Introduction to Machine Learning With Spark

DataSummit 2016

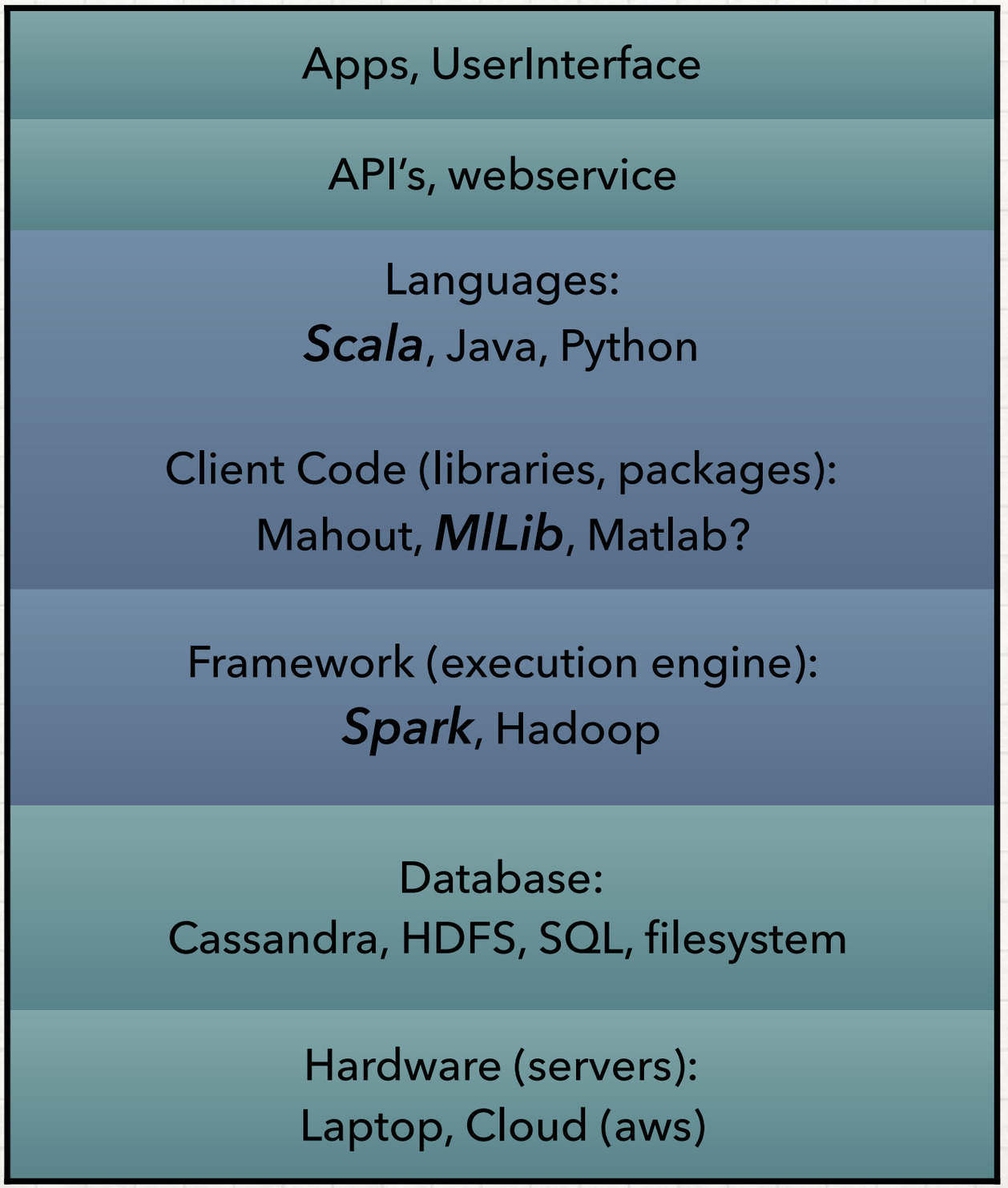
May 9, 2016

Nathan Halko *@nhalko*

Data Scientist, @SpotRight

All code and slides are on github MahoutApp

The Stack


Apps, UserInterface
API's, webservice
Languages:
Scala, Java, Python
Client Code (libraries, packages):
Mahout, *MlLib*, Matlab?
Framework (execution engine):
Spark, Hadoop
Database:
Cassandra, HDFS, SQL, filesystem
Hardware (servers):
Laptop, Cloud (aws)

spotright

Nathan Halko, Data Scientist

n8halko@gmail.com

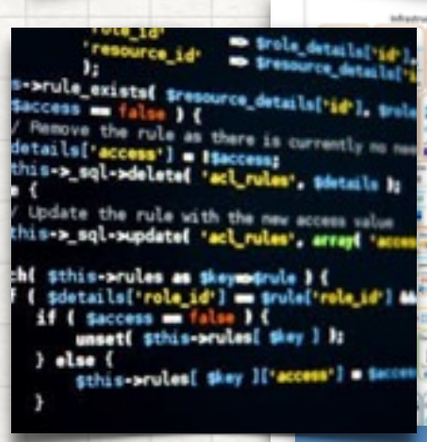
Data Science

Clients

Sales/Marketing

Product

Engineering



Algorithms

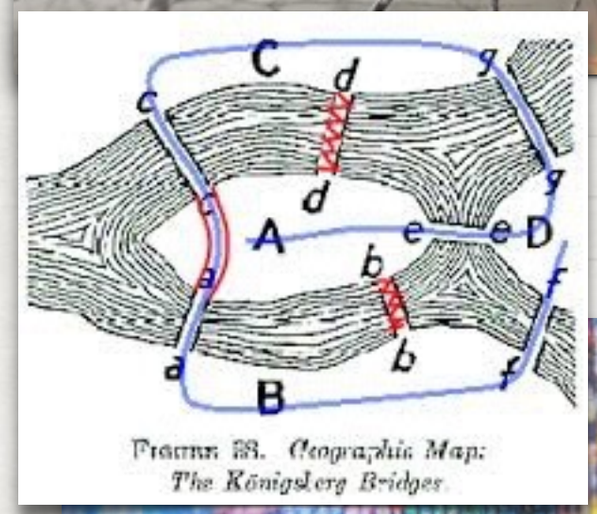
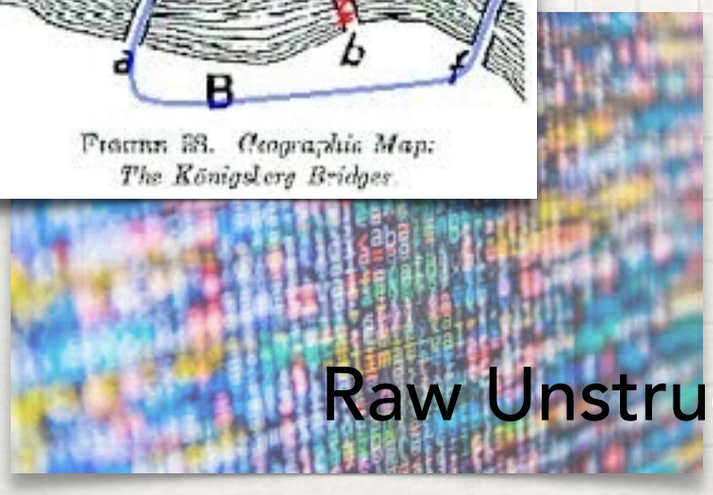


Fig. 28. Geographic Map: The Königsberg Bridges

Infrastructure/IT



Raw Unstructured Data

Machine Learning

Putting some *things* into some *buckets*.

Supervised: you know the names of the buckets.
Unsupervised: you don't.

Tutorials and Credits

1. Decision Tree **classification** for predicting flight delays.
   Carol McDonald, MapR
   tutorial
2. **Clustering** the News with *unsupervised* kmeans
   Dan McClary, Big Data Science Bootcamp
   tutorial
3. **Collaborative Filtering** with MovieLens
   Guy Ernest, AWS
   DataBricks and Grouplens
   tutorial

Apache Zeppelin

Download 0.5.6-incubating Binary package here

Instructions for install

```
 > tar -xzvf ~/Downloads/zeppelin-0.5.6-incubating-bin-all.tgz
  > ln -s zeppelin-0.5.6-incubating-bin-all.tgz zeppelin
  > cd zeppelin
```

Lets give it more memory:

```
 > cp conf/zeppelin-env.sh{.template,}
```

add to zeppelin-env.sh

```
 export ZEPPELIN_PORT=8082    *** optional ***
 export ZEPPELIN_MEM=-Xmx4G


 > bin/zeppelin-daemon.sh start
```

Go to http://localhost:8080

**** Use a new note per exercise and multiple paragraphs within for easy re-compilation ****

Collaborative Filtering with MovieLens

::credit::

Guy Ernest, Solutions Architect with AWS

tutorial

Code/data/material from

DataBricks/spark-training

Download datasets

```
> curl -O https://raw.githubusercontent.com/databricks/spark-training/master/data/movielens/medium/movies.dat
> curl -O https://raw.githubusercontent.com/databricks/spark-training/master/data/movielens/medium/ratings.dat
```

Create a sample set for easy debugging

```
> head -10000 ratings.dat > ratings_sample.dat
```

Skip down to the *"Build the Recommender with SparkML"* section of the tutorial, the basic steps are as follows:

Load data in Spark
Partition: Training 60%, Validation 20%, Test 20%
Define *RMSE* to evaluate model performance
Train the model and select the best fit
Evaluate on the Test data set
Recommend!

```scala
import java.io.File
import scala.io.Source

import org.apache.log4j.Logger
import org.apache.log4j.Level

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
import org.apache.spark.mllib.recommendation.{ALS, Rating, MatrixFactorizationModel}

/**
 * Load the data into Spark
 */
val movieLensHomeDir = "/Users/nhalko/Documents/NYC_2016/datasets/movielens"

val movies = sc.textFile(s"${movieLensHomeDir}/movies.dat").map { line =>
  // MovieID::Title::Genres
  // 1::Toy Story (1995)::Animation|Children's|Comedy
  val Array(id, name, _) = line.split("::", 3)
  id.toInt -> name
}.collect.toMap

val ratings = sc.textFile(s"${movieLensHomeDir}/ratings.dat").map { line =>
  // format: (timestamp % 10, Rating(userId, movieId, rating))
  // UserID::MovieID::Rating::Timestamp
  // 1::1193::5::978300760
  val Array(userId, movieId, rating, timeStamp) = line.split("::", 4)

  (timeStamp.toLong % 10, Rating(userId.toInt, movieId.toInt, rating.toDouble))
}

/**
 * Partition into Training 60%, Validation 20%, Test 20%
 */

val training   = ratings.filter{case (ts, _) => ts < 6}.values.cache()
val validation = ratings.filter{case (ts, _) => ts >= 6 && ts < 8}.values.cache()
val test       = ratings.filter{case (ts, _) => ts >= 8}.values.cache()

println(s"""%table DataSet\tCount
Training\t${training.count()}
Validation\t${validation.count()}
Test\t${test.count()}""")
```

Load data in Spark

Partition: Training 60%, Validation 20%, Test 20%

```scala
/**
 * Define the RMSE for evaluating model performance
 */
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating]): Double = {

    val predictions: RDD[Rating] = model.predict( data.map {r => r.user -> r.product} )

    val predictionsAndRatings = predictions
      .map {p => (p.user, p.product) -> p.rating}
      .join(data.map {r => (r.user, r.product) -> r.rating})
      .values

    math.sqrt(
        predictionsAndRatings
          .map{ case (pred, actual) => math.pow(pred - actual, 2) }
          .mean
        )
}

/**
 * Train the model, find the best one and evaluate performance
 */

case class ModelResult(
    model: Option[MatrixFactorizationModel] = None,
    rmse: Double = Double.MaxValue,
    rank: Int =  0,
    lambda: Double = -1.0,
    numIter: Int = -1
    )


val modelResults = for {
    rank <- List(8, 12)
    lambda <- List(0.1, 10.0)
    numIter <- List(10) // List(10, 20)
} yield {
    val model = ALS.train(training, rank, numIter, lambda)
    val validationRmse = computeRmse(model, validation)

    ModelResult(Some(model), validationRmse, rank, lambda, numIter)
}

// find the model with minimum error
val best = modelResults.minBy(_.rmse)
val bestModel = best.model.get
// display results
println(s"""%table RMSE (validation)\trank\tlambda\tnumIters
${modelResults.map(mr => s"${mr.rmse}\t${mr.rank}\t${mr.lambda}\t${mr.numIter}").mkString("\n")}""")

```

RMSE

Train the model

Pick the best

Evaluate and compare against naive baseline

```scala
// test it on the test data
val testRmse = computeRmse(bestModel, test)
println(s"The best model was trained with rank = ${best.rank} and lambda = ${best.lambda} and numIter = ${best.numIter} and its RMSE on the test set is $testRmse")

// create a naive baseline and compare it with the best model
val meanRating   = training.union(validation).map(_.rating).mean
val baselineRmse = math.sqrt(test.map {r => math.pow(meanRating - r.rating, 2)}.mean)
val improvement  = (baselineRmse - testRmse) / baselineRmse * 100

println("The best model improves the baseline by " + "%1.2f".format(improvement) + "%.")

// save for use in making recommendations (output dir must not exist)
//bestModel.save(sc, s"$movieLensHomeDir/model")

```

```scala
/**
 * Create a method to make the recommendations
 */
val moviesWithGenres = sc.textFile(s"${movieLensHomeDir}/movies.dat").map { line =>
  // MovieID::Title::Genres
  // 1::Toy Story (1995)::Animation|Children's|Comedy
  val Array(id, _, genres) = line.split("::", 3)
  id.toInt -> genres
}.collect.toMap


def recommend(userId: Int, genre: String = "") = {
    val candidates = sc.parallelize(
        moviesWithGenres.filter {case (id, genres) => genres.toLowerCase.contains(genre.toLowerCase)}.keys.toSeq
        )

    val userRatings = ratings
        .filter {case (_, r) => r.user == userId}
        .map {case (_, r) => movies(r.product) -> r.rating}
        .collect()

    val recs =  bestModel
      .predict(candidates.map {movieId => userId -> movieId})
      .collect()
      .sortBy(- _.rating)
      .take(5)
      .map {r =>
          movies(r.product)
      }

    (recs, userRatings)
}
```

Recommend!

```
1
2  /**
3   * Make some recommendations
4   */
5
6  val genres = moviesWithGenres.values.toList.flatMap(_.split("\\|")).distinct.map(g => (g,g))
7  val genre  = z.select("genre", genres)
8  val userId = z.input("userId", "100").asInstanceOf[String]
9
10 val (recs, userRatings) = recommend(userId.toInt, genre.asInstanceOf[String])
11 println(s"\n\n\nRecommended for user: \n${recs.mkString("\n")}")
12 println(s"\n\nBased on:\n${userRatings.mkString("\n")}")
13
14
15
16
```

genre: Sci-Fi

userId: 10

```
Recommended for user:
Star Wars: Episode IV - A New Hope (1977)
Matrix, The (1999)
Star Wars: Episode V - The Empire Strikes Back (1980)
Visitors, The (Les Visiteurs) (1993)
Galaxy Quest (1999)


Based on:
(Midsummer Night's Dream, A (1999),5.0)
(Mission: Impossible (1996),4.0)
(Star Wars: Episode I - The Phantom Menace (1999),3.0)
(Defending Your Life (1991),5.0)
(Breaking Away (1979),3.0)
(Truman Show, The (1998),5.0)
(Prophecy II, The (1998),4.0)
(Alien (1992),3.0)
```

Create a little recommender gui

Decision Tree for predicting flight delays

::credit::

Carol McDonald, SOLUTIONS ARCHITECT, MAPR

tutorial

spark mllib-decision-tree

Get dataset

`https://github.com/caroljmcdonald/sparkmldecisiontree/blob/master/data/rita2014jan.csv.zip`

click *View Raw* to download

```
> unzip rita2014jan.csv.zip
```

Load data into Spark

Transform data into LabeledPoints

Partition data, Training 70% Test 30%

Train model

Evaluate

Imports

```scala
import scala.util.Try

import org.apache.spark._
import org.apache.spark.rdd.RDD
// Import classes for MLLib
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
```

Load data into Spark

```scala
// define the Flight Schema
case class Flight(dofM: String, dofW: String, carrier: String, tailnum: String, flnum: Int, org_id: String,
                  origin: String, dest_id: String, dest: String, crsdeptime: Double, deptime: Double, depdelaymins: Double,
                  crsarrtime: Double, arrtime: Double, arrdelay: Double, crselapsedtime: Double, dist: Int)

// function to parse input into Flight class
def parseFlight(str: String): Flight = {
  val line = str.split(",")
  Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5), line(6), line(7), line(8),
    line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble,
    line(14).toDouble, line(15).toDouble, line(16).toInt)
}

// parse the RDD of csv lines into an RDD of flight classes
val linesCnt = sc.accumulator(0L, "total lines")
val flightsRDD = sc.textFile("/Users/nhalko/Documents/NYC_2016/datasets/flights/rita2014jan.csv")
  .flatMap { line =>
      linesCnt += 1
      Try { parseFlight(line) }.toOption
  }
  .cache()

// use .count to trigger the computation and tick the counters
val validCnt = flightsRDD.count()
val badCnt = linesCnt.value - validCnt
println(s"$badCnt ${"%.4f".format(badCnt / linesCnt.value.toDouble * 100)}% bad lines out of ${linesCnt.value} total.")
```

Map all data to Doubles

```scala
/**
 * Transform non-numeric data into numeric values
 */

val carrierMap: Map[String, Int] = flightsRDD.map(_.carrier).distinct.collect()
  .zipWithIndex
  .toMap

val originMap: Map[String, Int] = flightsRDD.map(_.origin).distinct.collect
  .zipWithIndex
  .toMap

val destMap: Map[String, Int] = flightsRDD.map(_.dest).distinct.collect
  .zipWithIndex
  .toMap

/**
 * Define the features array as a LabeledPoint.  Select only relevant features
 */
val mldata = flightsRDD.map { flight =>
      LabeledPoint(
        if (flight.depdelaymins > 40) 1.0 else 0.0,
        Vectors.dense(
          flight.dofM.toDouble - 1,           // day of month
          flight.dofW.toDouble - 1,           // day of week
          flight.crsdeptime,                  // departure time
          flight.crsarrtime,                  // arrival time
          carrierMap(flight.carrier).toDouble,// carrier
          flight.crselapsedtime,              // elapsed flight time
          originMap(flight.origin).toDouble,  // departure city
          destMap(flight.dest).toDouble       // arrival city
        )
      )
    }.cache()
```

Create ***LabeledPoints*** from relevant features

Notice *continuous* vs *discrete* features

```scala
/**
 * Partition the data into training and test data sets
 */
 // mldata0 is %85 not delayed flights
val mldata0 = mldata.filter(x => x.label == 0).randomSplit(Array(0.85, 0.15))(1)
// mldata1 is %100 delayed flights
val mldata1 = mldata.filter(x => x.label != 0)
// mldata2 is delayed and not delayed
val mldata2 = mldata0 ++ mldata1

//  split mldata2 into training and test data
val splits = mldata2.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
```

Partition data
Training 70%
Test 30%

```scala
/**
 *  Define some parameters and train the model.
 */

// airity of non-continuous features
val categoricalFeaturesInfo = Map(
  0 -> 31,
  1 -> 7,
  4 -> carrierMap.size,
  6 -> originMap.size,
  7 -> destMap.size
)

val numClasses = 2      // 0,1
val impurity   = "gini" // or 'entropy'
val maxDepth   = 3 //9      // depth of the decision tree
val maxBins    = 500 //7000   // discretization of continuous features

// call DecisionTree trainClassifier with the trainingData , which returns the model
val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
impurity, maxDepth, maxBins)


// print out the decision tree
println(model.toDebugString)
```

Train the model

Evaluate

```scala

/**
 * Evaluate model on test instances and compute test error
 */

val obsCnt = sc.accumulator(0.0, "observation count")
val wrongPred = testData.filter { point =>
    obsCnt += 1
    point.label != model.predict(point.features)
}.count()

val ratioWrong = wrongPred / obsCnt.value
println(s"The model is wrong ${"%.4f".format(ratioWrong * 100)}% of the time.")
```

Clustering the News with unsupervised k-means

Dan McClary, Big Data Science Bootcamp

tutorial

Get the data from datahub.io

```
> curl -O ‘https://ckannet-storage.commondatastorage.googleapis.com/2014-05-14T08:57:08.036Z/enwiki-20140116-xml.zip'
> unzip enwiki-20140116-xml.zip
```

Create just file of titles (poor man’s xml parser)

```
> cat enwiki-20140116.xml | egrep "<title>" | sed -e 's/.*<title>//' -e 's/<\/title>//' > wikinews_titles.txt
```

We’ll be skipping the json stuff in the tutorial in favor of this brute force method. The the basic steps are:

Setup with some helper functions.
Load the data and create an ‘entry’ for each word.
Create a Word2Vec model
Create title vectors
Train the kmeans clustering model
See what topics the titles belong to.

```scala
import scala.util.Try
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.feature.Word2Vec
import org.apache.spark.mllib.feature.Word2VecModel
import org.apache.spark.mllib.linalg._
```

Setup with some helper functions.
Load the data and create an ‘entry’
for each word.

```scala
/**
 * Define some helper functions for later use
 */

// vector addition
def sumArray(m: Array[Double], n: Array[Double]): Array[Double] = m.zip(n).map {case (i,j) => i + j}

// scalar multiplication 1/a
def divArray(m: Array[Double], divisor: Double) : Array[Double] = m.map(_ / divisor)

def wordToVector(w: String, m: Word2VecModel): Vector = {
  Try {
    m.transform(w)
  } getOrElse {
      println(s"$w failed to transform")
      Vectors.zeros(100)
  }
}
```

```scala
/**
 * ... Seq(President, of, China, lunches, with, Brazilian, President)
 *     Seq(Palestinians, to, elect, new, president, on, January, 9) ...
 */
val news_titles: RDD[Seq[String]] = sc.textFile("/Users/nhalko/Documents/NYC_2016/datasets/wikinews/wikinews_titles.txt")
    .map(_.split(" ").toSeq)
    .cache()

/**
 * ... Seq(President)
 *     Seq(of)
 *     Seq(China) ...
 */
val news_titles_words: RDD[Seq[String]] = news_titles.flatMap {
    words =>
        words.map(word => Seq(word))
}.cache()

```

Create a Word2Vec model

```
// fit a model that can compute synonyms
val word2vec = new Word2Vec()
val model = word2vec.fit(news_titles_words)
```

Notice we are skipping adding extra words to the titles

```
/**
 * Create a title vector from each word vector by taking
 * an average.  So the title vector is the average vector of all
 * its words.
 */
val title_vectors = news_titles.map { title =>
    val tVec = title
        .map(word => wordToVector(word, model).toArray)
        .reduceLeft(sumArray)
        .map(_ / title.length)

    new DenseVector(tVec).asInstanceOf[Vector]
}.cache()

// Make a tuple of (actualTitle, title_vector)
val title_pairs = news_titles.zip(title_vectors)
```

Create title vectors by averaging the vectors for each word in the title

Train the kmeans clustering model

```
val numClusters = 100
val numIterations = 25

val clusters = KMeans.train(title_vectors, numClusters, numIterations)

// Evaluate clustering by computing Within Set Sum of Squared Errors
val wssse = clusters.computeCost(title_vectors)
```

```scala
// predict the topic for each title
val article_membership: RDD[(Int, String)] = title_pairs.map { case (titleSeq, titleVector) =>
                    // topicId -> title String
    clusters.predict(titleVector) -> titleSeq.mkString(" ")
}.cache()

// number each cluster center
val cluster_centers: RDD[(Int, Vector)] = sc.parallelize(
    clusters.clusterCenters.zipWithIndex.map {case (center, idx) => idx -> center}
    )

// synonyms are an Array of length 5 with (word, distance) tuples
// Map away the 'distance' and concatenate to form the cluster topic
val cluster_topics: RDD[(Int, String)] = cluster_centers.map {case (idx, center) =>
    idx -> model.findSynonyms(center, 5)
         .sortBy {case (word, dist) => dist}  // sort by the distance to put the best fits first
         .map {case (word, dist) => word}
         .mkString(" ")
}
println(s"${cluster_topics.take(3).mkString("\n")}")
```

Assign each title to a topic

```scala
val sample_topics = cluster_topics.collect
def sample_members(topicId: Int) = {
    article_membership
        .filter {case (id, title) => id == topicId}
        .map {case (_, title) => title}
        .take(25)
}

val topic = z.select("topic", sample_topics.map {case (id, title) => id.toString -> title})

println(s"%table Related Articles:\n${sample_members(topic.toString.toInt).mkString("\n")}")
```

topic

actress 76 aged wrestler poet

Make a gui to show some titles for a selected topic

```scala
val topicMap = sample_topics.toMap

def topicMe(title: String) = {
    val tVec = {
        val tSeq = title.split(" ")
        val tArr = tSeq
            .map(word => wordToVector(word, model).toArray)
            .reduceLeft(sumArray)
            .map(_ / tSeq.length)

        new DenseVector(tArr).asInstanceOf[Vector]
    }

    val topicId = clusters.predict(tVec)

    topicMap(topicId)
}

println(s"\n\nBelongs to topic: ${topicMe(z.input("Type some words:", "Here comes the sun").toString)}")
```

Type some words:

spark machine learning is fun

See where some random titles might fall!

Next steps:

- integrate into an sbt project with editor to explore more source code *(done)*
- Code dive into algorithms for optimizations:
  - .cache() vs .persist(DISK) *speed*
  - local vs distributed *memory*

- Do I *need to* break up the algorithms or replace/optimize anything?

Mahout

https://mahout.apache.org/ , https://github.com/apache/mahout, mahout-samsara-book , https://github.com/andrewpalumbo/mahout-samsara-book

Spark:: http://spark.apache.org/ , http://spark.apache.org/docs/latest/quick-start.html
Other:: https://git-scm.com/ , https://maven.apache.org/

Git Mahout and Spark (from download):

```
project> git clone https://github.com/apache/mahout.git; cd mahout
project/mahout> echo “this is your project directory”; cd ..
project> tar -xzvf ~/Downloads/spark-1.5.2-bin-hadoop2.6.tgz
project> ln -s spark-mahout spark-1.5.2-bin-hadoop2.6; cd spark-mahout
project/spark-mahout> bin/spark-shell –master local[4]
```

Try it out!

```
spark> val rdd = sc.parallelize(List.fill(1000000)(scala.util.Random.nextDouble))
spark> rdd.count()

SparkUI -> localhost:4041
```

Setup Mahout (from project/mahout directory):

```
- create a file named setup with:
  export MAHOUT_HOME=/project/mahout
  export MAHOUT_LOCAL=true
  export MAHOUT_OPTS=“-Xmx4g"
  export SPARK_HOME=/project/spark-mahout
> source setup # activate the environment variables
> mvn clean install -DskipTests
> MASTER=local[4] bin/mahout spark-shell
```

Try it out!

```
mahout> val mtxA = Matrices.symmetricUniformView(5000, 5000, 1234)
mahout> mtxA.rowSums
```

Creating a project

https://www.jetbrains.com/idea/ , http://www.scala-sbt.org/

Create the bones:

```
> mkdir MahoutApp; cd MahoutApp
MahoutApp> mkdir -p src/{main,test}/scala/com/nhalko/mahoutapp
```

Create a *build.sbt* file with contents:

```
lazy val root = (project in file(".")).
settings(
  name := "mahoutapp",
  version := "0.1",
  scalaVersion := “2.10.4"
 ))
```

Open the project in IDEA and follow the wizard.

Add dependencies and some code:

Update build.sbt to look like this

Create file src/test/scala/com/nhalko/mahoutapp/RidgeRegression.scala

Run it or play with it in the repl!

```
> sbt “testOnly *RidgeRegression”
> sbt test:console
```

Create and run (from the shell) .mscala scripts

```
mahout> :load my_script.mscala
```

Use your .jar in Mahout shell

```
MahoutApp> sbt package # create a .jar in target/scala-2.10 dir of your project
mahout> :cp path/to/your/MahoutApp.jar // access your project’s classes and methods from mahout shell
```