Big Data Management and Analytics
Session: Mining Big Datasets With Spark MLlib

Lecturer: Pedro González and Sergi Nadal

July 13, 2016

In this session we will use Spark MLlib to write some algorithms aimed to large-scale machine learning. Particularly we will create models for classification, recommendation and clustering.

All in-class practice will be performed in standalone mode (non-distributed), in order to avoid devoting time to setup and configuration issues.

1 Required Tools

- `eclipse` IDE with Java 7 and Maven plugin installed
- `git`
- The following Java libraries (already included in the provided `pom.xml`):
  - `Spark Core`
  - `Spark MLlib`

2 Tasks To Do Before The Session

- Read the slides. Prepare questions (if any) to be answered in the first 30' of class.
- Checkout the `Spark MLlib` programming guide (`https://spark.apache.org/docs/latest/mllib-guide.html`).

3 Part A: Examples & Questions (30min)

During the first 30 minutes of session we will go through the slides, answering all possible questions that have raised.

4 Part B: In-class Practice (2h30min)

As we previously did in Spark's lab session, first we have to setup the environment. Download the enclosed project and follow the same steps to create an `eclipse` project. All datasets are included in a file named *resources.tar.gz*, you should uncompress such folder.

After this is done, you should be able to see the following packages:

- **(default package)**: with the `main` method.
- **exercises (1, 2, 3)**: with the required classes per exercise to complete.

4.1 Exercise 1 (1h): Spam Detection

In this exercise we will make no use of the cluster, hence all coding will be directly performed in eclipse. Furthermore, the datasets are already provided in the `resources` folder of the Java project. They consist of two folders `1_TR` and `1_TE`, respectively the training and test sets. Each folder contains a bunch of `eml` emails that we will use to build our model. On the other hand, the file `1_spam-mail.tr.label` contains pairs `<id, label>`, identifying each training email whether is spam (0) or ham (1).

The Java class `Exercise_1` provides hints and code that you need to complete in order to complete the data curation task. You will find your tasks marked with the `TO DO` tag.

4.2 Data Analysis

The goal of the second part of the exercise is to build a model capable of detecting spam emails, i.e. a spam classifier. The idea of this exercise is to use two MLlib algorithms: `HashingTF` which builds *term frequency* feature vectors from text data, and `LogisticRegressionWithSGD`, which implements the logistic regression procedure using *stochastic gradient descent* (SGD). Similarly as before, follow the hints to build the model.

In order to test your model, after it has been built, append the following code:

```
1 Vector posTestExample = tf.transform(Arrays.asList("O M G GET cheap
    stuff by sending money to ...".split(" ")));
2 Vector negTestExample = tf.transform(Arrays.asList("Hi Dad, I
    started studying Spark the other ...".split(" ")));
3 System.out.println("Prediction for positive test example: "+ model.
    predict(posTestExample));
4 System.out.println("Prediction for negative test example: "+ model.
    predict(negTestExample));
```

Where `tf` is an instance of `HashingTF` and `model` is an instance of `LogisticRegressionModel`. You should get the following output in console:

```
Prediction for positive test example: 1.0
Prediction for negative test example: 0.0
```

4.3 Exercise 2 (1h30min): Recommending Music with the Audioscrobbler Data Set

4.3.1 Data Set and the Alternating Least Squares Recommender Algorithm

This exercise will use a data set published by Audioscrobbler. Audioscrobbler was the first music recommendation system for last.fm, one of the first Internet streaming radio sites, founded in 2002. Audioscrobbler provided an open API for "scrobbling," or recording listeners' plays of artists' songs. last.fm used this information to build a powerful music recommender engine. The system reached millions of users because third-party apps and sites could provide listening data back to the recommender engine.

The Audioscrobbler data set is interesting because it merely records plays: "Bob played a Prince track." A play carries less information than a rating. Just because Bob played the track doesn't mean he actually liked it. Anyone may occasionally play a song by an artist which doesn't care for, or even play an album and walk out of the room. However, listeners rate music far less frequently than they play music. A data set like this is therefore much larger, covers more users and artists, and contains more total information than a rating data set, even if each individual data point carries less information. This type of data is often called implicit feedback data because the user-artist connections are implied as a side effect of other actions, and not given as explicit ratings or thumbs-up.

A snapshot of a data set distributed by last.fm in 2005 can be found in `http://www.bit.ly/1KiJdOR`. The main data set is in the *user_artist_data.txt* file. It contains about 141,000 unique users, and 1.6 million unique artists. About 24.2 million users' plays of artists are recorded, along with their count. For practical reasons, as we are not working in a distributed environment, for the purpose of this session it has been drastically reduced.

The data set also gives the names of each artist by ID in the *artist_data.txt* file. Note that when plays are scrobbled, the client application submits the name of the artist being played. This name could be misspelled or non-standard, and this may only be detected later. For example, "The Smiths," "Smiths, The," and "the smiths" may appear as distinct artist IDs in the data set, even though they are plainly the same. So, the data set also in-

cludes *artist_alias.txt*, which maps artist IDs that are known misspellings or variants to the canonical ID of that artist.

The specification of files is as follows:

```
user_artist_data.txt
    3 columns: userid artistid playcount

artist_data.txt
    2 columns: artistid artist_name

artist_alias.txt
    2 columns: badid, goodid
    known incorrectly spelt artists and the correct artist id.
```

We need to choose a recommender algorithm that is suitable for this implicit feedback data. The data set consists entirely of interactions between users and artists' songs. It contains no information about the users, or about the artists other than their names. We need an algorithm that learns without access to user or artist attributes. These are typically called *collaborative filtering* algorithms.

In this exercise we will employ a member of a broad class of algorithms called latent-factor models. They try to explain observed interactions between large numbers of users and products through a relatively small number of unobserved, underlying reasons. It is analogous to explaining why millions of people buy a particular few of thousands of possible albums by describing users and albums in terms of tastes for perhaps tens of genres, tastes that are not directly observable or given as data.

In the nexts steps, you will be guided on how to build such model, however before being able to do so, the first step to fulfil consists in cleaning and preparing the data to be processed, i.e. pre-processing.

4.3.2 Pre-analysis of Data

You already have the three required files (*artist_alias.txt*, *artist_data.txt* and *user_artist_data.txt*) in the resource folder of your project. The first step in building a model is to understand the data that is available, and parse or transform it into forms that are useful for analysis in Spark. Therefore, our goal here is to study the data we are facing. One small limitation of Spark MLlib's ALS implementation is that it requires numeric IDs for users and items, and further requires them to be nonnegative 32-bit integers. This means that IDs larger than about `Integer.MAX_VALUE`, or 2147483647, can't be used. Does this data set conform to this requirement already?

In order to check it, we will focus in the file *user_artist_data.txt* as it is the (potentially) biggest. Each line of the file contains a user ID, an artist

ID, and a play count, separated by spaces. To compute statistics on the user ID, we split the line by space, and the first (0-indexed) value is parsed as a number. Statistics can be obtained via the method `Statistics.colStats` which will return a `MultivariateStatisticalSummary`, on it you can ask for code such as:

```
1 System.out.println("Statistics for user IDs: [min] "+userIDs_stats.
    min()+", [max] "+userIDs_stats.max()+"; [count] "+userIDs_stats.
    count());
2 System.out.println("Statistics for artist IDs: [min] "+
    artistIDs_stats.min()+", [max] "+artistIDs_stats.max()+"; [count
    ] "+artistIDs_stats.count());
```

4.3.3 Pre-processing

It will be useful later to know the artist names corresponding to the opaque numeric IDs. This information is contained in *artist_data.txt*. This time, it contains the artist ID and name separated by a tab. However, a straightforward parsing of the file into `(Int,String)` tuples will fail as a small number of the lines appear to be corrupted. They don't contain a tab, or they inadvertently include a newline character. These lines cause a `NumberFormatException`, and ideally, they would not map to anything at all. Therefore, you should apply the `flatMap` function (which allows returning 0, 1 or many results) to build a `JavaPairRDD` containing the pair `(ID, Name)`. You can make use of the helper function `Utils.isInteger()` for avoiding exception handling within your transformations.

The *artist_alias.txt* file maps artist IDs that may be misspelled or nonstandard to the ID of the artist's canonical name. It contains two IDs per line, separated by a tab. This file is relatively small, containing about 200,000 entries. It will be useful to collect it as a Map, mapping "bad" artist IDs to "good" ones, instead of just using it as an RDD of pairs of artist IDs. Again, some lines are missing the first artist ID, and you should apply the same strategy as before to build `JavaPairRDD` containing pairs `(bad ID, good ID)`.

4.3.4 Building a First Model

Although the data set is in nearly the right form for use with Spark MLlib's ALS implementation, it requires two small extra transformations. First, the aliases data set should be applied to convert all artist IDs to a canonical ID, if a different canonical ID exists. Second, the data should be converted into `Rating` objects, which is the implementation's abstraction for user-product-value data. Despite the name, `Rating` is suitable for use with implicit data. Note also that MLlib refers to "products" throughout its API, but the "products" here are artists. The underlying model is not at

all specific to recommending products, or for that matter, to recommending things to people.

In order to achieve the first, you will need to access the *artistAlias* `JavaPairRDD`, however it is not possible to access a variable within a transformation. The solution for that is to create a broadcast variable, those variables are useful in situations where many tasks need access to the same (immutable) data structure. You can broadcast a variable using the following code:

```
1 final Broadcast<JavaPairRDD<Integer,Integer>> bArtistAlias = sc.
    broadcast(artistAlias.cache());
```

Now, you should be able to check if such canonical ID exists using the `lookup` function over the broadcasted variable. Finally, you should obtain a variable of type `JavaRDD<Rating>` which will represent your training data set. Note that the process of looking up on the broadcasted variable will create multiple tasks and it is not suitable for a non-distributed environment, therefore for performance reasons it is highly recommended that you reduce the size of the dataset by sampling it, for instance reducing it to 1% of its size:

```
1 JavaRDD<Rating> trainData = rawUserArtistData.sample(true,0.01).map
    (...)
```

Finally, now you can build the model with the following code, which will construct and instance of `MatrixFactorizationModel`:

```
1 ALS.trainImplicit(trainData.rdd(), 10, 5, 0.01, 1.0);
```

The other arguments to `trainImplicit()` are *hyperparameters* whose value can affect the quality of the recommendations that the model makes. The more important first question is, is the model any good? Does it produce good recommendations?

4.3.5 Spot Checking Recommendations

We should first see if the artist recommendations make any intuitive sense, by examining a user, his or her plays, and recommendations for that user. To achieve that we will focus on a band, for instance ID 606 corresponds to "The Rolling Stones", now you can ask the model to provide you a list of users who like this band with the method `recommendUsers`.

With that list of ratings you can now ask the model again for recommended artists for each of those users with the method `recommendProducts`

Finally, if you lookup the name of the artist and write the result in console, you should get something like:

```
User 1000033
```

```
        [The Smiths]
        [The Rolling Stones]
        [Weezer]
        [Bob Dylan]
User 1000239
        [Elvis Costello]
        [Ramones]
        [Cake]
        [John Coltrane]
User 1000022
        [Green Day]
        [Linkin Park]
        [Ramones]
        [At the Drive-In]
User 1000274
        [Pixies]
        [The New Pornographers]
        [Gorillaz]
        [The Clash]
```

This doesn't look like a great set of recommendations, at first glance. However, take into account that has been computed from a 1% sample of an already reduced data set. Additionally, we have not looked at *hyperparameter* selection which makes a great difference in most ML algorithms.

5 Part C: Optional Practice (3h)

5.1 Exercise 3: Anomaly Detection in Network Traffic with K-means Clustering

5.1.1 Introduction

The problem of anomaly detection is, as its name implies, that of finding unusual things. If we already knew what "anomalous" meant for a data set, we could easily detect anomalies in the data with supervised learning. An algorithm would receive inputs labeled "normal" and "anomaly" and learn to distinguish the two. However, the nature of anomalies is that they are unknown unknowns. Put another way, an anomaly that has been observed and understood is no longer an anomaly.

Anomaly detection is often used to find fraud, detect network attacks, or discover problems in servers or other sensor-equipped machinery. In these cases, it's important to be able to find new types of anomalies that have never been seen before—new forms of fraud, new intrusions, new failure modes for servers.

Unsupervised learning techniques are useful in these cases, because they can learn what input data normally looks like, and therefore detect when new data is unlike past data. Such new data is not necessarily attacks or fraud; it is simply unusual, and therefore, worth further investigation.

Clustering is the best-known type of unsupervised learning. Clustering algorithms try to find natural groupings in data. Data points that are like one another, but unlike others, are likely to represent a meaningful grouping, and so clustering algorithms try to put such data into the same cluster. K-means clustering is maybe the most widely used clustering algorithm. It attempts to detect k clusters in a data set, where k is given by the data scientist. k is a hyperparameter of the model, and the right value will depend on the data set.

Here, unsupervised learning techniques like K-means can be used to detect anomalous network connections. K-means can cluster connections based on statistics about each of them. The resulting clusters themselves aren't interesting per se, but they colectively define types of connections that are like past connections. Anything not close to a cluster could be anomalous. Clusters are interesting insofar as they define regions of normal connections; everything else outside is unusual and potentially anomalous.

5.1.2 KDD Cup 1999 Data Set

The KDD Cup was an annual data mining competition organized by a special interest group of the ACM. Each year, a machine learning problem was posed, along with a data set, and researchers were invited to submit a paper detailing their best solution to the problem. For each connection, the data set contains information like the number of bytes sent, login attempts, TCP errors, and so on. Each connection is one line of CSV-formatted data, containing 38 features, like this:

```
0,tcp,http,SF,215,45076,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,
0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,
0.00,0.00,0.00,0.00,0.00,0.00,0.00,normal.
```

This connection, for example, was a TCP connection to an HTTP service—215 bytes were sent and 45,706 bytes were received. The user was logged in, and so on. Many features are counts, like *num_file_creations* in the 17th column.

The complete description of each column of the data set is as follows:

```
duration: numerical.
protocol_type: categorical.
service: categorical.
```

```
flag: categorical.
src_bytes: numerical.
dst_bytes: numerical.
land: categorical.
wrong_fragment: numerical.
urgent: numerical.
hot: numerical.
num_failed_logins: numerical.
logged_in: categorical.
num_compromised: numerical.
root_shell: numerical.
su_attempted: numerical.
num_root: numerical.
num_file_creations: numerical.
num_shells: numerical.
num_access_files: numerical.
num_outbound_cmds: numerical.
is_host_login: categorical.
is_guest_login: categorical.
count: numerical.
srv_count: numerical.
serror_rate: numerical.
srv_serror_rate: numerical.
rerror_rate: numerical.
srv_rerror_rate: numerical.
same_srv_rate: numerical.
diff_srv_rate: numerical.
srv_diff_host_rate: numerical.
dst_host_count: numerical.
dst_host_srv_count: numerical.
dst_host_same_srv_rate: numerical.
dst_host_diff_srv_rate: numerical.
dst_host_same_src_port_rate: numerical.
dst_host_srv_diff_host_rate: numerical.
dst_host_serror_rate: numerical.
dst_host_srv_serror_rate: numerical.
dst_host_rerror_rate: numerical.
dst_host_srv_rerror_rate: numerical.
```

5.1.3 Building the model

Note that the data contains nonnumeric features. For example, the second column may be tcp, udp, or icmp, but K-means clustering requires numeric features. The final label column is also nonnumeric. You can get rid of

those.

You should map the rest of values to a `Vector`. Clustering the data with Spark MLlib is as simple as importing the `KMeans` implementation and calling the `run` method. Once created, you can access the `clusterCenters()` of it, if you print them in console two vectors will be printed, meaning K-means was fitting k = 2 clusters to the data.

For a complex data set that is known to exhibit at least 23 distinct types of connections, this is almost certainly not enough to accurately model the distinct groupings within the data.