SEIS | 736

Analyzing song title term frequency in the Million Song Dataset

Jason Baker
Graduate Programs in Software
Fall 2014

University of St. Thomas, St. Paul, Minnesota

Overview

Researchers and businesses are using tools such as Hadoop to solve Big Data problems. The Hadoop platform allows data scientists to rapidly sift through enormous amounts of data using distributed, low-cost computing infrastructure. Hadoop utilizes a map-reduce programming model to process data in a distributed manner. The purpose of this paper is to describe my Hadoop project focused on the term frequency analysis of one million song titles. The paper will describe the nature of the analysis, the map-reduce programming algorithms, and the challenges encountered while working with a large dataset.

Data Source

One of the first challenges with the assigned Big Data project was identifying a suitable dataset. The only project requirement we were given was that the dataset could not be larger than 10GB in size. I recognized three important qualities a potential project dataset should have:

1) The dataset should be large enough to be considered Big Data, yet not so large as to exceed the project constraint.

2) The data elements in the dataset should be accessible and not stored in a proprietary data format.

3) The dataset should contain a variety of data elements, offering a number of different paths for data exploration.

After considering numerous datasets, I selected the Million Song Dataset curated by the Laboratory for the Recognition and Organization of Speed and Audio (LabROSA) at Columbia University (http://labrosa.ee.columbia.edu/millionsong/). The Million Song Dataset represents a free collection of song and audio metadata derived from one million contemporary popular music tracks. The songs span a wide variety of musical genres – from rock to blues to classical. The dataset does not include any actual audio data. It contains the audio metadata associated with songs such as the number of song measures, tempo, and loudness as well as key song attributes such as the song title, artist name, and release date.

The Million Song Dataset was created by a partnership between LabROSA and a company called The Echo Nest. This company is focused on curating millions of songs for the purpose of delivering useful analytics to other businesses. For example, one of the company's core products is a song recommendation engine used by Internet music streaming companies like Spotify.

LabROSA and The Echo Next have made the Million Song Dataset available to researchers to encourage further development of large-scale analytical algorithms.

These organizations hope that interested parties will push the boundaries of current Music Information Retrieval (MIR) technologies.

As a musician and avid music listener, I eagerly selected the Million Song Dataset for my project. In hindsight, this eagerness should have been tempered by more due diligence as I began to face the challenge of working with Big Data.

Data Acquisition

The Million Song Dataset is comprised of one million individual data files organized in a multi-directory hierarchy. Each data file contains the metadata associated with one song track – the artist name, release name, audio analysis data, etc. The dataset developers could not put a million files in a single folder, so they devised a file directory hierarchy to store the individual files. Each song track has an associated Echo Nest track ID, and this ID was used as the basis for the directory naming structure. Every song file is located within a directory folder named after the 3rd, 4th, and 5th letters of the corresponding track ID. For example, a typical directory structure looks like this:

```
A ->
        A ->
        …
        D ->
                A ->
                B ->
                …
                H -> TRADHRX124352CD29834.h5
                …
        E ->
        …
B ->
…
```

The Million Song Dataset is comprised of over a million files and over 300GB of data. While the dataset was interesting and diverse, it greatly exceeded the size constraints for this project. Fortunately, LabROSA created a smaller dataset containing 10,000 song files for the purpose of testing algorithms on a smaller and more manageable set of data. My intention was to use this smaller dataset for my project since, at 1.8GB, it neatly fit into the size constraints for the project. That plan was sensible until I started looking more closely at the data.

Hadoop supports a variety of data input formats for the map-reduce programming model. During our Big Data Architecture course, we primarily worked with text or key-value data input formats. These input formats are relatively easy to work with as long as the data is stored in a format that can be parsed by the mapper programmatically. I discovered that the data files in the Million Song Dataset were stored in a format that was not easily digestible by Hadoop – a data format called HDF5.

HDF5 is a data storage format that was developed by NASA to support "large, heterogeneous, hierarchical datasets" (hdfgroup.org). HDF5 is a binary storage format that supports complex data types and data compression. Tools such as Matlab can work with HDF5 files natively. Also, a number of Python, Java, and C programming libraries are available to interact with HDF5 files.

My initial excitement working with the Million Song Dataset was tempered by the additional work required to figure out a new data format. I was not able to find any resources showing how to consume HDF5 data using Hadoop. Therefore, I went to work trying to figure out how to convert HDF5 files to a more useful format, such as TSV text files.

The HDF Group, developers and maintainers of the HDF data format, did not offer any software that specifically translated HDF5 files to a TSV file format. However, LabROSA had some software code in Github that in theory provided the data format translation I needed.

The LabROSA software was written in Java and required the installation of specialized Python and Java libraries on my workstation. Basically, the software recursively read HDF5 files from a set of file directories and stored the data field values in appropriate in-memory data structures. The data was then written back to disk in a TSV text format.

After downloading and building the application in Eclipse, I tested it on a directory containing a dozen song data files. I encountered a number of issues while executing the software. For example, the number of columns was not consistent from one row of data to the next. This was due to a bug in the software where columns were not created for null data elements. After correcting a few defects, I tested the conversion software against the 10,000-song dataset and found that only about 50% of the song tracks were converted properly. The data records for the remaining tracks were skipped due to a variety of data conversion errors. After a full day of trying to get this software to work properly, I gave up and decided to follow a different path.

Switching to Plan B

The 10,000-song dataset was appealing because of its smaller size and the fact that it neatly fit the storage constraints of the project. However, the data wasn't in a format that was readily accessible to Hadoop. I began to search for alternate sources of the Million Song Dataset, preferably data sources that were already in a text file format.

The Echo Nest periodically sponsored community challenges using the Million Song Dataset. They would provide community participants with access to the dataset along with anonymized listener playlists. The goal was to see if the community could come up with a better music recommendation algorithm for music listeners. After researching a few of these challenges, I discovered that The Echo Nest made the entire Million Song Dataset available to participants in TSV format on Amazon S3.

The dataset was easily accessible on Amazon S3, but it presented a number of logistical problems. First, the entire dataset was over 300GB – clearly too large for this project. Second, downloading this amount of data would potentially take several days on a slower broadband connection. Third, my workstation only had 40GB of storage space available. Finally, even if I could load the dataset on my workstation I couldn't work with the entire dataset in my local Cloudera VM or the UST cluster.

Despite these logistical challenges, I decided to forge ahead with using the full dataset. I knew that I could come up with a workflow that would allow me to partition and efficiently process the data. I also knew that ultimately I wouldn't need most of the actual data in the dataset. If I could pull out only the data I needed, I could parse the dataset down to a much more manageable working size for the project.

The full dataset contains dozens of fields related to various song metadata and audio attributes (see Appendix A). Several of the individual fields hold arrays containing dozens of values. I was mostly interested in analyzing term frequencies in song titles, so that meant I could eliminate 90% of the fields and reduce the amount of data in the dataset. I ultimately settled on the following list of data fields for this project:

Data field	Description
track_id	The Echo Nest track ID
artist_familiarity	The Echo Nest artist familiarity value
artist_hotttness	The Echo Nest artist hotness value
artist_name	name of the music artist
artist_terms	tags assigned to the song track (genre)
loudness	general loudness of the track
release	music album the track is located on
tempo	musical tempo of the song
title	song title
year	year the song was released

Some of the selected data fields contain audio attributes or other miscellaneous song rating data (i.e., artist familiarity). I decided to retain a few of these data fields for two reasons: 1) there was little cost involved in retaining the additional data, and 2) if I decided to take the project in a different direction I would have several data field options available without having to go back and parse the full dataset again.

Data Wrangling

The biggest challenge in this project was obtaining and parsing the Million Song Dataset. While the process was not necessarily technically challenging, it represented a significant labor investment over the course of several days. The full Million Song Dataset in TSV format is located on Amazon S3 at: https://tbmmsd.s3.amazonaws.com. The dataset is organized into 339 separate TSV files in the following file hierarchy:

A.tsv.a
A.tsv.b
…
A.tsv.m
B.tsv.a
…
Z.tsv.m

Each alphabetized set of files (A.tsv.a…A.tsv.m) represents approximately 8GB of data. Due to storage constraints on my laptop, I could only work with a handful of files at one time. My local Cloudera VM represented an even greater constraint since the VM was installed with only about 5GB of free disk storage – not nearly enough storage to hold even one set of TSV files.

I solved the Cloudera VM storage restriction by reconfiguring the VM installation. I shut down the VM, increased the disk storage allocated to the VM, and booted the Cloudera system into single user mode. Once I was in single user mode, I modified the existing disk partition to include the newly allocated VM disk storage. I rebooted the virtual machine and used a linux command to add the unused disk partition space to the existing file system partition. Now I could work with much larger data files in the Cloudera VM.

I followed a relatively simple workflow to process each group of TSV files in the Million Song Dataset. The workflow included the following steps:

1. Download a group of TSV files from S3 (approximately 8GB data) to the Cloudera VM, storing them in a directory structure (ex. ../millionsong/A)

2. Add the group of files to HDFS.

3. Execute the Million Song Parse job on the group of files in HDFS, storing the result in a directory called millionsongparsed.

4. Copy the parsed file output to a new local directory (ex. ../millionsonparsed/A/A.tsv.snappy)

5. Clean up the HDFS working directories and repeat step 1 for each group.

The millionsongparse Hadoop job was designed primarily as a mapper. It performed several important functions including: reading the data, parsing out selected data fields, identifying incorrectly formatted fields, and producing a compressed binary file output. The job specified a single default reducer to ensure that all the data ended up in one output file. The Snappy codec was used to compress the file.

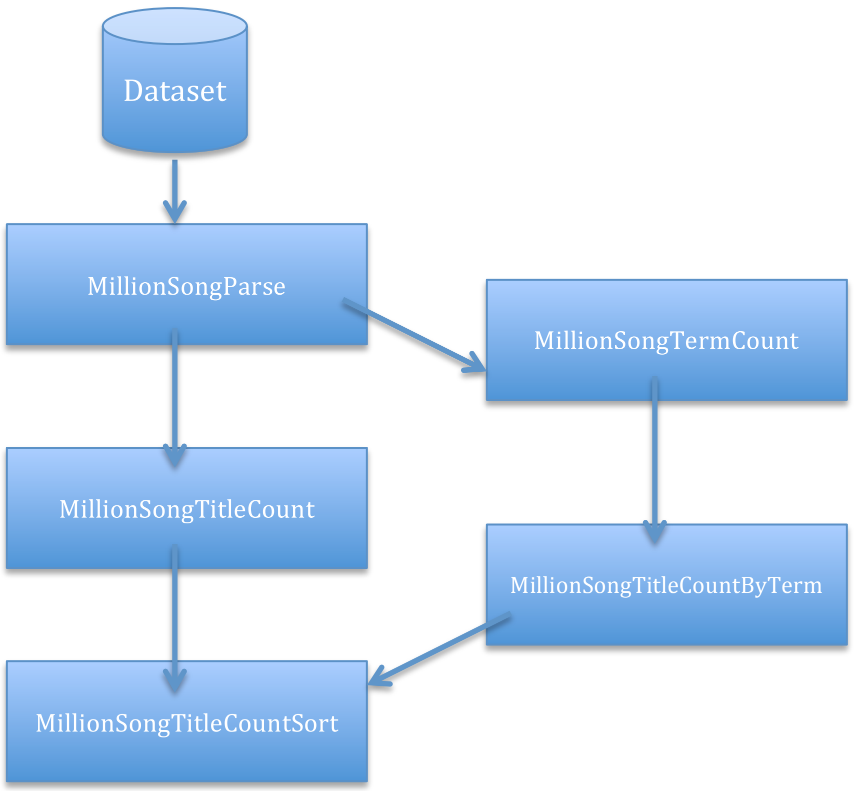
The millionsongparse mapper read each line from the tab-delimited text files and converted the lines into an array of strings representing each data field. If the line contained at least 53 fields, a new tab-delimited string was generated using specific array indexes. Counters were set based on whether or not the line contained valid data fields. Finally, a new key and value were sent to the default reducer. The unique track ID was used as the key and the new tab-delimited string was used as the value.

Each group of TSV files represented about 9GB of data. The millionsongparse job took approximately 4-5 minutes to execute for each file group. The total amount of time Hadoop spent parsing data is estimated around 120 minutes (note this includes job setup time for each group). The compressed output from each parse job resulted in a file that consumed approximately 9MB of disk storage – representing a 1000% reduction in storage. The full 300GB dataset was reduced to a mere 230MB of storage after the parsing process. It took two days to parse all 26 groups of TSV files.

Song Title Analysis

The purpose of this project was to analyze the frequency of terms in song titles, and the core algorithm used for this analysis was the basic WordCount map-reduce algorithm. If this represented the full complexity of my project, it wouldn't warrant much discussion. However, I expanded on this algorithm by introducing a few technical concepts that were discussed, but not implemented, in our class. I made use of Distributed Cache, MultipleOutputs, custom sorting and more complex key transformations to produce the project results. Ultimately, I designed a map-reduce workflow that executed a word count on song title terms, filtered out stop words, partitioned the terms into separate files based on music genre, and ranked the results.

I created five different map-reduce jobs and algorithms (see Appendix C) to analyze the Million Song Dataset. The first job, millionsongparse, was discussed in the previous section. Once the dataset was parsed, four additional jobs were used to process the data. The job workflow looked like the following:



The MillionSongTitleCount job performs a word count of the terms in each song title. It sounds fairly simple, but this job implements one nifty trick. The job uses the Hadoop Distributed Cache to filter song title terms using a hashed stop word list.

One of the challenges with counting terms in song titles is that titles can contain a copious amount of extraneous words. Oftentimes the words describe some particular aspect of the recording, such as: digital, live, acoustic, remastered, unreleased or remixed. For example, the song title could be "All about that bass (DJ Freddie Remix)". For the purposes of this project, I contend that the first four terms constitute the song title and the remaining terms are simply additional metadata. The map-reduce process needs to filter out words such as "remix" otherwise these metadata terms would skew the final results.

Not only do song titles have additional metadata, but also some of the words in the titles add very little meaning – words such as "has, is, and, the". These words are so commonly found in the English language that their presence could also skew the word count results. If we discovered that the most common word found in one million song titles was the word "the", would this really tell us anything?

I used a custom stop word list to filter out less valuable terms from the word count algorithm. I started with a copy of Fox's 421-word list (Fox 1990) as the basis for the stop list. The complete Fox stop list was a little too aggressive, so I ultimately removed approximately 30% of the terms (see Appendix C). A word such as "you" may not have much meaning in a general document, but in a song title it takes on greater significance.

I also ran a number of MillionSongTitleCount jobs and looked at the set of highly ranked words. I was able to iteratively add new words to the stop list as extraneous words bubbled up the ranking order. One caveat is that a few valid words were likely filtered by the stop list. For example, the word "live" is likely found in a number of song titles. However, it's also found as metadata in thousands of titles: i.e., "What you mean to me (Live version)". I was fairly aggressive in filtering out metadata for this project, and I likely sacrificed a bit of accuracy as a result.

The MillionSongTitleCount job adds the stop word list to the Distributed Cache at the start of the job. The stop word list is a simple text file containing one word per line. The mapper job locates the stop word file in the Distributed Cache and loads each word from the file into a hash set. This is a useful data structure for the algorithm because the mapper will compare every single word in the dataset to the stop word list. The hash set offers the fastest mechanism for this comparison. During the processing of the dataset, the stop list filter removed 1,415,493 words from the mapper output.

The mapper algorithm reads in each song record from the Million Song Dataset and retrieves the song title. It splits the song title terms into a string array and compares each array element with the stop word hashset. If a word is not located in the stop word list, the mapper emits the word as the key with a value of 1 to the reducer. The MillionSongTitleCount reducer sorts the words and generates a total count for each word.

The terms are now listed in a single file along with their word counts. The MillionSongTitleCountSort job produces a sorted ranking of the terms in descending order. The MillionSongTitleCountSort mapper uses the KeyValueTextInputFormat to read in each word and word count from the preceding job output. The mapper swaps the key and the value and emits the new pair to the default reducer. The job uses a custom integer comparator to properly sort the keys before output. The result of this job is a ranked list of the most common words used in one million popular song titles. The top-20 words are listed below:

<u>Count/ term</u>

49575	you
28806	love
26220	my
9422	time
8262	we
8230	man
7553	down
7148	out

6651	life
6519	night
6477	like
6432	world
6161	day
5903	back
5794	heart
5777	baby
5648	little
5214	good
4986	girl
4915	take

One more challenge

Parsing 300GB of data was time consuming, but not necessarily technically challenging from an implementation perspective. The word count algorithm was relatively straightforward to code and test. The use of stop word filtering and the Distributed Cache definitely increased the complexity of the project. The Distributed Cache was challenging to implement due to the variety of conflicting code examples on the Internet due to the API differences between Hadoop 1 and 2. Trying to unit test code using the Distributed Cache was a whole project in itself because the MRUnit library version (0.8) installed on the Cloudera VM did not support Distributed Cache unit testing. I had to download and install MRUnit 1.0 in order to unit test the cache code properly. Once I had pushed through a seemingly never-ending set of roadblocks, I decided to add one more layer of complexity.

The MillionSongTitleCount and MillionSongTitleCountSort jobs produce a ranked list of the most frequently used words in song titles. Not just any song titles, but all one million song titles. I wondered if the ranked list of words would differ from one genre of music to the next. Would the most common song title terms in rock music be the same as hip hop or country? I wanted to find out.

In order to understand the challenges related to partitioning the data, I needed to know the types of music genres stored in the dataset. Additionally, I thought that understanding how many song tracks were associated with each genre would be useful. A music genre with thousands of associated songs was likely a better sample set than one with only dozens of songs.

The MillionSongTermCount job counts the number of songs associated with each music genre in the dataset. The mapper reads in each line of the parsed song dataset, and extracts the artist term list for each track. The list is a comma-delimited string of music genres associated with the track. The first element in the list is the most relevant music genre, and each successive element in the list has a decreasing level of relevancy. The MillionSongTermCount job retrieves the first term from the list and uses this term to categorize the music genre for the song track. The mapper job outputs the music genre as the key with an associated value of 1. The reducer job simply sorts the list of music genres and counts the values to determine a total count of songs associated with each genre (see Appendix B).

The results I desired were fairly clear, but the implementation was not. The goal was to partition the words into separate files based on the music genre. My first thought was that a custom partitioner was just the thing I needed to solve this problem.

A custom partitioner presented a few challenges though. A partitioner is dependent on the number of reducers configured for the job. I would need to configure one reducer for each type of music genre, and with over seven hundred different genres that seemed like overkill. The number of reducers is a static number. If the data changed and more types of music genre were added, I would have to go back and modify the number of reducers.

Another challenge was figuring out how to design a partitioning algorithm. The algorithm would need to use a hash or assign a value to each music genre and use that to determine the reducer. After spending a few hours going down this implementation dead end, I knew there had to be a better way.

Hadoop's MultipleOutputs functionality seemed like a logical way to partition data based on the music genre. The feature isn't dependent on the number of reducers employed by the job. Data is partitioned after being processed by the reducer versus the mapper. MultipleOutputs provides a great deal of control over the file naming convention of the resulting output files, a big bonus when you are dealing with hundreds of different files.

The MillionSongTitleCountByTerm job reads the parsed Million Song Dataset, calculates a filtered song title term count, and creates an output file for each music genre. It does this by building on the algorithm used by the MillionSongTitleCount job and adding a little key manipulation trickery.

Like its parent, the MillionSongTitleCountByTerm job reads in each record from the Million Song Dataset files. The song title is retrieved from the record, parsed into individual words, and filtered against the stop word hashset. The job also retrieves the music genre most closely associated with the track – the first term in the comma-delimited artist terms data field. Instead of just emitting a word and the value 1 like MillionSongTitleCount, the job emits a compound key comprised of the music genre and a title word separated by a comma. For example, the mapper might emit data like this:

```
rock,love	1
hip-hop,baby	1
country,sweet		1
```

The reducer sorts and calculates the total word count of each of the compound keys. The fact that the keys are comprised of two different values is useful from a calculation standpoint because it actually partitions the calculations. After calculating the total word count, it splits the key back into two elements – the music genre and the word. The music genre string is passed to the MultipleOutputs object and used to generate the output filename. Whitespace is stripped from this name to prevent any awkward file naming issues. Finally, the MultipleOutputs object writes the song title word and the total count as a record in the specified output file.

Selected output files were sorted by the MillionSongTitleCountSort job to produce the following results (only top-10 counts listed):

Hip Hop		**Pop Rock**		**Blues Rock**	
901	you	1657	you	1357	you
568	my	935	love	819	love
477	love	741	my	756	my
344	we	279	time	321	down
236	out	229	man	308	baby
236	down	227	down	304	man
229	life	226	we	296	time
202	like	223	heart	211	out
201	back	220	like	209	back
194	man	206	girl	193	night
Vocal Jazz		**Techno**		**Country Blues**	
1066	you	94	you	231	you
614	love	80	my	226	my
446	my	56	love	118	man
157	time	44	time	112	down
118	man	37	life	94	take
116	heart	34	out	74	love
107	little	33	part	70	baby
99	day	33	we	63	old
98	know	30	more	62	woman
95	baby	29	world	56	good

Trusting the data

One of the challenges a data scientist is presented with in any Big Data project is understanding what types of questions to ask. The Million Song Dataset includes dozens of data fields for each individual song representing hundreds of millions of discreet data elements. Once you determine your path of inquiry and design map-reduce algorithms to probe the data, you are faced with an even greater question: Did my algorithms produce the correct results? The nature of Big Data is that it's too expensive to calculate the results manually, so a certain level of confidence is required when reviewing results.

I used three different processes to increase the level of confidence in the project results: the sensibility test, unit testing, and custom counters. This three-layered approach ensured that the algorithms met design requirements, the map-reduce jobs executed properly, and the results looked sensible. I also used an iterative development approach by testing immediately after design and then refining the design based on testing results.

I made heavy use of the Apache MRUnit test suite throughout the project, even updating the MRUnit library at one point simply to test the Distributed Cache functionality. I developed unit tests for each mapper and reducer in all five jobs. The unit tests incorporate real data from the Million Song Dataset wherever possible. This provided greater assurance that the map-reduce algorithms would work properly on the full dataset. I ran unit tests on individual map-reduce packages any time code modifications were made in order to identify any regression bugs.

Unit tests can show you that your algorithms are working with a known set of test data, but the tests can't tell you how well the algorithms will process large amounts of real data. Every job incorporates a set of custom counters to track various job statistics such as: the number of valid or invalid records processed, the number of words filtered by the stop words list, and whether or not the Distributed Cache file was loaded properly. The counters provide a way to monitor the execution of the jobs and identify any possible implementation issues. For example, the counters can show whether or not a mapper is actually using the stop words list to filter words. If this filter wasn't working properly the results could be skewed.

Finally, when looking at data results throughout the development of this project, I asked myself a simple question: Do these results make sense? You could call it the sensibility test or the smell test. I think that logical reasoning and intuition are powerful tools that any data scientist can use to analyze results. The first few MillionSongTermCount jobs I ran yielded high rankings for words such as "live" and "remix". I was surprised to see these words at first, but after reviewing the song titles more closely I recognized that these metadata terms were quite common in the dataset. The results actually taught me something I didn't know about the data, and I used these results to refine the stop word filtering. Eventually I ended up with results that seemed entirely plausible. It's interesting that the top words in song titles are words like "you", "love", and "baby". However, it's not at all surprising based on what I know as a music listener. The sensible results, combined with unit testing and execution counters, provided a high degree of confidence in the project results.

Lessons learned

I learned a number of important lessons while working on this Big Data project. Approximately 70% of my time was spent acquiring and processing the Million Song

Dataset, while 30% was spent on coding and implementation (not including this report). I should have spent more time upfront reviewing the HDF5 data format and understanding how to convert it into a more Hadoop-friendly format. Fortunately, the TSV dataset I found was fairly clean – with 996,217 records out of 1 million reported as valid.

The full dataset is comprised of hundreds of files consuming 300GB of storage. My parsed version of the dataset was a single 230MB file. In some ways, using Hadoop and map-reduce to process this parsed dataset was like swatting a fly with a hammer. Data I/O wasn't a performance constraint and a full Hadoop cluster was not really necessary to process the parsed data. Spark and other memory-based analytical solutions are probably better tools for this type of dataset.

Working with some of the less well-documented features of Hadoop such as Distributed Cache and MultipleOutputs was challenging, especially since the API's changed from Hadoop 1 to Hadoop 2. It was difficult to find working Hadoop 2 example code for these features– especially MultipleOutputs.

It seems like testing is always a challenge no matter what language or platform you are using. Hadoop is especially difficult to test because the testing documentation for many features is virtually non-existent. MRUnit support for some features like Distributed Cache and MultipleOutputs has only arrived in the past year.

Conclusion

Hadoop is an efficient and flexible platform for analyzing massive amounts of data. I was able to take earlier lessons from our Big Data course and extend those lessons using more advanced Hadoop features. I used those features to determine the most commonly used words in one million song titles, filtered by a custom stop word list, and partitioned by music genre. One of the more interesting conclusions from this project is that song title terms are relatively consistent from one genre of music to the next. The word "love" was found in the top-5 of most music genre lists. Perhaps Paul McCartney was right, people haven't had enough of silly love songs.

Appendix A: Million Song Dataset Fields

Field name	Value	Description
artist_mbid	**db92a151…**	musicbrainz.org ID…
artist_mbtags	**shape = (4,)**	this artist received 4 tags on musicbrainz.org
artist_mbtags_count	**shape = (4,)**	raw tag count of the 4 tags this artist received on musicbrainz.org
artist_name	**Rick Astley**	artist name
artist_playmeid	**1338**	the ID of that artist on the service playme.com
artist_terms	**shape = (12,)**	this artist has 12 terms (tags) from The Echo Nest
artist_terms_freq	**shape = (12,)**	frequency of the 12 terms from TEN (number between 0 and 1)
artist_terms_weight	**shape = (12,)**	weight of the 12 terms from TEN (number between 0 and 1)
audio_md5	**bf53f8113…**	hash code of the audio used for the analysis by The Echo Nest
bars_confidence:	**shape = (99,)**	confidence value (between 0 and 1) associated with each bar
bars_start	**shape = (99,)**	start time of each bar according to TEN this song has 99 bars
beats_confidence	**shape = (397,)**	confidence value (between 0 and 1) associated with each beat
beats_start	**shape = (397,)**	start time of each beat according to TEN, this song has 397 beats
danceability	**0.0**	danceability measure of song (between 0 and 1, 0=>not known)
duration	**211.69587**	duration of the track in seconds
end_of_fade_in	**0.139**	time of the end of the fade in, at the beginning of the song,
energy	**0.0**	energy measure (between 0 and 1, 0 => not analyzed)
key	**1**	estimation of the key the song is in by The Echo Nest
key_confidence	**0.324**	confidence of the key estimation
loudness	**-7.75**	general loudness of the track
mode	**1**	estimation of the mode the song is in by The Echo Nest
mode_confidence	**0.434**	confidence of the mode estimation
release	**Big Tunes…**	album name from which the track was taken
release_7digitalid	**786795**	the ID of the release (album) on the service 7digital.com
sections_confidence	**shape = (10,)**	confidence value (between 0 and 1) associated with each section
sections_start	**shape = (10,)**	start time of each section according to TEN
segments_confidence	**shape = (935,)**	confidence value (between 0 and 1) associated with segment
segments_loudness_max	**shape = (935,)**	max loudness during each segment
segments_loudness_max_time	**shape = (935,)**	time of the max loudness during each segment
segments_loudness_start	**shape = (935,)**	loudness at the beginning of each segment
segments_pitches	**shape = (935, 12)**	chroma features for each segment (normalized so max is 1.)
segments_start	**shape = (935,)**	start time of each segment (~ musical event, or onset)
segments_timbre	**shape = (935, 12)**	MFCC-like features for each segment
similar_artists	**shape = (100,)**	a list of 100 artists (their Echo Nest ID) similar to Rick Astley
song_hotttnesss	**0.8642488**	this song had a 'hotttnesss' of 0.8 (on a scale of 0 and 1)
song_id	**SOCWJDB…**	The Echo Nest song ID
start_of_fade_out	**198.536**	start time of the fade out, in seconds, at the end of the song,
tatums_confidence	**shape = (794,)**	confidence value (between 0 and 1) associated with each tatum
tatums_start	**shape = (794,)**	start time of each tatum, this song has 794 tatums
tempo	**113.359**	tempo in BPM according to The Echo Nest
time_signature	**4**	usual number of beats per bar
time_signature_confidence	**0.634**	confidence of the time signature estimation
title	**Never Gonna..**	song title
track_7digitalid	**8707738**	the ID of this song on the service 7digital.com
track_id	**TRAXLZU1…**	The Echo Nest ID of this particular track
year	**1987**	year when this song was released, according to musicbrainz.org

Appendix B: Sample list of music terms and corresponding track count

2 tone	250	country blues	3932	german rap	11	melodic metalcore	410
8-bit	335	country gospel	1617	germany	80	melodic trance	972
80s country	2	country music	402	ghetto tech	441	memphis blues	153
90s	11	country rock	17183	glam	31	memphis rap	73
acid jazz	1658	cover	34	glam metal	636	memphis soul	459
acoustic	205	cowboy	27	glam rock	2517	mento	20
adult contemporary	24	cowpunk	143	glitch	2772	merengue	1452
africa	42	creative commons	1	glitch hop	10	merseybeat	460
african	44	crossover thrash	242	gnawa	37	metal	120
afrobeat	1252	crunk	876	goa	226	metalcore	248
all-female	546	crust punk	280	goregrind	403	mexico	250
alternative	561	cuba	10	gospel	917	miami bass	68
alternative country	2000	cuddlecore	14	gothic	87	minimal	188
alternative dance	3404	cumbia	1915	gothic metal	402	minimal house	16
alternative hip hop	1633	dance	744	gothic rock	417	minimal techno	30
alternative metal	6108	dance music	104	greece	3	minnesota	204
alternative pop	25	dance pop	7717	greek	441	mm	10
alternative pop rock	107	dance rock	2454	greek music	1	modern laika	120
alternative rap	1	dance-punk	666	gregorian chant	1	modern rock	5292
alternative rock	7124	dancehall	7134	grime	1978	mondiovision	57
ambient	114	danish	18	grind	4	motown	2633
ambient black metal	28	dark ambient	906	grindcore	3094	motown and soul	3
american	9	dark cabaret	98	groove	1	movie	4
americana	164	dark pop	59	groove metal	418	murga	71
anime	4	dark wave	3791	grunge	2648	musette	323
anti-folk	508	darkcore	23	guitar	238	musica	23
arabesque	160	death core	1177	guitarist	10	musical theater	15
arabic	9	death metal	2856	gypsy jazz	851	nardcore	163
argentine rock	50	death/doom metal	35	gypsy punk	3	national socialist black metal	28
art rock	6400	deathgrind	292	happy hardcore	2585	nature	1
avant-garde	29	deathrock	361	hard bop	5471	nederpop	126
avantgarde metal	228	deep funk	1	hard house	3345	neo classical metal	386
bachata	443	deep house	9115	hard rock	4376	neo soul	5050
baile funk	17	delta blues	2317	hard trance	11028	neo-progressive	1878
ballad	12523	desi	178	hardcore	199	neoclassical	381
ballet	95	detroit	20	hardcore hip hop	64	neofolk	2573
banda	570	detroit rap	16	hardcore metal	157	net label	12
baroque	1	detroit techno	287	hardcore punk	4345	neue deutsche welle	183
baroque music	108	deutsche schlager	21	hardcore rap	87	new age	257
bass	31	deutschsprachig	16	hardcore techno	320	new age music	30
bass music	1888	digital hardcore	282	hardstyle	1339	new beat	1213
bastard pop	8	dirty rap	419	hardtechno	8	new jack swing	993
batucada	52	dirty south rap	4133	harmonica blues	853	new orleans blues	264
beach music	116	disco	11755	heartland rock	598	new orleans jazz	326
beat	745	disco house	442	heavy metal	10888	new prog	24
beatboxing	53	disco polo	36	highlife	85	new rave	697
bebop	1430	dj	72	hip hop	29941	new romantic	1417
bel canto	97	doo-wop	4387	hip house	1068	new wave	8671
belgium	14	doom metal	2074	hip pop	144	new weird america	673

Appendix C: Project Source Code & Resources

MillonSongParse
https://github.com/jasondbaker/seis736/tree/master/millionsongparse/src/stubs

MillionSongTermCount
https://github.com/jasondbaker/seis736/tree/master/millionsongtermcount/src/stubs

MillionSongTitleCount
https://github.com/jasondbaker/seis736/tree/master/millionsongtitlecount/src/stubs

MillionSongTitleCountByTerm
https://github.com/jasondbaker/seis736/tree/master/millionsongtitlecountbyterm/src/stubs

MillionSongTitleCountSort
https://github.com/jasondbaker/seis736/tree/master/millionsongtitlecountsort/src/stubs

Stop words list
https://github.com/jasondbaker/seis736/blob/master/millionsong-resources/stopwords.txt