

Machine Learning using Spark MLlib

Spark MLlib allows us to perform Machine Learning and Data Science operations for huge datasets on distributed clusters.

Before we start, we should realize that not all Machine Learning algorithms are easy to implement on distributed systems. This makes ML in Spark challenging. Spark's MLlib and the recent ML library has come far in creating algorithms, but it has a lot of ground to cover. Some of the algorithms covered in MLlib are:

1. Feature Extraction
2. Basic Statistics
3. Linear and Logistic Regression
4. Support Vector Machines
5. Naïve Bayes Classifier
6. Decision Trees
7. Decision Trees

MLlib comes with its specialized data types:

1. Vector – Vector is a way of representing large arrays of values, which may have missing values. For this we have Sparse and Dense vectors.
2. Labeled Point – Supervised ML is all about labelling data points based on some rules. Labeled Point helps us in making that association.
3. Rating – This data type is specialized for data which contains ratings for products given by consumers like in movies or online shopping.

Getting the Data:

We will be working on the movie rating dataset to predict movie recommendations. The data is read from the file and split using '\t' delimiter. We only take the columns that are needed by us (UserID, movieID and Rating) and store it in the final dataframe called 'ratings'.

Creating the ML Model:

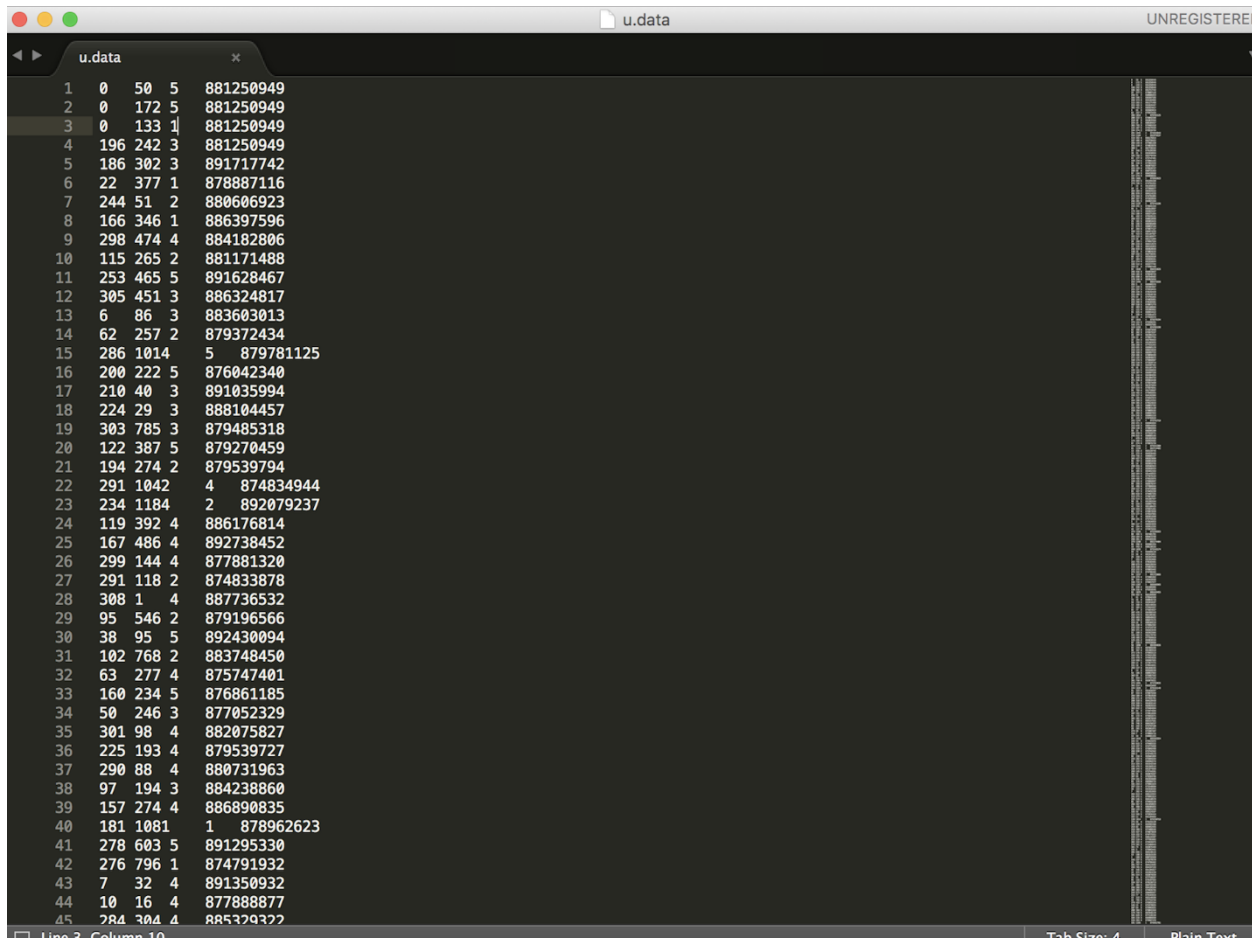
We will be using the Alternating Least Squares (ALS) model to learn from the ratings data. We will train this model using parameters like Rank and Number of Iterations on the training data (ratings dataframe). We store the trained model in the variable 'model'.

We later on use the 'recommendProducts' function to create 10 recommendations for any given userID provided through the command line.

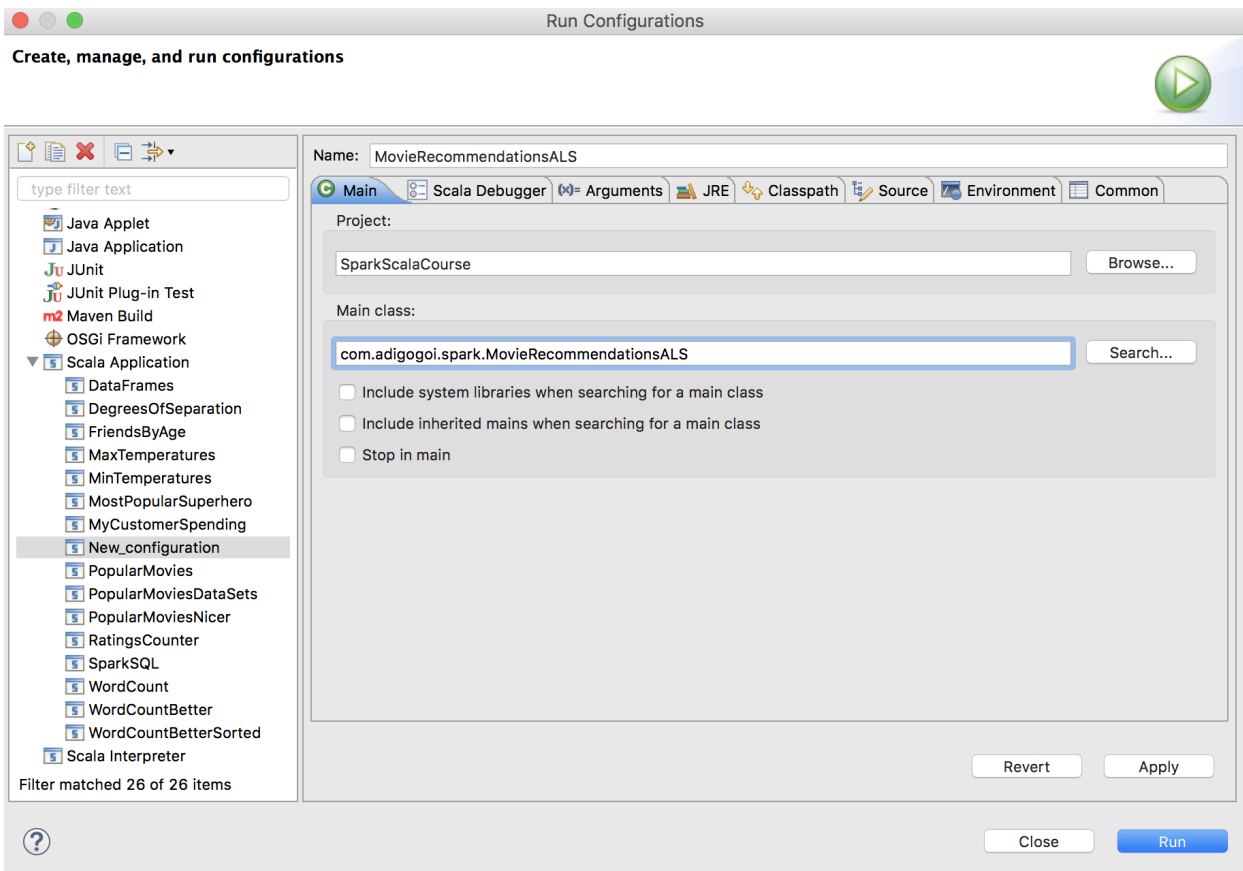
Running the Program:

To test how ALS model of MLlib works, we will have to make a few adjustments to the dataset and then run the code in a specific way.

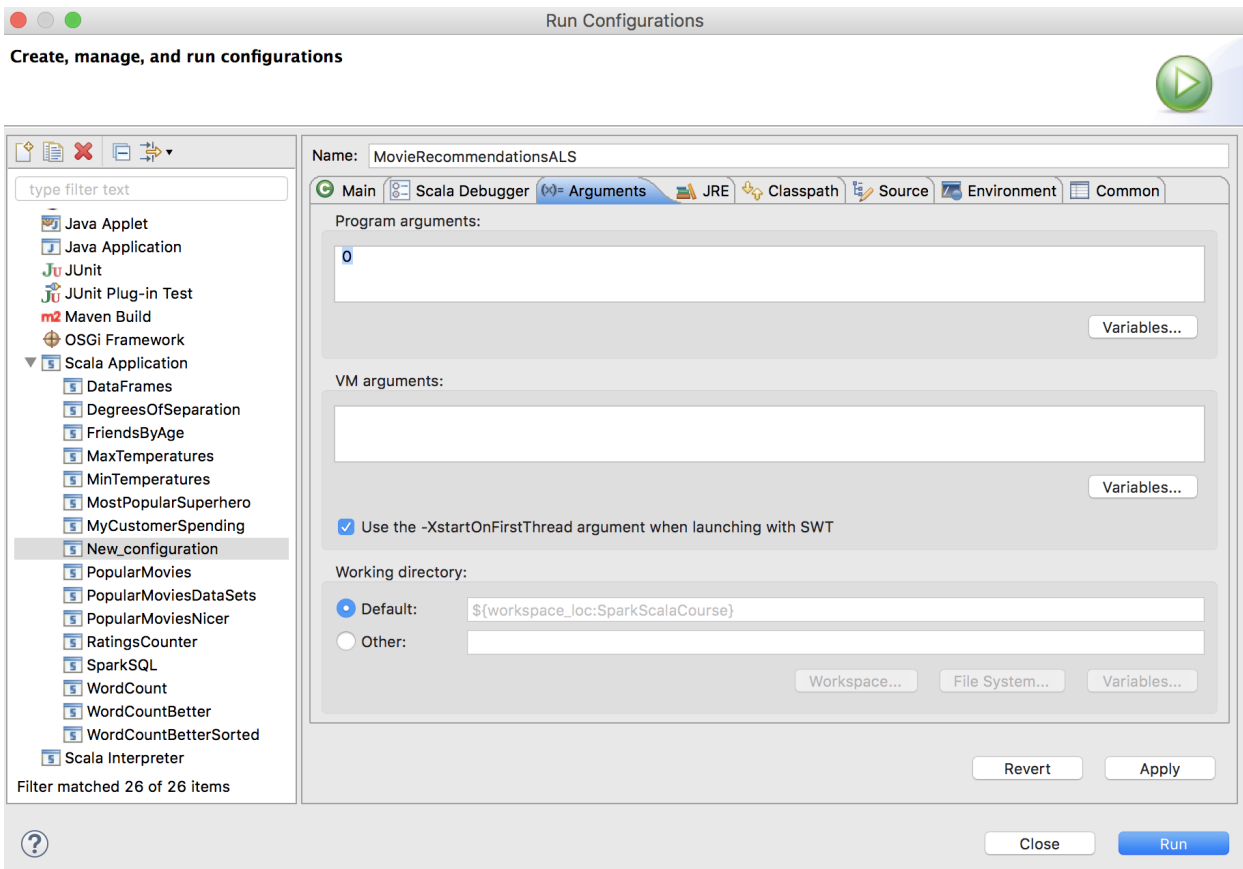
1. We will create a fictional user. This user loves Sci-fi movies and has hence rated Star Wars (Movie ID 50) a 5. This user hates romantic movies and has given Gone With The Wind (Movie ID 133) a 1 rating. This will be stored in our data file as User ID 0.

A screenshot of a text editor window titled 'u.data' with a dark background. The window shows a list of 45 lines of data, each containing five integers separated by spaces. The first column represents a line number (1 to 45), the second column represents a user ID (mostly 0, with some other values like 196, 186, 22, etc.), the third column represents a movie ID (50, 172, 133, etc.), the fourth column represents a rating (mostly 5, with some 1s and 2s), and the fifth column represents a long integer representing a timestamp or item ID (e.g., 881250949, 881250949, etc.). The window has standard macOS window controls (red, yellow, green buttons) at the top left and a status bar at the bottom right indicating 'Tab Size: 4' and 'Plain Text'.

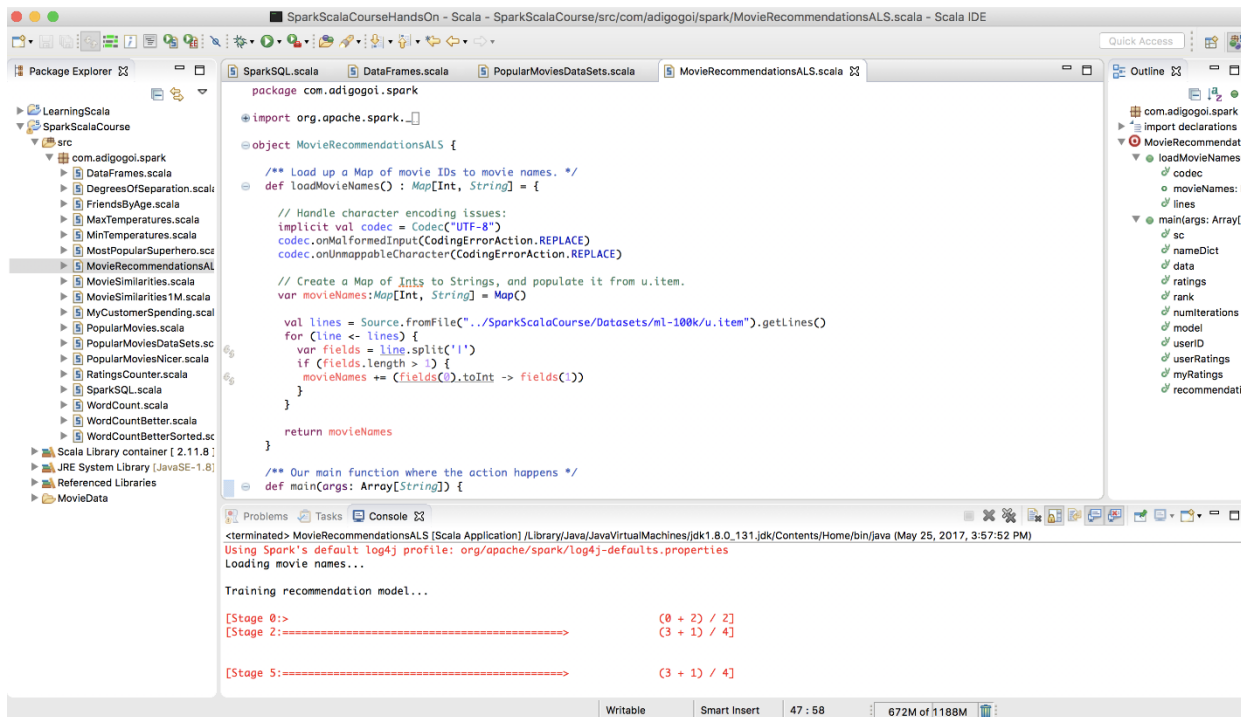
2. After that adjustment, we will run the code as if through a command line. For this we have to go to the 'Run Configuration' option in the Scala IDE for Eclipse that we are using and then give the project name and the Main class of my project.



3. Next we go to the 'Arguments' window and enter '0', which is the User ID we want to predict our recommendations for.



4. The program will train the model for the number of stages specified by us.



```
package com.adigogoi.spark

import org.apache.spark._

object MovieRecommendationsALS {

  /** Load up a Map of movie IDs to movie names. */
  def loadMovieNames(): Map[Int, String] = {

    // Handle character encoding issues:
    implicit val codec = Codec("UTF-8")
    codec.onMalformedInput(CodingErrorAction.REPLACE)
    codec.onUnmappableCharacter(CodingErrorAction.REPLACE)

    // Create a Map of Ints to Strings, and populate it from u.item.
    var movieNames: Map[Int, String] = Map()

    val lines = Source.fromFile("../SparkScalaCourse/Datasets/ml-100k/u.item").getLines()
    for (line <- lines) {
      var fields = line.split('|')
      if (fields.length > 1) {
        movieNames += (fields(0).toInt -> fields(1))
      }
    }

    return movieNames
  }

  /** Our main function where the action happens */
  def main(args: Array[String]) {

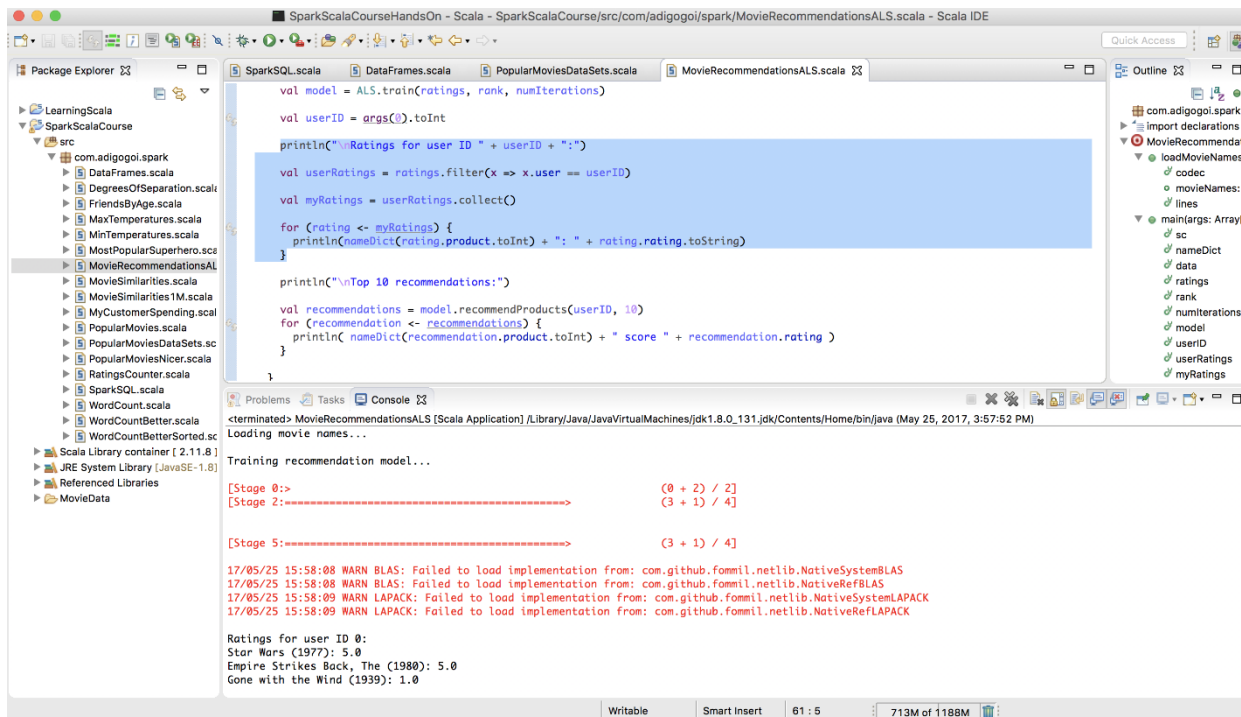
    <terminated> MovieRecommendationsALS [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 25, 2017, 3:57:52 PM)
    Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
    Loading movie names...

    Training recommendation model...

    [Stage 0:>] (0 + 2) / 2]
    [Stage 2:>] (3 + 1) / 4]

    [Stage 5:>] (3 + 1) / 4]
```

5. According to my code, the next thing to be displayed will be the movies rated by the User ID 0 and the rating given for each movie.



```
val model = ALS.train(ratings, rank, numIterations)
val userID = args(0).toInt

println("\nRatings for user ID " + userID + ":")

val userRatings = ratings.filter(x => x.user == userID)
val myRatings = userRatings.collect()

for (rating <- myRatings) {
  println(nameDict(rating.product.toInt) + ": " + rating.rating.toString)
}

println("\nTop 10 recommendations:")

val recommendations = model.recommendProducts(userID, 10)
for (recommendation <- recommendations) {
  println(nameDict(recommendation.product.toInt) + " score " + recommendation.rating)
}

<terminated> MovieRecommendationsALS [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 25, 2017, 3:57:52 PM)
Loading movie names...

Training recommendation model...

[Stage 0:>] (0 + 2) / 2]
[Stage 2:>] (3 + 1) / 4]

[Stage 5:>] (3 + 1) / 4]

17/05/25 15:58:08 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
17/05/25 15:58:08 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
17/05/25 15:58:09 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeSystemLAPACK
17/05/25 15:58:09 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeRefLAPACK

Ratings for user ID 0:
Star Wars (1977): 5.0
Empire Strikes Back, The (1980): 5.0
Gone with the Wind (1939): 1.0
```

6. The ALS model will then make predictions and provide 10 recommendations for User ID 0.

The screenshot shows an IDE window titled "SparkScalaCourseHandsOn - Scala - SparkScalaCourse/src/com/adigogoi/spark/MovieRecommendationsALS.scala - Scala IDE". The code in the editor defines a function to generate movie recommendations for a user. The console output shows the ratings for user ID 0 and the top 10 recommendations.

```
val userID = args(0).toInt
println("\nRatings for user ID " + userID + ":")
val userRatings = ratings.filter(x => x.user == userID)
val myRatings = userRatings.collect()
for (rating <- myRatings) {
  println(nameDict(rating.product.toInt) + ": " + rating.rating.toString)
}

println("\nTop 10 recommendations:")
val recommendations = model.recommendProducts(userID, 10)
for (recommendation <- recommendations) {
  println(nameDict(recommendation.product.toInt) + " score " + recommendation.rating)
}
```

Console Output:

```
<terminated> MovieRecommendationsALS [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 25, 2017, 3:57:52 PM)
17/03/23 13:58:09 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeSyscallLAPACK
17/05/25 15:58:09 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeRefLAPACK

Ratings for user ID 0:
Star Wars (1977): 5.0
Empire Strikes Back, The (1980): 5.0
Gone with the Wind (1939): 1.0

Top 10 recommendations:
Crooklyn (1994) score 13.240224066011272
Endless Summer 2, The (1994) score 13.133363045306
Mina Tannenbaum (1994) score 9.971678134982028
Ruby in Paradise (1993) score 9.784220778876378
Roommates (1995) score 9.626321791084441
Alphaville (1965) score 9.614378680607668
Lost in Space (1998) score 9.191704548861624
Twin Town (1997) score 9.04128185293381
Angel Baby (1995) score 8.938748617629315
Denise Calls Up (1995) score 8.60506315149851
```

Conclusions:

After watching the model work and seeing its inaccurate predictions, I have come to the following conclusions:

1. The ALS algorithm is very sensitive to the parameters provided to it. It takes a little time to find the ideal parameters for a dataset. But due to the black-box nature of the algorithms, it becomes difficult to determine the correct parameters.
2. Complicated is not always better. Sometimes a simple approach to the problem may provide better results.
3. Never blindly trust results when analyzing Big Data. Small problems in algorithms magnify when dealing with such large datasets. Quality of data is the real issue here.