# Advanced Analytics Assignment

*Devon Ankar (dy9186), DSE 6300*

---

## Question 1 (Chapters 1 & 2)

### Chapter 1

Chapter 1 contained no executable code, so here is my summary of the major concepts:

**Key truths of data science, per textbook:**

1. The vast majority of work in producing a successful analysis is in preprocessing data: cleaning, combining, etc. is necessary.
    - This becomes especially complex with larger datasets that cannot be manually examined, and thus require computational methods to determine what cleaning processes are required.
    - Once the dataset is cleaned, more time is spent on feature engineering than algorithm selection.
2. Iteration is fundamental: modeling and analysis both require multiple passes over the same data. Data scientists don't, and shouldn't, focus on getting models right on the first try - they investigate to get a feel for a dataset via exploratory methods and experimentation first.
3. Some practical use needs to come of the analysis. In a business context, the goal is to improve the business's profit by increasing their revenue and/or lowering costs. In an academic context, the final product may be nothing more than a report. In the business world, the final product may be a data application (including an algorithmic trading application like the final project) or a memo sent internally or to clients with actionable recommendations.

**Spark:**

Spark was developed in order to collapse the development-to-production pipeline. The key bottleneck is the productivity of the data scientist. If data scientist has to develop models in R, Python, or a similar language, while the full stack developer must use Java or C++, there can be no direct integration. Spark allows for the full pipeline, from preprocessing, to model evaluation, to production deployment, to be performed on one platform.

### Chapter 2

**Executable code with explanations:**

```
sc
```

Variable referencing `SparkContext`. Returns string form of the object: the object's name plus its address in memory

```
sc.[\t]
```

Returns methods associated with the `sc` object.

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
```

Creates an RDD using the `parallelize` method of `sc` with the specified collection of objects to parallelize; such an RDD is laid out over multiple machines as a collection of partitions (subsets of the data), which define the unit of parallelism in Spark

*To the client machine:*

```
val rddfromhdfs = sc.textFile("hdfs:///path/to/file.txt")
```

Creates an RDD from file(s) in HDFS. Passes the name of the file (or directory containing the files). If Spark is being run on a local machine, then you can pass paths that reside on the local filesystem.

In this line, we are also declaring a new variable called `rddfromhdfs`. All new variables in Scala need to be prefaced with `val` (immutable) or `var` (can be changed), although `val` variables can be redeclared if needed. Also, the type is inferred from context; Scala (much like Python) infers type, so the type does not need to be specified in our variable declaration when creating a new variable.

```
rdd.first
```

An RDD-specific method that allows us to read the first element of the RDD into the client.

```
rdd.collect
```

When we know an RDD contains a small number of records, use this method to return all contents to the client as an array.

```
rdd.take(n)
```

Method that allows us to read `n` number of records into an array on the client. A new array with the first 10 records can be specified with:

```
val head = rdd.take(10)
```

`head.length` can be used to check the length of the array.

*Actions* are required for distributed computing to take place; simply declaring an RDD does not cause any computation.

```
rdd.count()
```

This method returns the number of objects in an RDD, similar to the length attribute of the `.shape` method of a Pandas DataFrame in Python. But because the RDD is not already loaded into memory, its length needs to be counted through a method, rather than being already available as an attribute in the case of Pandas DataFrames (which already reside in memory).

If you want to save the returns to persistent storage instead of just to the local memory, use:

```
rdd.saveAsTextFile("hdfs:///path/to/save/location")
```

```
head.foreach(println)
```

Prints out each value in the array on its own line, for readability. This is a higher order function similar to `apply` and `map` used in Python and R.

The header row can be filtered out as follows. A function is defined:

```
def isHeader(line: String) = line.contains("id_1")
```

The `contains` method returns `True` or `False`, so it's boolean. Here, the author says exactly what I was thinking (as someone whose coding experience is almost exclusively with Python):

> *Like Python, we declare functions in Scala using the keyword def. Unlike Python, we have to specify the types of the arguments to our function [...]*

To test the function against the data in the array:

```
head.filterNot(isHeader).foreach(println)
```

`filterNot` is necessary to filter *out* the header rows! With just `filter`, only the header rows will be returned.

An alternative proposed in the textbook is very similar to Python's `lambda`:

```
head.filter(x => !isHeader(x)).length
```

This is called an anonymous function in Scala. In this case, it's used to negate the `isHeader` inside `filter`.

*Back to the cluster:*

But at some point, we'll need to write our code back to the cluster. So far, we've done all transformations on the local machine with just the head array. We want to apply it to the millions, or billions, of records in the full RDD that resides on the cluster.

```
val noheader = rdd.filter(x => !isHeader(x))
```

Thus, we can interactively develop and debug our data cleaning code against a smaller set of data sampled from the RDD, then ship it back to the entire dataset when we are satisfied.

*Data Frames* - used for datasets that already have a reasonable structure in place, e.g. database tables with regularized schema, or data that has already been cleaned

Similar to a table in an RDB, a Data Frame has records where each record is made up of a set of columns, and columns where each column has a well-defined data type.

Spark Data Frames are NOT similar to Pandas DataFrames in Python!

`spark` refers to the `SparkSession` object

```
val df = spark.read.csv("dataset")
```

Creates a data frame from `spark` using the Reader API

```
val df = spark.read.
    option("header", "true").
    option("nullValue", "?").
    option("inferSchema", "true").
    csv("dataset")
```

Read and parse the dataset with the above code, and use the `.show()` method to show the result

Data Frames can be read from a variety of different formats, such as json:

```
val fromjson = spark.read.json("file.json")
```

and can be stored in persistent memory too:

```
fromjson.write.parquet("file.parquet")
```

Since the Data Frame wraps an RDD, we can use one procedure to aggregate data that will work for any size of data set:

```
df.
    groupBy("is_match").
    count().
    orderBy($"count".desc)
    show()
```

Aggregations can also be completed as follows, in a manner conceptually similar to the SQL group by:

```
df.agg(avg($"field1"), stddev($"field1")).show()
```

And in fact, SQL can directly be used:

```
spark.sql("""
    SELECT is_match, COUNT(*) cnt
    FROM df
    GROUP BY is_match
    ORDER BY cnt DESC
""").show()
```

Returns:

```
is_match    cnt
false       5728201
true        20931
```

A more complex query using an SQL `INNER JOIN` is also possible:

```
spark.sql("""
    SELECT a.field, a.count + b.count total, a.mean - b.mean delta
    FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
    WHERE a.field NOT IN ("id_1", "id_2")
    ORDER BY delta DESC, total DESC
""").show()
```

How to summarize a data frame:

```
val summary = parsed.describe()
summary.show()
summary.select("summary", "field1", "field2").show()
```

Which is similar to the R `summary` or Pandas `describe` methods.

---

# Question 2 (Chapter 3)

### ALS Key Points:

- Matrix factorization model: treats the user and artist play data as if it were a large matrix, where the entry row and column exists if a user has played the given artist.
    - The matrix is sparse because most users have not played songs by most artists; some users may have listened to only a single artist, while even users with very wide-ranging music tastes are unlikely to have listened to over 1,000 artists (of 1.6 million total, so even 10,000 is a small percentage).
    - In this case, we have two matrices: one user-feature matrix (X), and one feature-artist matrix (Y). When they are multiplied, it yields a final user-artist interaction matrix (A) that is used to make recommendations of artists to users.
        - ♠ Y is initialized to a matrix of randomly chosen row vectors. Then linear algebra is used to calculate the best solution for X. This cannot be done exactly, so the goal in practice is to minimize the sum of the squared differences between each entry in the two matrices.
        - ♠ *Alternating* is used in the name because we alternate between X and Y.
        - ♠ Amazingly, even though Y was random, when Y is computed again based on X, and we alternate back and forth like that, X and Y do eventually converge to an optimal solution.
- Latent-factor model: uses a relatively small number of unobserved, underlying reasons to explain observed interactions between large numbers of users and artists.
    - Music tastes are not directly observable or given in the data, but they can be inferred from user behavior using a latent-factor model, even though user behavior may be but a small window into a much wider range of music tastes of a given user.
- Scales well: both in size - can build large models - and in speed - has the ability to create recommendations quickly in real time - the model has to know which song to play next long before a user has finished listening to any given song.
    - The reason ALS is so fast is that it relies on simple, optimized linear algebra, data parallelization, and sparsity of the input data.

### Building the ALS model (executable code):

The first step is to import and understand the available data:

```
val rawUserArtistData =
    spark.read.textFile("hdfs:///user/ds/user_artist_data.txt")

rawUserArtistData.take(5).foreach(println)
```

Each line contains a user ID, artist ID, and play count, separated by spaces. So we split the line on a space character and parse the IDs as integers. We also rename the columns for future convenience.

```
val userArtistDF = rawUserArtistData.map { line =>
    val Array(user, artist, _*) = line.split(' ')
    (user.toInt, artist.toInt)
}.toDF("user", "artist")

userArtistDF.agg(
    min("user"), max("user"), min("artist"), max("artist")).show()
```

We then want to import the artist_data to sort out artists that have songs published under more than one name (including misspelled names) using `.span()` and `.flatMap()`:

```
val artistByID = rawArtistData.flatMap { line =>
    val (id, name) = line.span(_ != '\t')
    if (name.isEmpty) {
        None
    } else {
        try {
            Some((id.toInt, name.trim))
        } catch {
            case _: NumberFormatException => None
        }
    }
}.toDF("id", "name")
```

This gives a data frame with the artist ID and name as columns.

But we want to use the artist aliases as a map of *bad ID:good ID*, not just a dataset of pairs of artist IDs, so now we want to import it as a map:

```
val rawArtistAlias = spark.read.textFile("hdfs:///user/ds/artist_alias.txt")
val artistAlias = rawArtistAlias.flatMap { line =>
    val Array(artist, alias) = line.split('\t')
    if (artist.isEmpty) {
        None
    } else {
        Some((artist.toInt, alias.toInt))
    }
}.collect().toMap

artistAlias.head
```

Only one small transformation is necessary before building the model. We want artist aliases to convert to a canonical ID if a canonical ID exists. Then, we want to parse the lines of the file into separate columns. The following function accomplishes this:

```
import org.apache.spark.sql._
import org.apache.spark.broadcast._

def buildCounts(
        rawUserArtistData: Dataset[String],
        bArtistAlias: Broadcast[Map[Int,Int]]): DataFrame = {
    rawUserArtistData.map { line =>
        val Array(userID, artistID, count) = line.split(' ').map(_.toInt)
        val finalArtistID =
            bArtistAlias.value.getOrElse(artistID, artistID)
        (userID, finalArtistID, count)
    }.toDF("user", "artist", "count")
}

val bArtistAlias = spark.sparkContext.broadcast(artistAlias)

val trainData = buildCounts(rawUserArtistData, bArtistAlias)
trainData.cache()
```

where `bArtistAlias` is a broadcast variable to save memory, and `.cache()` tells Spark that this Data Frame should be stored in memory after being computed.

Now we can build the model:

```
import org.apache.spark.ml.recommendation._
import scala.util.Random

val model = new ALS().
    setSeed(Random.nextLong()).
    setImplicitPrefs(true).
    setRank(10).
    setRegParam(0.01).
    setAlpha(1.0).
    setMaxIter(5).
    setUserCol("user").
    setItemCol("artist").
    setRatingCol("count").
    setPredictionCol("prediction").
    fit(trainData)
```

Due to the size, this takes at least a few minutes to run depending on the system.

Now we ask the important initial question: is the model any good?!

Let's see what a user has listened to:

```
val userID = 2093760

val existingArtistIDs = trainData.
    filter($"user" === userID).
    select("artist").as[Int].collect()

artistByID.filter($"id" isin (existingArtistIDs:_*)).show()
```

Define a function that will return the top recommended artists for a user, by scoring all artists for that user and returning those with the highest predicted score:

```
def makeRecommendations(
    model: ALSModel,
    userID: Int,
    howMany: Int): DataFrame = {

val toRecommend = model.itemFactors.
    select($"id".as("artist")).
    withColumn("user", lit(userID))

model.transform(toRecommend).
    select("artist", "prediction").
    orderBy($"prediction".desc).
    limit(howMany)
}
```

Now, we can compute a user's recommendations, although it takes too long to be practical in real-time use:

```
val topRecommendations = makeRecommendations(model, userID, 5)

val recommendedArtistIDs = topRecommendations.select("artist").as[Int].collect()

artistByID.filter($"id" isin (recommendedArtistIDs:_*)).show()
```

In order to evaluate recommendation quality not just for one user but over the scale of all users, we can use training and test dataset splits, like we did for the final project. This is a widespread accuracy assessment technique in predictive analytics. To do this, we split the dataset into training and test sets, train the model on the training set, and use that model to make predictions for every user in the test set. Then the accuracy of those predictions can be evaluated to calculate the predictive accuracy of a machine learning model.

---

# Question 3

### Collaborative Filtering

I chose item-based collaborative filtering to compare with ALS. Item-based collaborative filtering is applicable here because we need to choose which items (artists, in this case) to recommend to a user based on which other items that user liked previously.

However, user-based collaborative filtering could also work. We could first determine that two users are "similar" in the sense that they listen to a lot of the same artists, then we could recommend new artists to one user based on artists listened to by the other user. We could decide that two users are likely to like the same artist because they previously played songs by other same artists.

So I will look primarily at item-based collaborative filtering, but will consider both approaches.

Please see the attached file `collaborative_filtering.py` for my model in Python.

### Collaborative Filtering vs. ALS

Like ALS, collaborative filtering learns with implicit feedback data, i.e. based on which tracks were played previously, without using information about the users, or about the artists (other than the artist name).

However, collaborative filtering provides, on average, less accurate recommendations than ALS does, at least in this particular implementation. Additionally, it does not run as quickly, though again this is probably due to the limitations of the local system vs. the distributed cloud computing system.

On a real-time deployed system, one would definitely want to use ALS over anything else, as it's the industry standard. However, when learning or just testing out ideas, sub-par, nonstandard systems are acceptable.

---

# References


1. Code to accompany Advanced Analytics with Spark from O'Reilly Media