Building A Music Recommender in Spark:

Last FM 360k User Recommender Engine Development Overview and Demo

John Bellamy

jlbdatasci.com

Agenda:

Part I: A discussion of the problem and discussion of steps needed to reach the proposed solution.

Part II: Let’s meet the data.

Part III: Data cleaning and formatting.

Part IV: Descriptive analytics. Additional munging.

Part V: Building the model

Part VI: Demonstration of model.

Part VII: Discussion.

Conclusion.

The Problem:

Problem: An executive in sales has come to our team to ask us to build a recommender based on their user data. The dataset is large enough that we will need a distributed framework like Hadoop and Spark to get the work done, and to build an application fast enough to deliver recommendations. Although the data does not contain an explicit rating, the executive hopes we can build a solution that will serve recommendations to our customers.

Solution: We can use ALS (Alternating Least Squares) which is an algorithm that works well for both implicit and explicit ratings, since we do not have an explicit rating. But first, we will load, clean and explore data. Once the data is in the right state, we will test the performance of various models through parameter optimization, and cross-validation.

Once we have selected the best model, we will test the performance and demonstrate the selected model.

Meet the data:

Below is the data in raw form. The fields in order are: md5-hash of User-ID; musicbrainz-artist-id; Artist Name; and Number of Plays.

```
In [1]: text_file="hdfs://hadoop-master:9000/users/spark/data/plays.tsv"

In [2]: plays = sc.textFile(text_file)

In [3]: plays.take(5)

Out[3]: ['00000c289a1829a808ac09c00daf10bc3c4e223b\t3bd73256-3905-4f3a-97e2-8b341527f805\tbetty blowtorch\t2137',
         '00000c289a1829a808ac09c00daf10bc3c4e223b\tf2fb0ff0-5679-42ec-a55c-15109ce6e320\tdie Ärzte\t1099',
         '00000c289a1829a808ac09c00daf10bc3c4e223b\tb3ae82c2-e60b-4551-a76d-6620f1b456aa\tmelissa etheridge\t897',
         '00000c289a1829a808ac09c00daf10bc3c4e223b\t3d6bbeb7-f90e-4d10-b440-e153c0d10b53\telvenking\t717',
         '00000c289a1829a808ac09c00daf10bc3c4e223b\tbbd2ffd7-17f4-4506-8572-c1ea58c3f9a8\tjuliette & the licks\t706']

In [4]: # user-mboxsha1 \t musicbrainz-artist-id \t artist-name \t plays
```

Meet the data:

We need to split the data into four fields.

```
#  user-mboxsha1 \t musicbrainz-artist-id \t artist-name \t plays
# SPlit on /t
ratings_data = plays.map(lambda line: line.split("\t"))
```

```
ratings_data.take(5)
```

```
[['00000c289a1829a808ac09c00daf10bc3c4e223b',
  '3bd73256-3905-4f3a-97e2-8b341527f805',
  'betty blowtorch',
  '2137'],
 ['00000c289a1829a808ac09c00daf10bc3c4e223b',
  'f2fb0ff0-5679-42ec-a55c-15109ce6e320',
  'die Ärzte',
  '1099'],
```

Now we can take the three fields we need.

```
#we only want userid, artist_id  and number of plays; these are found in the following columns
ratings_data_pertinent = ratings_data.map(lambda x: (x[0], x[2],(x[3])))
```

Data Cleaning & Formatting:

In the last slide we showed how we got the data we need. However, we need to perform some formatting on the data to get it in numeric form. The ALS algorithm requires numeric data. Thus, we must create a numeric id for user and for artist.

For the users, we can just replace the letters with numbers. However, we are left with huge numbers, so we need to cut the numbers down in size without using uniqueness.

For the artists, we can collect the unique artists in a dictionary and assign a number to each artist. Then, we can create a function to connect the id to the data joining on the artist name.

After these processes are done, we need to assign a unique index to each set.

For a complete run-through, with annotation, see the LastFM-Munging file.

Data Cleaning & Formatting:

Below are the letters we need to replace in the user hash.

```
alpha_dict = {"a":"1", "b":"2","c":"3","d":"4", "e":"5","f":"6","g":"7",
              "h":"8","i":"9","j":"10","k":"11","l":"12","m":"13","n":"14","o":"15",
                  "p":"16","q":"17","r":"18", "s":"19","t":"20","u":"21","v":"22","w":"23","x":"24",
                  "y":"25","z":"26"," ":"0", "ä":"27","ö":"28"}
```

```
def md5_replace(word,_dict):
    i = list(_dict.keys())
    j = list(_dict.values())
    k = len(j)-1
    _word = numerizer(word)
    while k >= 0:
        _word1 = str.replace(_word, i[k], j[k])
        _word =_word1
        k-= 1
    return _word1
```

```
# Get rid of leading zeroes which will be problematic and apply alpha character function
numeric_uid = ratings_data_pertinent.map(lambda x: md5_replace(x[0], alpha_dict)).map(lambda line: line.lstrip("0"))
```

Data Cleaning & Formatting:

We are still not quite done with getting our user id. We can see we have a massive number for user id.

```
numeric_uid.take(5)
```

```
['328911829180813093004161023334522 32',
 '328911829180813093004161023334522 32',
 '328911829180813093004161023334522 32',
```

```
def shrinker(num):
    places = len(num)-9

    if places > 0:
        zeroes = places

        cutter = "1"
        for x in range(zeroes):
            cutter+=str(0)
        return str(round(int(num)/float(cutter)))

    else:
        return num
```

```
numeric_uid = numeric_uid.map(lambda x : shrinker(x))
```

```
numeric_uid.take(10)
```

```
['328911829',
 '328911829',
```

Data Cleaning & Formatting:

Getting artist id is a bit simpler than user-id.

```
#To get all of the artists we will use groupby. Then we can assign a number to each of the unique artists
artist_name = ratings_data_pertinent.map(lambda x: (x[1])).distinct()
```

```
artist_name.top(5)
```

```
['', '', '', '', '']
```

```
#Make sure we still have all the artists
artist_name.count()
```

```
292589
```

```
#Now we can add a unique id for each artist name
artist_index=sc.parallelize(range(1,292590))
```

```
artist_index = artist_index.zipWithIndex()
artist_name = artist_name.zipWithIndex()
```

```
artist_index = artist_index.map(lambda x: (x[1],x[0]))
artist_name = artist_name.map(lambda x: (x[1],x[0]))
```

```
artist_name.take(5)
```

```
[(0, 'camo & krooked'),
 (1, 'ney-the sufi cry out'),
 (2, 'camela leierth'),
 (3,
  'chamy. ishi, rei kondoh, taiki endo, yasutaka hatade, satoshi miyashita'),
 (4, 'january star')]
```

Data Cleaning & Formatting:

After a little more work and getting an index for both artist, and user, we can join the data into the form we need:

```
#Heres our new dataset! It is in the form of index|user_id|artist_name|artist_id|number_of_listens
_RDD.take(5)
```

```
[(0, '328911829', 0, 'camo & krooked', '2137'),
 (13107200, '266493556', 219632, 'grupo montez de durango', '517'),
 (8912900, '821342251', 154398, 'bob marley', '424'),
 (4718600, '444164354', 82546, "the cat's miaow", '81'),
 (524300, '717350741', 9055, 'ozzy osbourne', '74')]
```

Descriptive Analytics:

In the next few slides I go over descriptive analytics. This step is critical, as we will see. In the first notebook, we performed some major transformations on the raw data, and ended with a dump back onto Hadoop of data in the form we need. In this, the Last FM-Descriptive-Analysis notebook, we start by loading that data and looking at summary statistics.

Descriptive Analytics:

Below is a table which covers the summary statistics.

Statistic	Value
Total number of listens (implicit ratings)	17,538,292
Number of Users	358974
Average number of listens	215
Largest number of listens	419157
Smallest number of listens	0
Highest-rated artist (number of listens)	77348
Average-rated artist (number of listens)	61
Lowest-rated artist (number of listens)	1

Descriptive Analytics:

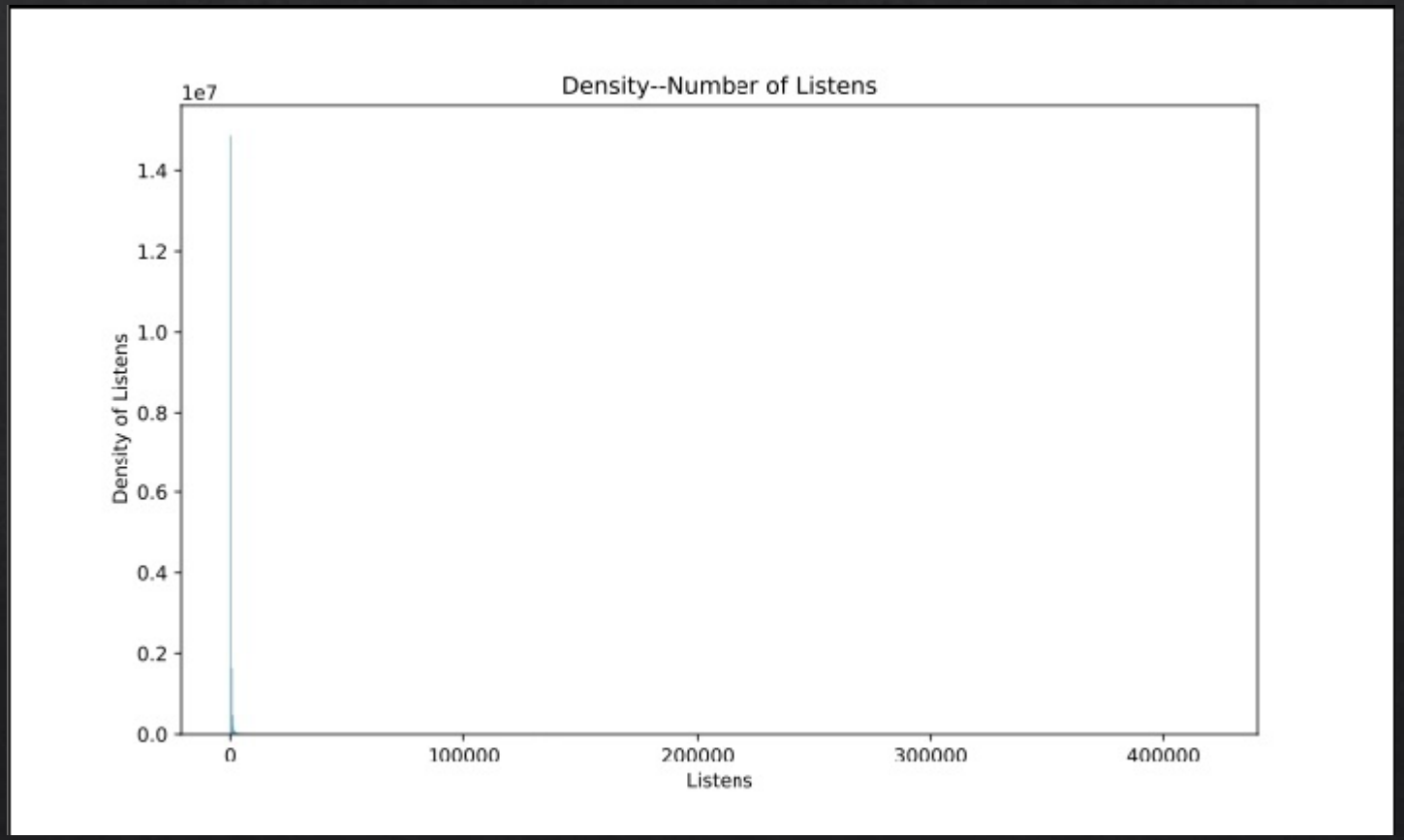
After going over the statistics something jumps out. The maximum number of listens was over 419k. Can a person really listen to a song that many times? A person would have to listen to a song 1,148 times a day for an entire year to reach that number. We obviously have an outlier here and need to discard this data point.

A decision has to be made about what a realistic number of listens is, at least for the purposes of the model. We have to be careful not to skew the model, but it is clear we should not see 419k listens.

Looking at the density of listens helps us see our problem.

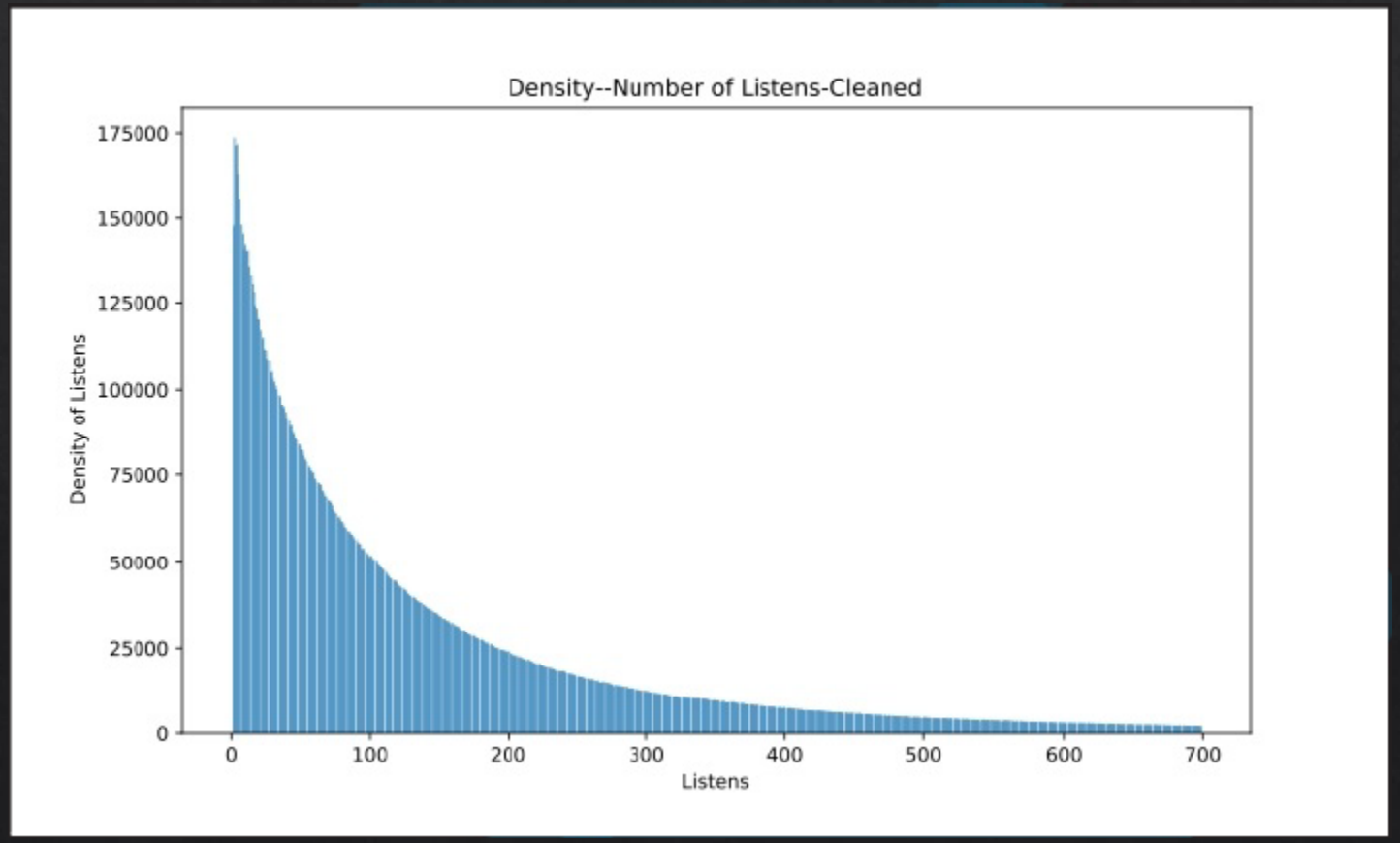
Descriptive Analytics:

Here, we can see two things: we can see that the majority of listens are on the low-end, but with the extreme values it is hard to understand where the low end is.



Descriptive Analytics:

Sometimes there isn't a clear way to proceed. This is the "art" part of data science. After a few iterations, 700 was chosen as the maximum number of listens, which is the implicit rating, as discussed in model section. Looking at the bar chart, this number looks reasonable.



Descriptive Analytics:

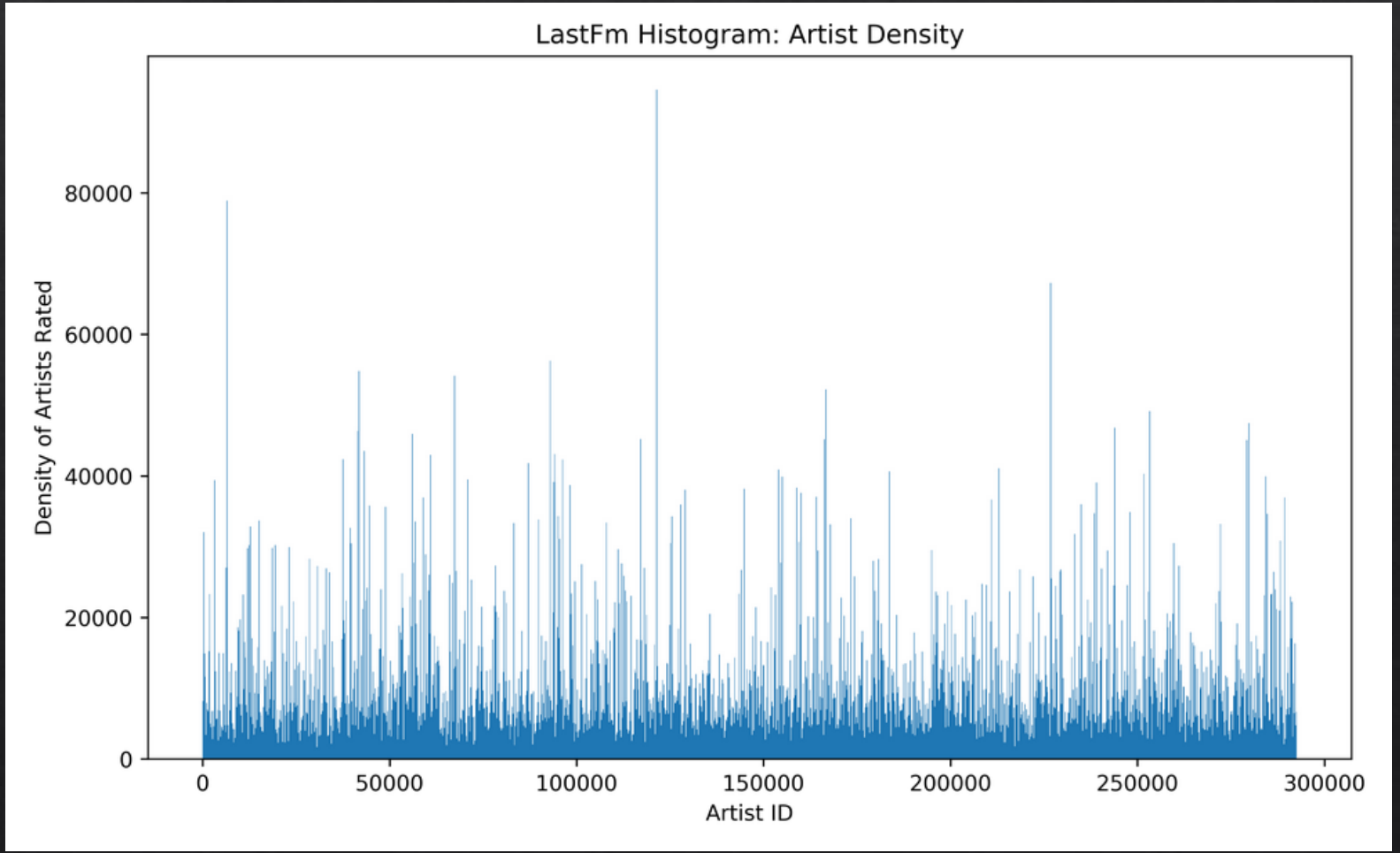
Another Problem: Some artists have as many as 77.3k listens. Others

have only 1. We don't want to eliminate an artist, so the artists that had been listened to (rated) more than 180 times were separated into another dataset through resampling; 180 was chosen, by trial-and-error and is slightly less than the mean. A random sample selecting 180 of each artist was then taken and those artists added back to the dataset. This balances the dataset. Because this process covers a lot of code, please refer to notebook.

The next two slides show the density of artists before and after this process.

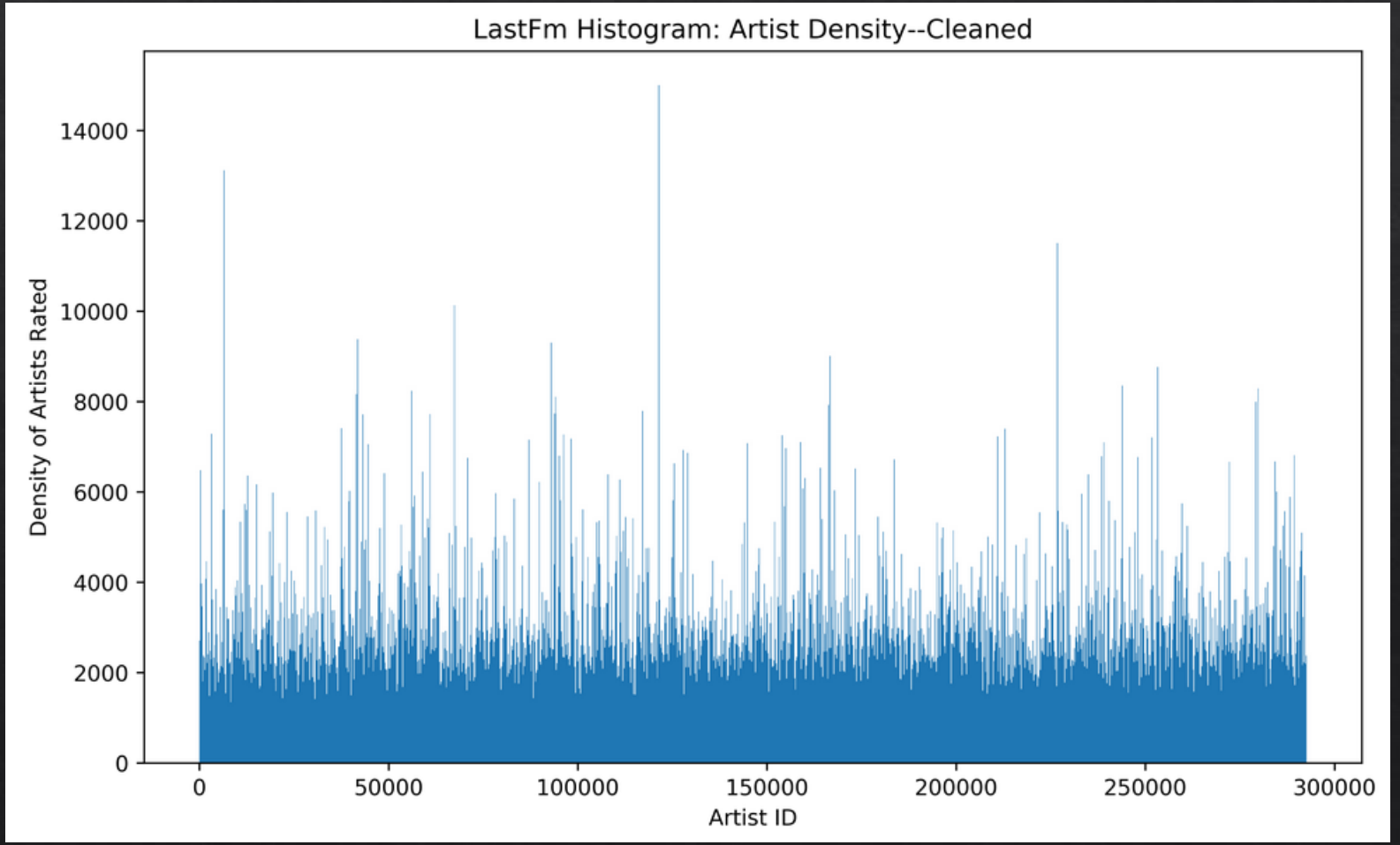
Descriptive Analytics:

Before:



Descriptive Analytics:

After:



Descriptive Analytics:

We took out a good portion of ratings, since they were all for the same artists. It seems that the dataset was dominated by 10,840 artists, or a little less than 5%. This 5% made up about 79% of the whole dataset. There are a little over 100 artists who had over 50k ratings. We are left with a dataset of only 4,673,605. But we will get a better model, without the added bias of the artists with an extremely high number of ratings.

We lost 6% of the data to users with a huge number of listens. We lost an additional 67% of the data when we got rid of all but a maximum of 180 ratings per artist. But if we are going to build an accurate model, we cannot let popular artists, or suspicious values skew our data. Quality of data will always trump quantity.

We are ready to start creating our model, and again push a final version of our data to Hadoop.

Building the Model:

About ALS Algorithm:

- ✓ Works well with parallelization, due to the (m + y) + (n +y), the matrix A = XY, and X = mn and Y is the transpose of m. Basically the cost is additive so to speak, not multiplicative. It is multiplicative only when needed.
- ✓ The goal is the assumption that the complete ratings matrix R is approximately equal to transpose(X)Y, where R is the ratings matrix.
- ✓ It is called alternating, because the transpose(X) side and the "Y" or predictor side are optimized alternatingly; in other words, once Y is optimized, for errors, X is optimized. When both sides are optimized, the algorithm ends, or converges.
- ✓ Spark has the mllib (Machine Learning Library) from which we can use the recommendation module that contains an implementation of ALS among other algorithms. The ALS algorithm can be tuned to “build” recommendations for explicit ratings, and implicit ratings. In other words ratings that are on a pre-defined scale and explicitly chosen to represent a user’s sentiment for the item, or implicitly chosen, since if one likes a song one listens to it a high number of times. Again, our case is the later.

Building the Model:

Hyperparameters:

In predictive analytics we have something called hyperparameter optimization. We have these algorithms that take parameters, which are important to get the best model.

Often, you can run an algorithm over-and-over with different parameters record the results of each round and combination of parameters and choose the best set of parameters at the end.

Building the Model:

In the past, I have noticed that the implicit recommender works fairly well for things like "number of listens," or "number of sales," if the predictor's interval is somewhat

compact. In order to compact the interval a bit more (normalize), I divided each value by the maximum value to obtain an interval of [0,1].

In the forthcoming slides we see two models, one using PySpark's ALS implicit and explicit options. The explicit model seemed to converge very quickly, and had an RMSE of .12.

Hyperparameter- finding techniques did little good for either model, and for the implicit model found hyperparameters that resulted in an overfit model.

On the right, we see the hyperparameters for Pyspark's ALS algorithm.

```
Optimal hyperparameters
alpha              40.000000
min_error                inf
model_number       82.000000
n_factors          80.000000
n_iter             10.000000
second_best         0.000000
train_mse           0.138185
dtype: float64
```

Building the Model:

Implicit Model:

```
model = ALS.trainImplicit(rdd_training, rank=rank, iterations=n_iter, alpha=alpha, lambda_=lambda_,
predictions = model.predictAll(predict_test).map(lambda r: ((r[0], r[1]), r[2]))
rates_and_preds = rdd_test.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print('The error for the test data is {:.2f}'.format(error))
```

```
The error for the test data is 0.25
```

Above, we can see the results after making "educated guesses" for the model. Below we see the results after tuning.

```
predict_test = rdd_test.map(lambda x: (x[0], x[1]))

predictions = complete_model.predictAll(predict_test).map(lambda r: ((r[0], r[
rates_and_preds = rdd_test.map(lambda r: ((int(r[0]), int(r[1])), float(r[2])))
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print('For testing data the RMSE is {:.2f}'.format(error))
```

```
For testing data the RMSE is 0.17
```

Building the Model:

Explicit Model:

```
New optimal hyperparameters
lam                                                          0.01
min_error                                                     inf
model          <pyspark.mllib.recommendation.MatrixFactorizat...
n_factors                                                      80
n_iter                                                         20
test_mse                                                  0.12245
dtype: object
```

Above, we can see the results after making "educated guesses" for the model. Below we see the results after tuning.

```
# Now for the tst dataset
model = ALS.train(rdd_training, rank=5, iterations=5, seed=5)
predictions = model.predictAll(predict_test).map(lambda r: ((r[0], r[1]), r[2]))
rates_and_preds = rdd_test.map(lambda r: ((int(r[0]), int(r[1])), float(r[2])))
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print('The error for the test data is {:.2f}'.format(error))
```

```
The error for the test data is 0.12
```

Building the Model:

Now that we have selected the “best” hyperparameters, we can build our model.

Demonstration of Model:

To the right, we see the "ratings" for a new user—me. The same user and ratings was entered for each model. Initial runs of the algorithm gave me results like Madonna, Miley Cyrus, and non-music results which I penalized by rating them .0001 and running the algorithm again.

```
[('56','3018',.0001),
('56','153162',.0001), #Not an actual artist
('56','288963',.0001),# Alain Dzukam. Another
 ('56','14368',.0001), #David guetta. This is a
 ('56','204066',.1), #Rihanna
 ('56','174344',.0001),# Kaiser Chiefs
 ('56','95545',.0001), #Iron maiden
 ('56','58275',.0001),# Dire straits
 ('56','43548',.18),#Regina Spektor
 ('56','68390',.19), #Artst is 'edith piaf'
 ('56','281461',.20), #Artist is 'maria callas'
 ('56','201482',.17),# Atist is 'bryn terfel'
 ('56','243041',.17), # Anna netrebko
 ('56','244054',.17), # The residents
 ('56','256484',.18), #The Legendary Pink Dots
 ('56','206805',.17), #bjork
 ('56','127801',.0001), #Madonna
 ('56','96194',.16), #The killers
 ('56','216449',.0001), #Taylor Swift
 ('56','113418',.0001), #Miley Cyrus
 ('56','253951',.0001), # recommended: Madonna
 ('56','279301',.0001) # recommended: t-pain
```

Demonstration of Model:

Recommendations from implicit model:

```
print('Your top ten recommended artists:')
for x in top_items:
    artist = str(x[1])
    print(set(artist_name_lookup_table.lookup(artist)))
```

```
Your top ten recommended artists:
{'modest mouse'}
{'eric clapton'}
{'andrew bird'}
{'die Ärzte'}
{'blink-182'}
{'yann tiersen'}
{'bob dylan'}
{'nightwish'}
{'u2'}
{'katie melua'}
{'band of horses'}
{'the future sound of london'}
{'the prodigy'}
{'tiësto'}
{'finntroll'}
{'george michael'}
{'bright eyes'}
{'extremoduro'}
{'herbie hancock'}
{'stars'}
```

Demonstration of Model:

Recommendations from Explicit model:

```
print('Your top ten recommended artists:')
for x in top_items:
    artist = str(x[1])
    print(set(artist_name_lookup_table.lookup(artist)))
```

```
Your top ten recommended artists:
{'bippp (v.a.)'}
{'heinz holliger - zehetmair - larcher - holliger - etc.'}
{'dj philip'}
{'active coma'}
{'alberto cortéz y facundo cabral'}
{'jens rachut'}
{'marco antonios solis'}
{'bellhouse'}
{'mike shiflet'}
{'digital mystikz & loefah'}
{'andy vores'}
{'andrew davis'}
{'electric eel-shock'}
{'alexander schatten'}
{'lech jankowski'}
{'manu chao !'}
{'c.a.r.n.e.'}
{'□□'}
{'die dødelsäcke'}
{'amor-te'}
```

Discussion:

Neither model does too badly, and the implicit model does a decent job of picking up on my tastes. However, it is no surprise we cannot generalize this model as something useful. But the good news is, for once it *is* the data's fault.

We have some recommendations for artists that are only on Last FM. Even more troublesome, I got several recommendations for what were supposed to be artist names, but in fact were movies or even clips for movies. These should not be in this dataset.

Despite the above, this dataset is great to practice building a recommender, because it is

fairly big and it is dirty. Perhaps the model could be decent if one could take all

the non-music non-Last-Fm-exclusive and non-music content out.

Discussion:

ALS is not necessarily the easiest algorithm to implement, because we need the input tuple to be numeric. In the case of this dataset, we only had one numeric column. In fact, that's why this dataset and ALS was chosen. As a data scientist, you should be able to "recode" a column to fit your needs.

Furthermore, rarely will one find the best model, the best hyperparameters, or the right permutation of the dataset straightaway. It takes work, a procedural attitude towards model development, and an "intuition" of what is working and what can be improved.

An RMSE of .12 and a model which seems to be fit quite well, is an accomplishment for what appears to be a terribly noisy dataset.

One final note: I mentioned when discussing parameter tuning that the implicit algorithm seems to works best with a compact interval. The minimum value of the interval is 0, while the max is 1. The mean is .24. Most of the values are less-than .5. We do not have a compact interval, but an erratic one. Perhaps trying different normalization techniques will result in a better model.

Conclusion:

In this project, 17+ million artist ratings were loaded into a Hadoop cluster. Using PySpark and a Jupyter Notebook, the encrypted user id was recoded into a number that is less that the maximum Java int. An artist id was then created for each artist.

Looking at descriptive statistics, outliers for number of listens were removed, and regularization for artist listens was performed. The final output was then modeled.

Loaded in the third and last notebook, the "clean" dataset was then broken into the seven-fold datasets. An implicit and explicit model was compared, with a graph used to help find the ideal hyper-parameters. The best parameters and the best model were then selected.

Finally, the completed model was demonstrated.