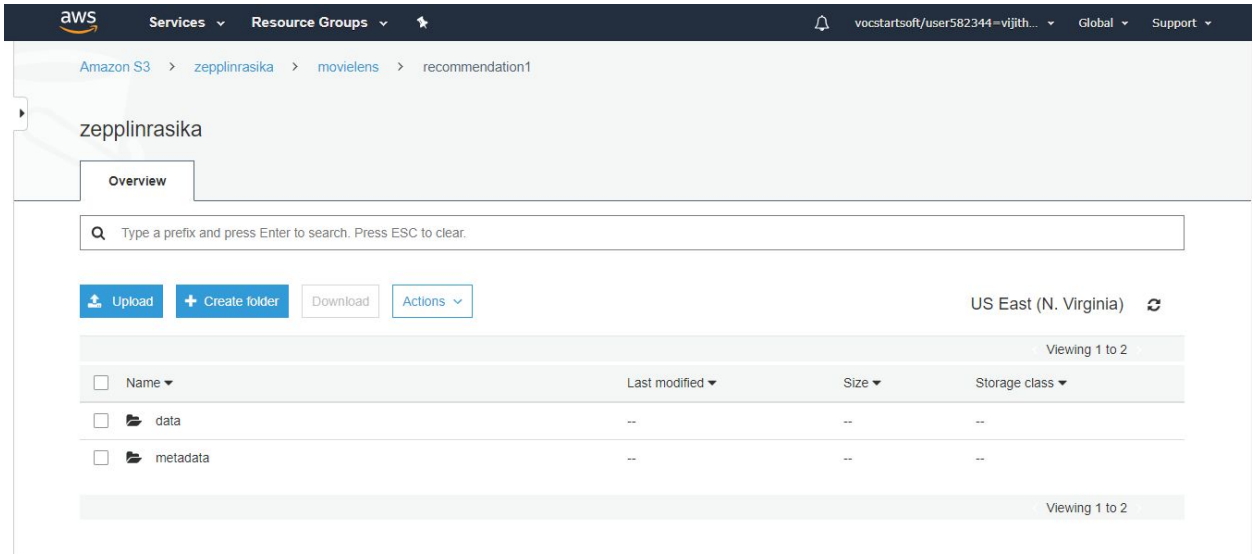
CS5229 - Big Data Analytics

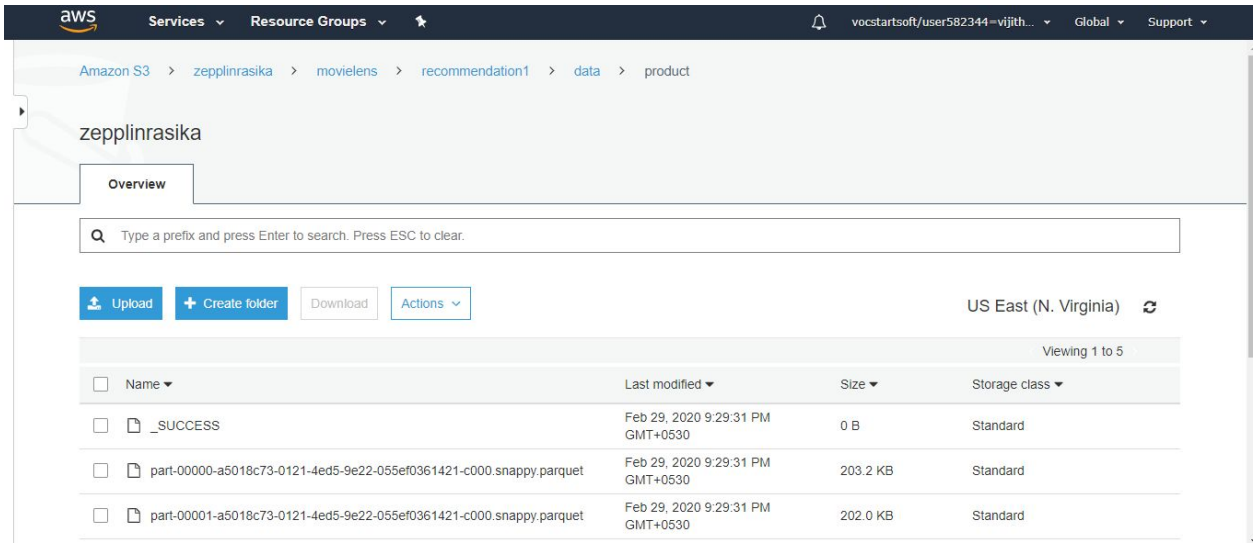
Zepplin Assignment

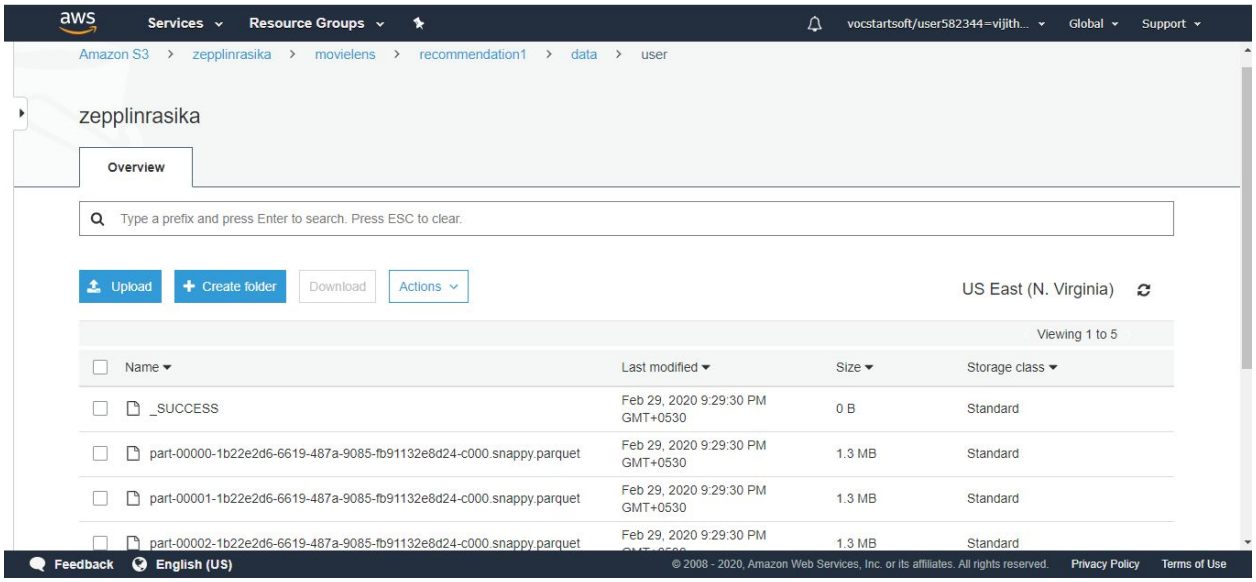
M.R.G.Vijithasena- 209387N

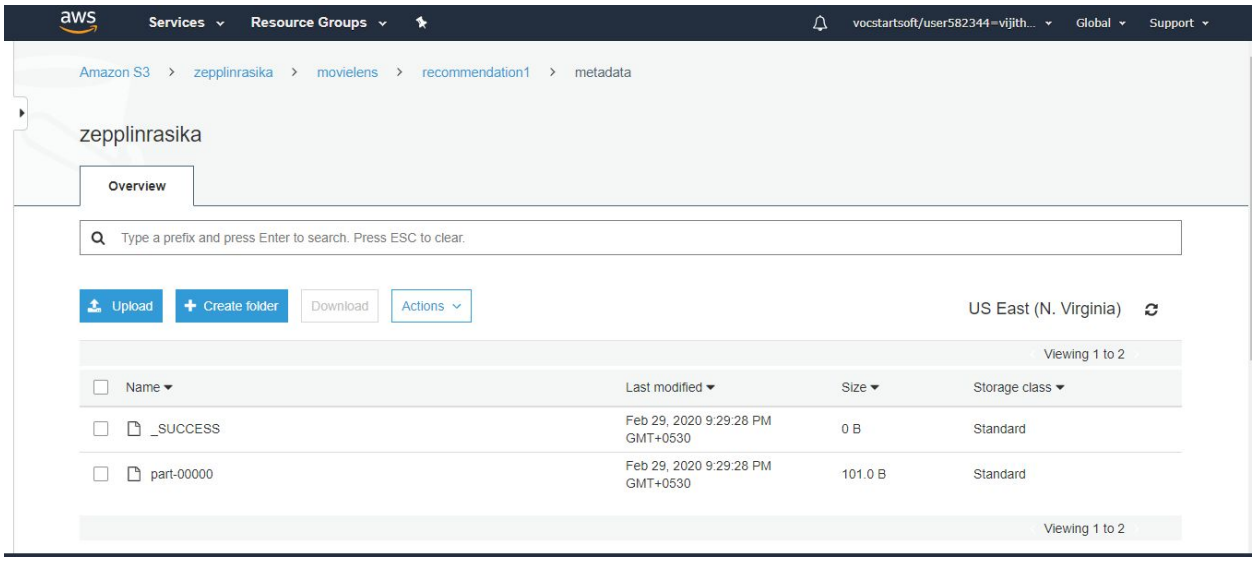
Github Repo
https://github.com/rasikavijithasena/recommendationEngineSparkML

Screenshots of output









Spark Script

```
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}

val movieLensHomeDir = "s3://zepplinrasika/movielens/"

val movies = sc.textFile(movieLensHomeDir + "movies.dat").map { line =>
  val fields = line.split("::")
  // format: (movieId, movieName)
  (fields(0).toInt, fields(1))
}.collect.toMap

val ratings = sc.textFile(movieLensHomeDir + "ratings.dat").map { line =>
  val fields = line.split("::")
  // format: (timestamp % 10, Rating(userId, movieId, rating))
  (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))
}


val numRatings = ratings.count
val numUsers = ratings.map(_._2.user).distinct.count
val numMovies = ratings.map(_._2.product).distinct.count

println("Got " + numRatings + " ratings from "
  + numUsers + " users on " + numMovies + " movies.")


  val training = ratings.filter(x => x._1 < 6)
  .values
  .cache()
val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
  .values
  .cache()
val test = ratings.filter(x => x._1 >= 8).values.cache()

val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

println("Training: " + numTraining + ", validation: " + numValidation + ", test: " + numTest)



/** Compute RMSE (Root Mean Squared Error). */
```

```
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long): Double = {
    val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
    val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
    .join(data.map(x => ((x.user, x.product), x.rating))).values
    math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_ + _) / n)
}


val ranks = List(8, 12)
val lambdas = List(0.1, 10.0)
val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
  val model = ALS.train(training, rank, numIter, lambda)
  val validationRmse = computeRmse(model, validation, numValidation)
  println("RMSE (validation) = " + validationRmse + " for the model trained with rank = "
    + rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
  if (validationRmse < bestValidationRmse) {
    bestModel = Some(model)
    bestValidationRmse = validationRmse
    bestRank = rank
    bestLambda = lambda
    bestNumIter = numIter
  }
}



// evaluate the best model on the test set
val testRmse = computeRmse(bestModel.get, test, numTest)

println("The best model was trained with rank = " + bestRank + " and lambda = " + bestLambda
  + ", and numIter = " + bestNumIter + ", 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(x => (meanRating - x.rating) * (meanRating - x.rating)).mean)
```

```
val improvement = (baselineRmse - testRmse) / baselineRmse * 100
println("The best model improves the baseline by " + "%1.2f".format(improvement) + "%.")


val candidates = sc.parallelize(movies.keys.toSeq)
val recommendations = bestModel.get
  .predict(candidates.map((100, _)))
  .collect()
  .sortBy(- _.rating)
  .take(10)

var i = 1
println("Movies recommended for you:")
recommendations.foreach { r =>
  println("%2d".format(i) + ": " + movies(r.product))
  i += 1
}



val moviesWithGenres = sc.textFile(movieLensHomeDir + "movies.dat").map { line =>
  val fields = line.split("::")
  // format: (movieId, movieName, genre information)
  (fields(0).toInt, fields(2))
}.collect.toMap



val comedyMovies = moviesWithGenres.filter(_._2.matches(".*Comedy.*")).keys
val candidates = sc.parallelize(comedyMovies.toSeq)
val recommendations = bestModel.get
  .predict(candidates.map((100, _)))
  .collect()
  .sortBy(- _.rating)
  .take(5)

var i = 1
println("Comedy Movies recommended for you:")
recommendations.foreach { r =>
  println("%2d".format(i) + ": " + movies(r.product))
  i += 1
}
```

```
// Save and load model
bestModel.get.save(sc, "s3://zepplinrasika/movielens/recommendation1")
val sameModel = MatrixFactorizationModel.load(sc,
"s3://zepplinrasika/movielens/recommendation1")
```