal & Streaming Data Analysis

Following: [3] "**Advanced Analytics with Spark**", Chapter 8

Prof. Dr. Martin Theobald
Faculty of Science, Technology & Communication
University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

What is Temporal & Spatial Data?

Spatial data is a form of multidimensional data in which some – typically a few – attributes denote annotations over a continuous (spatial) domain. Most common such annotations are:

- **Temporal**: 1d annotations (*time points* where a series of consecutive measurements from the same source are then called a "**session**" or a "**time-series**")
- **Geographic**: mostly 2d (*latitude*/*longitude* pairs as in the **GIS** and **GeoJSON** standards), perhaps 3d annotations (plus *elevation* as in the **GPS** standard)

The most common mathematical representation for multidimensional data points is the **Euclidian space** with **Euclidian distance** as distance metric:

$$\mathrm{d}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Spheres & Polygons

However, Euclidian distance does not work well for the **actual surface distance** of two points **on a spherical shape** (e.g., the shortest path between north and south pole would go through the center of the earth).

- Thus, for the surface distance of two pairs of longitude/latitude points, the **Haversine formula** is more appropriate:

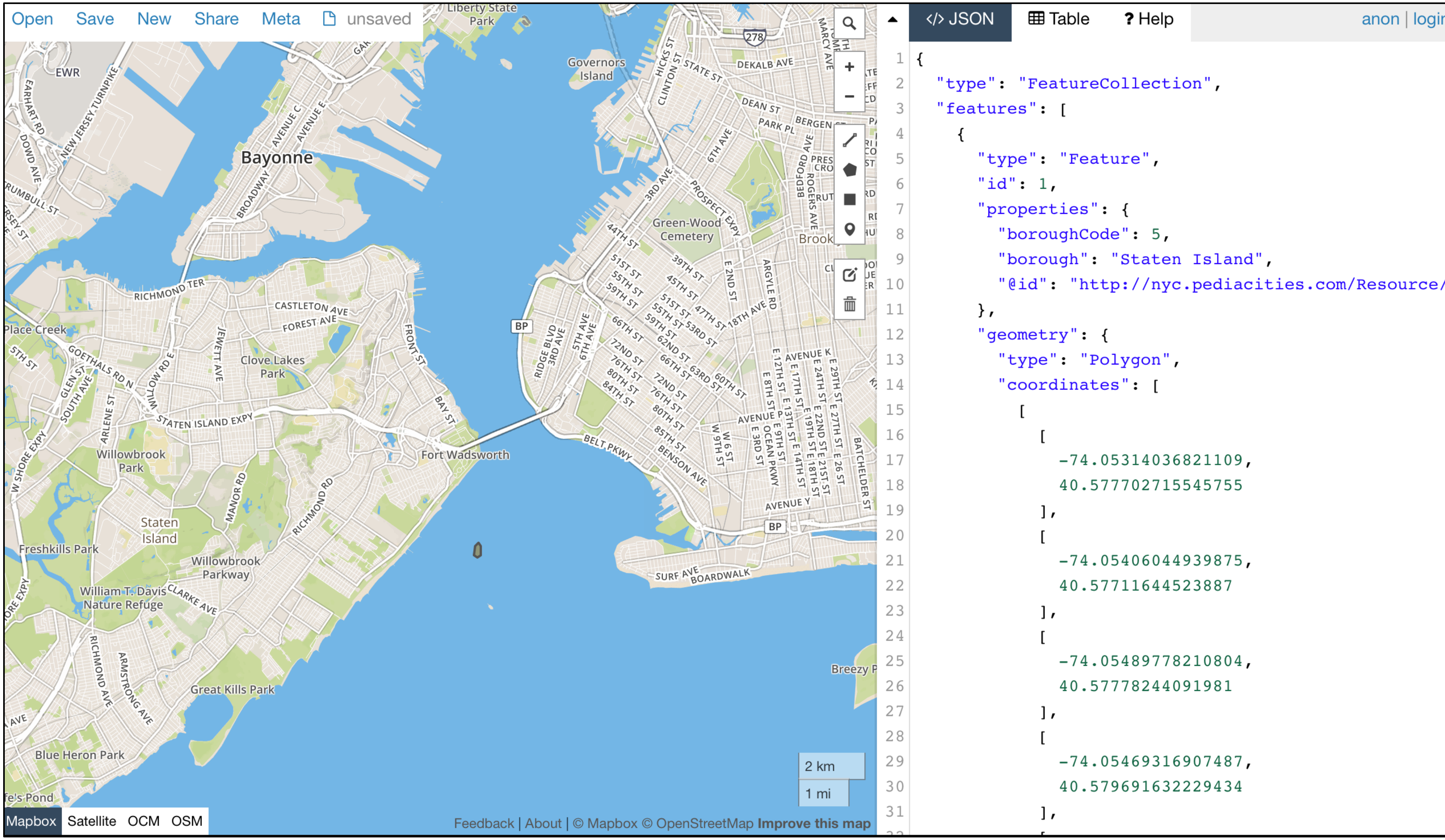
$$\begin{aligned} &\mathrm{d}\left((lat_1, lon_1), (lat_2, lon_2)\right) \\ &\quad = 2R \arcsin\left(\sqrt{\sin^2\left(\tfrac{lat_2 - lat_1}{2}\right) + \cos(lat_1)\cos(lat_2)\sin^2\left(\tfrac{lon_2 - lon_1}{2}\right)}\right) \end{aligned}$$

(→ use $R = 6{,}371$ km for this planet!)

Finally, **polygons** can approximate arbitrary 2d and 3d surfaces (which may then additionally also be projected onto a sphere) by a discrete set of polygon boundaries.

- **Convex polygon**: all internal angles among adjacent edges are $\leq 180°$.
- **Concave polygon**: at least one internal angle among a pair of adjacent edges is > 180°.

GeoJSON Example & Online Demo



http://geojson.io/#map=12/40.6097/-74.0657

The NYC-Taxi-Trips & NYC-Boroughs Data Sets

The **NYC Taxi Trips** data set is a collection of taxi trips and fares in downtown New York City from January 2013 in CSV format:

http://www.andresmh.com/nyctaxitrips/

- The uncompressed file contains 2.5 GB of data for 14.8 million individual taxi rides. Each line contains a driver's license id, start- and end-time, a pick-up GPS location and a drop-off GPS location.

The second data set contains polygons of **NYC city areas** ("**boroughs**") such as Manhattan, Brooklyn, Queens, etc. in GeoJSON format:

https://github.com/haghard/streams-recipes/blob/master/master/nyc-borough-boundaries-polygon.geojson

A basic **geospatial & temporal analytical query pattern** (which we will try to solve in the following slides) may look as follows:

- **How long does it take on average for a taxi driver to find a new customer with respect to the borough and the hour-of-day?**

See here for a very cool visualization of the NYC taxi trips!

Date & Time APIs

- The Java "built-in" `DateTime` and `SimpleDateFormat` libraries are used for **parsing date/time strings**:

```
import java.text.SimpleDateFormat
val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = format.parse("2014-10-12 10:30:44")
val datetime = new DateTime(date)
```

- The `joda` and `nscala` libraries are used to capture **durations**:

```
import com.github.nscala_time.time.Imports._
val dt1 = new DateTime(2014, 9, 4, 9, 0)
val dt2 = new DateTime(2014, 10, 31, 15, 0)
val d = new Duration(dt1, dt2)
d.getMillis
d.getStandardHours
d.getStandardDays
```

Geometric APIs

- The `esri` library provides a special `Geometry` object as a basic data structure for **2d geospatial objects** that are encoded as polygons of longitude/latitude pairs.

```
import com.esri.core.geometry.Geometry
import com.esri.core.geometry.GeometryEngine
import com.esri.core.geometry.SpatialReference
```

Extended Geometry API (I)

- To spare syntax in our further programs, we define a new helper class, called `RichGeometry`, which serves as a wrapper for the `Geometry` class.

- The standard we will use for **spherical distance computations** among a pair of GPS coordinates is WKID 4326.

```
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) }
}
```

Extended Geometry API (II)

- Next, we declare a companion object for `RichGeometry` that provides support for implicitly converting instances of the `Geometry` class into the `RichGeometry` class:

```
object RichGeometry {
  implicit def wrapRichGeo(g: Geometry) = {
    new RichGeometry(g) } }
```

- And make sure to import the implicit function definition into the Scala environment:

```
import RichGeometry._
```

GeoJSON API

- The `spray` package provides a commonly used API for the **GeoJSON standard**:

```
import spray.json.JsValue

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) }
```

- Polygons are encoded in GeoJson as so-called **feature collections**:

```
case class FeatureCollection(features: Array[Feature])
    extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length }
```

Reading and Writing a Collection of GeoJSON Objects

```
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) }
```

Geospatial Data Structure for the Taxi-Trip Data Set

- Based on the `esri` geometric and `nscala` date/time packages, we now define a **custom data structure** for our taxi trips:

```
import com.esri.core.geometry.Point
import com.github.nscala_time.time.Imports._

case class TaxiTrip(
  pickupTime: DateTime,
  dropoffTime: DateTime,
  pickupLoc: Point,
  dropoffLoc: Point)
```

- The parser for the specific date/time format used in the taxi data:

```
val formatter = new SimpleDateFormat(
  "yyyy-MM-dd HH:mm:ss")
```

- And finally a GPS coordinate is parsed from two input strings:

```
def point(longitude: String, latitude: String): Point = {
  new Point(longitude.toDouble, latitude.toDouble) }
```

Parse the Taxi-Trip Data Set

- The `parse` function extracts all the information we need capture taxi drivers and their trip data from each line of the CSV file:

```
def parse(line: String): (String, TaxiTrip) = {
  val fields = line.split(',')
  val license = fields(1)
  val pickupTime = new DateTime(formatter.parse(fields(5)))
  val dropoffTime = new DateTime(formatter.parse(fields(6)))
  val pickupLoc = point(fields(10), fields(11))
  val dropoffLoc = point(fields(12), fields(13))
  val trip = TaxTrip(pickupTime, dropoffTime, pickupLoc,
    dropoffLoc)
  (license, trip) }
```

Interactively Analyze Invalid Records

- `Either[left, right]` provides an interesting mechanism in Scala to **split** the output of a function **into two disjoint sets**. These are initialized from the `Left` and `Right` constructors.

- We can employ this as a wrapper for our actual parse function to debug interactively inspect and possibly debug mistakes in the CSV data.

```
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))
      }
    } } }
```

Wrapper for Safe Parsing

- Now we may finally read the taxi data from the CSV file into an RDD by using the regular line-by-line input format:

```
val taxiRaw = sc.textFile("./path-to-taxi-trip-data")
val taxiHead = taxiRaw.take(10)
taxiHead.foreach(println)
```

- This first RDD is transformed into a second RDD using the safe parsing function:

```
val safeParse = safe(parse)
val taxiParsed = taxiRaw.map(safeParse)
taxiParsed.cache()
```

- And inspect the good vs. the bad (i.e., erroneous) tuples:

```
taxiParsed.map(_.isLeft).countByValue().foreach(println)
```

Handling Bad Records

- We need two steps to **filter the out the bad tuples** from the RDD with all parsed tuples:

```
val taxiBad = taxiParsed.filter(_.isRight).map(_.right.get)
```

```
val taxiBad = taxiParsed.collect({
  case t if t.isRight => t.right.get
})
```

- And inspect the bad tuples (these are actually not too many):

```
taxiBad.collect().foreach(println)
```

Handling Good Records

- The **good tuples are kept and also cached** for further processing:

```
val taxiGood = taxiParsed.collect({
  case t if t.isLeft => t.left.get
})
taxiGood.cache()
```

- The further inspect the good tuples for **unusual trip durations**:

```
import org.joda.time.Duration

def getHours(trip: TaxiTrip): Long = {
  val d = new Duration(
    trip.pickupTime,
    trip.dropoffTime)
    d.getStandardHours }

taxiGood.values.map(getHours).countByValue().
  toList.sorted.foreach(println)
```

Filtering Out Meaningless Records

- Suspicious taxi trips are those that have a **negative trip time** or that take **more than 3 hours** (at least for NYC circumstances). The remaining ones are kept in a third RDD:

```
val taxiClean = taxiGood.filter {
  case (lic, trip) => {
    val hrs = hours(trip)
    0 <= hrs && hrs < 3
  }
}
```

Geospatial Data Analysis: Load the NYC Boroughs Geometry

- GeoJSON objects are first loaded from a file …

```
val geojson = scala.io.Source.
  fromFile("./nyc-borough-boundaries-polygon.geojson").mkString
```

- ... and can then very conveniently be parsed directly via the `spray` API:

```
import com.cloudera.science.geojson._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]
```

- Let's test our new geometry function to look up the borough of a given query point:

```
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
```

Mapping from Borough Codes to Geometry Objects

- Next, we will store the polygons captured by the feature collections and map them to their borough names (such as Queens, Brooklyn, etc.).

```
val areaSortedFeatures = features.sortBy(f => {
  val borough = f("boroughCode").convertTo[Int]
  (borough, -f.geometry.area2D())
})
```

- The polgyons are broadcast and accessed via a new `borough` function:

```
val bFeatures = sc.broadcast(areaSortedFeatures)

def borough(trip: TaxiTrip): Option[String] = {
    val feature: Option[Feature] = bFeatures.value.find(f => {
    f.geometry.contains(trip.dropoffLoc) })
  feature.map(f => {
    f("borough").convertTo[String]
  })}
```

Analyze the Filtered Records

- We can now for the first time **combine the information from both data sets** by analyzing the frequencies of the individual boroughs as the drop-off locations for the tax trips:

```
taxiClean.values.map(borough).countByValue().foreach(println)
```

- Since there still seem to be many empty ones, we'll do more filtering and check for the most frequent drop-off boroughs:

```
taxiClean.values.filter(t => borough(t).isEmpty).
  take(10).foreach(println)
```

More Filtering

- If the GPS recording of a trip in the taxi data set did not work properly, the coordinates seem to have been set to a default value of (0.0/0.0).
- We can filter out these entries as follows:

```
def hasZero(trip: Trip): Boolean = {
  val zero = new Point(0.0, 0.0)
  (zero.equals(trip.pickupLoc) || zero.equals(trip.dropoffLoc))
}
```

- This will result in our final RDD which we will cache for further processing:

```
val taxiDone = taxiClean.filter {
  case (lic, trip) => !hasZero(trip)
}.cache()

taxiDone.values.map(borough).countByValue().foreach(println)
```

"Sessionization" of Taxi Trips

- A **session** of consecutive tax trips is a series of rides offered by the same driver.

- We can obtain all of these sessions in a **single pass** over all taxi trips by:
  1. using the drivers' license ids as **primary sorting condition**;
  2. using the trips starting times as **secondary sorting condition**;
  3. **splitting** sessions that take more than a certain time span.

Sessionization via Secondary Sorting (I)

- Recall that the first attribute in the `taxiDone` RDD is the driver's license. This attribute will also serve as the **primary sorting condition** for our taxi trips.
- A **secondary sorting condition** can be defined by the pickup times of each driver and taxi trip, which are selected by an additional function:

```
def secondaryKey(trip: TaxiTrip) = trip.pickupTime.getMillis
```

- An optional split function can be applied to split groups of drivers/consecutive trips with a duration of more than 4 hours (because then a driver is obliged to take a break):

```
def split(t1: TaxiTrip, t2: TaxiTrip): Boolean = {
  val p1 = t1.pickupTime
  val p2 = t2.pickupTime
  val d = new Duration(p1, p2)
  d.getStandardHours >= 4 }
```

Sessionization via Secondary Sorting (II)

- The final sessions are then obtained by a single `groupByKeyAndSortValues` call over the `taxiDone` RDD.
- The resulting RDD is also cached. Here, `30` denotes the desired number of partitions in this RDD.

```
val sessions = groupByKeyAndSortValues(
  taxiDone, secondaryKey, split, 30)
sessions.cache()
```

Analyzing Sessions by the City Boroughs

- The next function **analyzes all pairs of taxi trips**, in which the first trip ended in a particular borough and the second trip ended in any (other) borough.
- The function returns the drop-off borough of the first trip together with the duration between the drop-off time of the first trip and the pick-up time of the second trip.
- The function will be called for pairs of trips that are done by the same taxi driver/license id:

```
def boroughDuration(t1: TaxiTrip, t2: TaxiTrip) = {
  val b = borough(t1)
  val d = new Duration(t1.dropoffTime, t2.pickupTime)
  (b, d)
}
```

The "Sliding" Operator

- Using the `sliding` operator we may obtain an iterator over the sorted trips we defined to entail our sessions.
- The `filter` function ensures that we only analyze sessions that consist of exactly two trips.
- The result is of type `RDD[(Option[String], Duration)]` which we cache.

```
val boroughDurations: RDD[(Option[String], Duration)] =
  sessions.values.flatMap(trips => {
    val iter: Iterator[Seq[TaxiTrip]] = trips.sliding(2)
    val viter = iter.filter(_.size == 2)
    viter.map(p => boroughDuration(p(0), p(1)))
}).cache()
```

Analyze the Session Durations

- Another inspection of these durations (in hours) reveals that very few remaining **trip pairs seem to be invalid** (e.g., have a negative session duration). We may decide to either keep or ignore these in our final analysis.

```
boroughDurations.values.map(_.getStandardHours).
  countByValue().toList.sorted.foreach(println)
```

Finally Analyze the Idle-Time Durations by City Borough

- Recall our initial geospatial and temporal analytical query pattern we defined in the beginning. Here is the solution:

```
import org.apache.spark.util.StatCounter

boroughDurations.filter {
  case (b, d) => d.getMillis >= 0
}.mapValues(d => {
  val s = new StatCounter()
  s.merge(d.getStandardSeconds)
}).
  reduceByKey((a, b) => a.merge(b)).collect().foreach(println)
```

Summary

- The `nscala` and `Joda` packages provide **extended APIs** for managing temporal data (various date/time formats, durations, etc.).

- The `spray` and `esri` packages provide **very rich APIs** for geospatial data (GeoJson, Points, FeatureCollections, etc.)

- Noise and meaningless annotations in the source data usually require a good understanding of the application setting by a human analyst to first **clean the raw data sets**.

- Both geospatial and temporal annotations may **reveal interesting patterns and dependencies** among individual data objects that would otherwise remain concealed behind a simple CSV or TSV format.

- If **linked with other data sources** (such as twitter streams, RSS feeds, or RDF data), this allows us to perform some very exciting data-analytics tasks…


Rank
GFS
JSON
Purity
Systems
XML
MLlib
Sampling
Variable
Random
GeoJSON
Recommender
Analytics
Ham
Confusion
Clustering
Science
Reduce
Parallelism
Language
MeSH
RDD
Streaming
Multivariate
AudioScrobbler
Twitter
Recall
Prediction
Distribution
Precision
HDFS
Broadcast
Visualization
Decision
Validation
Hadoop
Normal
Triangles
Statistical
Component
Value
Data
Outlier
Classification
Spark
Combiner
Action
Gradient
ALS
CoverType
Trees
Partitioner
GPS
Cosine
GraphX
Java
Eigenvalue
Cross
Semantic
Vector
HITS
Entropy
Linear
Analytics
Efficiency
Map
Tests
Natural
WordCount
Decomposition
Networks
API
Trips
Connected
Scala
Forests
Transformation
Matrix
Cliques
Latent
Processing
Gini
Big
Singular
Medline
Scalability
Weather
Medical
Taxi
Predictive
Risk
Strongly
Detection
Sparse
Financial
Factorization
PageRank
Eigenvector
Anomaly
Probability
Accumulator
Confidence
Dense
DStream
K-Means
Spam
Regression
Graphs
DataFrame
Dataset
Dataflow
Boosted
SQL
Open
NoSQL
GIS