

15000807

This thesis submitted by Mary Thrall in partial fulfillment of the requirements for the Degree of Master of Science at St. Cloud State University is hereby approved by the final

CREATING A SONG LYRIC CORPUS GENERATOR AND IDENTIFYING
VERB-PARTICLE CONSTRUCTIONS IN INFORMAL LANGUAGE

by

Mary Thrall

B.A., University of Minnesota, Minneapolis, 2011

A Thesis

Submitted to the Graduate Faculty

of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

St. Cloud, Minnesota

December, 2014

Patricia Hager
Dean

School of Graduate Studies

This thesis submitted by Mary Thrall in partial fulfillment of the requirements for the Degree of Master of Science at St. Cloud State University is hereby approved by the final evaluation committee.

Mary Thrall

Song lyric corpora - collections of text for use in linguistic analysis - have proven to be publicly unavailable despite many previous studies on the subject. We designed and created two programs which generate a song lyric corpus. These programs can be shared or distributed, and they avoid potential copyright issues while also allowing future researchers to generate corpora of popular song lyrics which contain songs as current as the previous week. We used these programs to create a corpus of approximately 800 modern US popular songs, the *Song Lyric Corpus*.

Verb-particle constructions, a type of multiword expression, often cause difficulties for natural language processing tasks because of their colloquial nature and variable syntax. We hypothesized that there would be a higher density of verb-particle constructions in song lyrics than in formal speech because song lyrics contain more colloquial speech.

We selected two corpora to perform a comparison of verb-particle construction occurrences. The *Song Lyric Corpus* generated during this study and a section of the *Brown Corpus* were tokenized and tagged using Python's Natural Language Processing Toolkit (Bird, Loper, & Rockwell, 2009). We then manually examined the resulting tagged corpora to identify verb-particle constructions.

The *Song Lyric Corpus* had more than five times as many verb-particle constructions as the more formal *Brown Corpus*, indicating that song lyrics are a promising future medium in which to study verb-particle constructions.

Bryant Jukstora

Chairperson

[Signature]

Brian Reese

December 2014
Month Year

Patricia Hughes

Dean
School of Graduate Studies

Approved by Research Committee:

Bryant Jukstora Chairperson

CREATING A SONG LYRIC CORPUS GENERATOR AND IDENTIFYING VERB-PARTICLE CONSTRUCTIONS IN INFORMAL LANGUAGE

Mary Thrall

Song lyric *corpora* - collections of text for use in linguistic analysis—have proven to be publicly unavailable despite many previous studies on the subject. We designed and created two programs which generate a song lyric corpus. These programs can be shared or distributed, and they avoid potential copyright issues while also allowing future researchers to generate corpora of popular song lyrics which contain songs as current as the previous week. We used these programs to create a corpus of approximately 800 modern US popular songs, the *Song Lyric Corpus*.

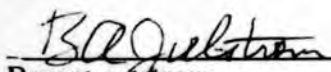
Verb-particle constructions, a type of multiword expression, often cause difficulties for natural language processing tasks because of their colloquial nature and variable syntax. We hypothesized that there would be a higher density of verb-particle constructions in song lyrics than in formal speech because song lyrics contain more colloquial speech.

We selected two corpora to perform a comparison of verb-particle construction occurrences. The *Song Lyric Corpus* generated during this research and the Editorials section of the *Brown Corpus* were tokenized and part-of-speech tagged using Python's Natural Language Processing Toolkit (Bird, Loper, & Klein, 2009). We then manually examined the resulting tagged corpora to identify verb-particle constructions.

The *Song Lyric Corpus* had more than five times as many verb-particle constructions as the more formal *Brown Corpus*, indicating that song lyrics are a promising future medium in which to study verb-particle constructions.

December 2014
Month Year

Approved by Research Committee:


Bryant Julstrom Chairperson

ACKNOWLEDGMENTS

A Note I would like to thank my committee members, Professors Bryant Julstrom, Andrew Anda, and Brian Reese. The Computational Linguistics Reading Group at the University of Minnesota offered helpful advice and kept me on task. This thesis would not have been possible without the help and constant encouragement of fellow graduate student Aleksandar Tomović throughout the past two years. My deepest gratitude goes to all.

A Note on Profanity

As is often the case when performing linguistic analysis on informal speech, certain words are considered by some to be profane or rude. Some of these words are included in this thesis, not to offend, but because to do otherwise would be to ignore important data and leave the analysis incomplete.

PREFACE

A Note on Gender

To avoid confusion, assumptions, or sexism, this thesis adopts the increasingly embraced convention of using third person plural pronouns while referring to a singular person of unknown gender. During discussions of others' work, if the gender of an author is not known, the pronouns *they/them/their/themselves* will be used, rather than *he/him/his/himself* or *she/her/her/herself*.

A Note on Profanity

As is often the case when performing linguistic analysis on informal speech, certain words are considered by some to be profane or rude. Some of these words are included in this thesis, not to offend, but because to do otherwise would be to ignore important data and leave the analysis incomplete.

ROYALS	12
Scrapy	12
Royals Design	16
Royals Implementation	19
Data Structures	22
THE SONG LYRIC CORPUS	32
Future Work	32

Chapter	Page
III. COMPARING VERB-PARTICLE CONSTRUCTIONS IN CORPORA OF VARYING FORMALITIES	34
MOTIVATION	34
Verb-Particle Construction	34
Machine Translation Example	35
Literature Review	36
LIST OF TABLES	ix
Goals of Verb-Particle Construction Analysis	37
LIST OF FIGURES	x
VPC IDENTIFICATION	37
Chapter	
I. INTRODUCTION	1
II. SONG LYRIC CORPUS GENERATOR	3
MOTIVATION	3
Automated Corpus Modifications: Tokenization and Tagging	40
Literature Review	3
The Brown Corpus	44
Reasons for Lack of Song Lyric Corpora	4
Manual Examination	45
SONG EXTRACTOR	6
FINDINGS	46
SongExtractor Design	8
Comparison of VPC Occurrences	46
ROYALS	12
New VPCs	48
Scrapy	12
Future Work	48
Royals Design	16
IV. CONCLUSION	50
Royals Implementation	19
REFERENCES	51
Data Structures	22
APPENDICES	
THE SONG LYRIC CORPUS	32
A. Source Code	56
Future Work	32

Chapter	Page
III. COMPARING VERB-PARTICLE CONSTRUCTIONS IN CORPORA OF VARYING FORMALITIES	34
MOTIVATION	34
Verb-Particle Construction Definition	34
Machine Translation Example	35
Literature Review	36
Goals of Verb-Particle Construction Analysis	37
VPC IDENTIFICATION	37
NLTK	37
Method Overview	38
Manual Corpus Processing	39
Automated Corpus Modifications: Tokenization and Tagging	40
The Brown Corpus	44
Manual Examination	45
FINDINGS	46
Comparison of VPC Occurrences	46
New VPCs	48
Future Work	48
IV. CONCLUSION	50
REFERENCES	51
APPENDICES	
A. Source Code	56

Chapter	Page
B. <i>SongExtractor</i> User Documentation	62
C. <i>Royals</i> User Documentation	65

LIST OF TABLES

Table	Page
1. XPath Navigation Operators	7
2. Results of Translating Sentences with and without VPCs	35
3. Findings of VPC Comparison	46
4. Newly Identified Verb-Particle Constructions	48
5.	51
6.	52
7.	53
8.	54
9.	55
10.	56
11.	57
12.	58
13.	59
14.	60
15.	61
16.	62
17.	63
18.	64
19.	65
20.	66
21.	67
22.	68
23.	69
24.	70
25.	71
26.	72
27.	73
28.	74
29.	75
30.	76
31.	77
32.	78
33.	79
34.	80
35.	81
36.	82
37.	83
38.	84
39.	85
40.	86
41.	87
42.	88
43.	89
44.	90
45.	91
46.	92
47.	93
48.	94
49.	95
50.	96
51.	97
52.	98
53.	99
54.	100
55.	101
56.	102
57.	103
58.	104
59.	105
60.	106
61.	107
62.	108
63.	109
64.	110
65.	111
66.	112
67.	113
68.	114
69.	115
70.	116
71.	117
72.	118
73.	119
74.	120
75.	121
76.	122
77.	123
78.	124
79.	125
80.	126
81.	127
82.	128
83.	129
84.	130
85.	131
86.	132
87.	133
88.	134
89.	135
90.	136
91.	137
92.	138
93.	139
94.	140
95.	141
96.	142
97.	143
98.	144
99.	145
100.	146

LIST OF TABLES

Table	Page
1. XPath Navigation Operators	7
2. Results of Translating Sentences with and without VPCs	35
3. Findings of VPC Comparison	46
4. Newly Identified Verb-Particle Constructions	48
5. Example Raw String Lyric before Processing	14
6. Pipelines.py File Used by the Royals Program	14
7. Settings.py File Used by the Royals Program	15
8. Format of Artist URL on <i>Metrolyrics</i>	17
9. Example Artist URL	17
10. Search Results for <i>All About That Bass</i>	18
11. <i>Metrolyrics</i> URL for a Listing of All Artists Beginning with L	19
12. Example Entry of the Dictionary <code>letterurl</code>	21
13. URL Leading to Artists Beginning with Non-Alphabetic Characters	21
14. <code>(key, value) letterurl</code> Data Structure	21
15. Format of Artist URLs on <i>Metrolyrics</i>	22
16. <code>(key, value) artistlinks</code> Data Structure	23

Figure	Page
17. Format of Pages beyond First for Alphabetical Artist List on <i>Metrolyrics</i>	24
18. URL to the Third Page of the Artists Beginning with <i>a</i> on <i>Metrolyrics</i>	25
19. Python Code Storing All the Artist Links for Artists Beginning with firstletter	Page

LIST OF FIGURES

1. XML Example	7
2. <i>(key, value)</i> Structure of Songs Dictionary	10
3. Portion of Output File Produced by the <i>SongExtractor</i> Program	11
4. <i>Items.py</i> Used by the Royals Program	13
5. Example Raw String Lyric before Processing	14
6. <i>Pipelines.py</i> File Used by the Royals Program	14
7. <i>Settings.py</i> File Used by the Royals Program	15
8. Format of Artist URL on <i>Metrolyrics</i>	17
9. Example Artist URL	17
10. Search Results for <i>All About That Bass</i>	18
11. <i>Metrolyrics</i> URL for a Listing of All Artists Beginning with L	19
12. Example Entry of the Dictionary <i>letterurl</i>	21
13. URL Leading to Artists Beginning with Non-Alphabetic Characters	21
14. <i>(key, value)</i> <i>letterurl</i> Data Structure	21
15. Format of Artist URLs on <i>Metrolyrics</i>	22
16. <i>(key, value)</i> <i>artistlinks</i> Data Structure	23

Figure	Page
17. Format of Pages beyond First for Alphabetical Artist List on <i>Metrolyrics</i>	24
18. URL to the Third Page of the Artists Beginning with m on <i>Metrolyrics</i>	25
19. Python Code Storing All the Artist Links for Artists Beginning with firstletter	26
20. Python Code Preventing Prefix Problem	28
21. artistlinksongnames Data Structure	29
22. Full Code for the Royals Spider	31
23. Example Tokenized Sentence “Keep me up till the sun is high”	40
24. Python Code Tokenizing the Corpus	41
25. Training the Tagger	43
26. Tagging the Corpus	43
27. A Tagged Sentence	44
28. Code Using Tagger t3 to Tag <i>Brown Editorial Corpus</i>	45

INTRODUCTION

A *corpus* is a collection of text for use in linguistic analysis. The word *corpus* comes from the Latin *corpus*, meaning *body* [1]. The plural of *corpus* is *corpora*. Many corpora are available for public use, such as the *Brown Corpus* [2] and the *British National Corpus* (BNC) [3].

We approached two problems in this research:

1. The lack of an available corpus of song lyrics with which to do linguistic analysis.
2. Determining whether song lyrics would contain a higher percentage of *verb-particle constructions* (VPCs)—a specific type of verb defined in section 3.1—than a corpus of formal speech due to their colloquial nature.

A corpus of song lyrics was necessary to do the analysis of verb-particle constructions, but no such corpus was available. Therefore, we created a corpus generator consisting of two programs, written in Python 2.7.8. We then used the corpus generator to make the *Song Lyric Corpus*, consisting of lyrics from modern popular US songs. We processed The *Song Lyric Corpus* and counted the verb-particle constructions. We also performed the same processing and verb-particle construction counting on the Editorials section of the *Brown Corpus*.

Chapter II details the motivation and design of two programs to generate a corpus of song lyrics. The **SongExtractor** program collects artist and song names from the *Billboard* Website [4], and the **Royals** program uses *web scraping* software—a technique for extracting information from Websites—to find the lyrics for those songs on *Metrolyrics* [5].

In Chapter III, verb-particle constructions are defined, and the motivation for their study in song lyrics is presented. The details of corpus processing and manual evaluation are explained, and the findings are presented. Possible future work is presented.

Chapter IV, the conclusion, summarizes the work and findings of Chapters II and III.

Literature Review

Werner [6] conducted a study of modern pop lyrics but was forced to create a corpus. The *BLUR Corpus* (short for Blues Lyrics collected at the University of Regensburg) was compiled to study African American Vernacular English (AAVE) in

Chapter II

SONG LYRIC CORPUS GENERATOR

MOTIVATION

Colloquial, or informal, speech, especially including modern slang, can pose difficulties for natural language processing because it has not been researched as thoroughly as formal speech. Many of the available corpora are more formal, as in the *Wall Street Journal Corpus*. One potential source of modern informal speech is popular song lyrics. There are no readily available corpora of song lyrics, despite numerous research projects expressing the need for such a corpus. The following literature review describes several studies which have been performed using song lyric corpora their respective authors created, but none are provided to the public. These authors noted the lack of lyric corpora, but these studies failed to provide the corpora they compiled in the course of their research. This may be due to potential copyright issues arising from the distribution of song lyrics.

Literature Review

Werner [6] conducted a study of modern pop lyrics but was forced to create a corpus. The *BLUR Corpus* (short for Blues Lyrics collected at the University of Regensburg) was compiled to study African American Vernacular English (AAVE) in

early blues lyrics [7], but the *BLUR Corpus* is not accessible online. Kreyer and Mukherjee introduce the *Giessen-Bonn Corpus of Popular Music* (GBoP) [8], which included twenty-seven of the top thirty US albums from 2003, but no access is provided to the corpus. Falk analyzes rock lyrics using findings from “a pilot investigation of the pilot version of GBoP” [9], but the author created the corpus themselves by choosing the songs and collecting the lyrics from various lyric Websites. The authors of *American Song Lyrics: A Corpus-Based Research Project* also created their own corpus to collect rock, pop, country, and hip-hop lyrics [10].

Reasons for Lack of Song Lyric Corpora

Much of the time these authors, and others, have spent on creating these corpora could instead have been spent on further analysis. The question remains: why are there no available song lyric corpora?

One reason for this became apparent when we contacted the University of Regensburg requesting access to their corpus of blues lyrics. The University responded that, due to copyright issues, the corpus cannot be made easily available, but researchers may access the corpus if they visit the University. It is kept on a computer which has no Internet access and a modified BIOS which does not allow copying.

Another problem faced while considering song lyric corpus construction for the study of modern language, especially colloquialisms, is that colloquial language evolves quickly. Even a corpus that is one year old is out-of-date when studying this

type of language. Perhaps, even if a corpus were available, it would be out of date, or in a genre the researchers were not interested in. One question faced by those in this research is: How can I find an up-to-date corpus?

There is demand in natural language processing for song lyric corpora, but fear of copyright issues is preventing them from being shared, and the constant publication of new songs cause lyrics to go out of date quickly. A different approach, then, must be taken to fulfill this demand. We implemented a solution to these issues using programs to automatically generate a corpus of current *Billboard Hot 100* song lyrics. These procedures and programs can be publicly shared without violating song copyrights, and each researcher can generate a new corpus on demand at any time, allowing a corpus to contain songs as recent as the current week's top 100 list.

Billboard. www.billboard.com is *Billboard Magazine's* Website [11]. We chose *Billboard* to select the songs for inclusion into the corpus because it is a commonly used source for top US music. *Billboard* ranks songs weekly in the *Hot 100* chart, which lists the top 100 US songs for that week.

Billboard describes how they rank their music [11]:

The week's most popular current songs across all genres, ranked by radio airplay audience impressions as measured by Nielsen BDS, sales data as compiled by Nielsen SoundScan and streaming activity data from online music sources tracked by Nielsen BDS. Songs are defined as current if they are newly-released titles, or songs receiving widespread airplay and/or sales activity for the first time.

SONG EXTRACTOR

We wrote two programs to facilitate the building of a song lyric corpus. The first, the **SongExtractor** program, gathers song titles and artist names for which lyrics will be collected. The second program takes the information gathered from the **SongExtractor** program and scrapes the lyrics for the specified songs. Both programs are written in Python Version 2.7.8 [12].¹

XPath. XPath, a query language used to find information in HTML or XML [13], is used both in the **SongExtractor** program and the **Royals spider**, a program that performs web scraping. XPath expressions select nodes or node sets in the document hierarchy. XML/HTML documents are considered trees of nodes. XPath syntax denotes actions analogous to navigating a file system. Paths to nodes can be specified as a relative path (a path starting from the current node) or an absolute path (starting from the root node). A simple XML example is provided in Figure 1.

XPath Navigation Operators [13]

Expression	Description
/	Selects from the root node
..	Selects the parent node
@	Selects attributes

¹ The most recent version of Python is version 3.4.2, but some software does not support Python 3, including the web crawling software used for this research.

```

The ax:
<collection>
  <movie>
    /collection/movie selects both
    <title lang="en">Jurassic Park</title>
    <year>1993</year>
  </movie>
  <movie>
    Attributes are selected
    <title lang="en">Star Wars</title>
    <year>1977</year>
  </movie>
</collection>

```

Figure 1

XML Example

SongExtractor Design

In Figure 1, the root node is *collection*, and it contains two *movie* nodes. Each *movie* node contains *title* and *year* nodes. Table 1 lists some of the useful path operators in XPath syntax. For example, the node name can be specified directly to select all nodes with that name. The XPath expression **movie** selects both occurrences of the node *movie*.

Table 1

XPath Navigation Operators [13]

Expression	Description
Nodename	Selects all nodes with the name "nodename"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
@	Selects attributes

The expression `/collection` selects the root node *collection*, and `/collection/movie` selects both *movie* nodes. `//year` selects both *year* nodes, as would `year` and `/collection/movie/year`.

Attributes are selected using `@`. `//title[@lang]` selects *title* nodes with the *lang* attribute, which in the case of this example would be both *title* nodes. A much more detailed description of XPath syntax can be found at the XPath tutorial on W3schools [13].

SongExtractor Design

Originally the **SongExtractor** program was written using Scrapy [14], the same software used for the **Royals** program, but it had redirect issues with *Billboard* [4], the Website containing the lists of artists and songs. The *Billboard* Website was designed to display the top 100 songs in pages of ten each. However, when the Scrapy software attempted to navigate to the next page, *Billboard* redirected to the first page.

This redirection problem was solved by writing a program in Python to directly parse *Billboard* Website's HTML source. This program was not subject to the same automatic redirects as the spider originally written with Scrapy. The **SongExtractor** program uses the Python **requests** module to download each page, uses the *lxml* module to extract the HTML, and uses XPath expressions to navigate the hierarchy.

Because the goal of the corpus is to cover modern popular US songs, the *Billboard Hot 100* charts were chosen to select the songs for inclusion. One of the main reasons for generating a corpus of modern song lyrics was to analyze current

linguistic trends. For this reason, the **SongExtractor** program starts by selecting the songs from the current week, then works backwards through successively older weekly listings, up to a specified number of weeks. Thus the selection of songs, and thereby the corpus generated by the **Royals** program, is current within a week.

A future enhancement of the **SongExtractor** program could allow the user to specify the range of weeks from which to select songs. This would allow the building of a corpus containing the popular songs from a specific timeframe. For example, popular songs from the early 1990s could be collected by specifying the weeks in the years 1990-1994.

The **SongExtractor** program has a global constant, **NUM_WEEKS**, which can be changed to vary the size of the corpus. Each week has 100 top songs, but many songs are duplicates from week to week, so the total number of songs in the resulting corpus is far less than 100 times the number of weeks selected. Each song is included only once in the corpus regardless of how many occurrences appeared in the selected weeks.

First, the **SongExtractor** program uses the **requests** module to load the current *Billboard Hot 100* chart. This contains the top 100 songs for the current week. The program then uses XPath expressions to parse the HTML for each song and artist, which are processed using regular expressions to normalize the format of the output, performing actions such as stripping trailing and leading whitespace, and making every character lowercase. Lowercase is required to match the all-lowercase URLs on *Metrolyrics*. These processed strings are stored in a Python *dictionary*—an associative

array with key-value pairs—called songs, in which the song is the key and the artist is the value. For example, `songs["royals"] = "lorde"`. In this case, the song must be the key, because one artist can have multiple songs in the top 100, and each key must be unique. This continues until `NUM_WEEKS` pages have been visited. The structure of songs can be seen in Figure 2.

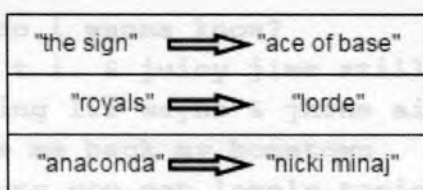


Figure 2

(key, value) Structure of Songs Dictionary

This design has a flaw in that it does not accommodate multiple songs with the same name, such as *Say Something*, which is a song by Christina Aguilera, and also a song by Drake [5]. The list of future enhancements to the **SongExtractor** program includes changing this data structure to a list of tuples, or immutable ordered pairs.²

This would not only allow for multiple songs of the same name, but also preserve ordering. Ordering of songs in the corpus has no meaning, but if a user modified the code of the **SongExtractor** program for their research, they could preserve ordering, creating a stable transformation.

² In Python, a tuple is immutable, or unchangeable. Once a tuple has been assigned, the values cannot be changed.

Once all the artist and song names have been collected, they are written into the output file in the following format: song|artist. The | character was chosen as a separator because it is unlikely to be used in an artist or song name, while other special characters, such as \$ and *, are occasionally used. An example of the output file generated is shown in Figure 3.

```
arctic monkeys|do i wanna know?
b.o.b featuring t.i. & juicy j|we still in this b****
big sean featuring lil wayne & jhene aiko|beware
eric church|give me back my hometown
cole swindell|hope you get lonely tonight
```

Figure 3

Portion of Output File Produced by the *SongExtractor* Program

Complexity. This program iterates through each week specified and stores each artist and song name. There are **NUM_WEEKS** pages per execution, 100 songs per page, and constant operations per song. Then the complexity is $NUM_WEEKS * 100$ songs/page. Since 100 is a constant, both the time and space complexities are $O(n)$, where $n = NUM_WEEKS$.

There are a small number of instances in which the artist name was not captured. This is due to an inconsistency in the HTML used on the *Billboard* page. Most artists on the *Billboard Hot 100* chart have a link from the chart to their artist page. In this majority of cases, the artist name text is located within the link HTML tag. In the few cases that the artist is not linked, the HTML link does not exist. In the 100-week corpus, this only happened for 27 of the 815 songs, or about 3.3%. A future

enhancement would be to implement handling of these special cases. This could be accomplished by first investigating if all the exceptions are formatted in the same way, then adding conditional statements to the source code to handle these cases. This inconsistency resulted in that 3% of songs not being included in the final corpus.

ROYALS

Scrapy

The majority of the web scraping work was done with Scrapy [14], an open source web crawling program written in Python 2.7. Scrapy, like other web scrapers, is designed to extract information from Websites. The installation of Scrapy is described on the Scrapy documentation Website [15], and requires the installation of numerous other items, such as pip [16], a tool that manages Python packages, and the lxml XML toolkit, which is “a Pythonic binding for the C libraries libxml2 and libxslt” [17].

A (very thorough and clear) tutorial for the use of Scrapy is available in the Scrapy documentation [18]. To create a new Scrapy project, only one simple command is needed: “scrapy startproject projectname”. This command creates a directory named **projectname** for the project which contains the following:

- **scrapy.cfg**: the project configuration file
- **projectname/**: the project’s Python module
- **projectname/items.py**
- **projectname/pipelines.py**

- **projectname/settings.py**
- **projectname/spiders/**: a directory to store the project's spiders [18]

Items.py. The **items.py** file defines the data structure the project will use to store the data scraped from Websites. It holds fields for each piece of information extracted from the provided URL or XPath expression. The **items.py** file used by the **Royals** program can be seen in Figure 4.

```
import scrapy

class RoyalsItem(scrapy.Item):
    lyrics = scrapy.Field()
    pass
```

Figure 4

Items.py Used by the Royals Program

This items file is extremely simple because no metadata for the song lyrics was desired. There was no need to keep track of the artists or titles for each lyric. If this information was necessary, only a few small adjustments would be necessary. The lines **artist = scrapy.Field()** and **song = scrapy.Field()** would be added, and **lyrics** would become a list of strings.

Pipelines.py. Once an item has been scraped, it is processed through this item pipeline before output. This pipeline can serve many uses by automatically processing the data. For example, all the letters could be changed to uppercase, or numbers could be multiplied by a specific percentage based on their value. The default **pipelines.py**

file created when the project is started simply returns what is passed into it with no modification. Originally, the **Royals** program did not contain a pipeline because the data was desired without modification. However, when each lyric was extracted from *Settings.py*. This file is required and allows customization of the behavior of Scrapy [20]. By default, three settings exist: `BOT_NAME`, `SPIDER_MODULES`, and `NEWSPIDER_MODULE`. See Figure 7 for the full content of the *Royals* `settings.py` file.

```
That kind of lux just ain't for us\n
```

Figure 5

```
SPIDER_MODULES = ['Royals.spiders']
NEWSPIDER_MODULE = 'Royals.spiders'
```

```
ITEM_PIPELINES = ('Royals.pipelines.PipelinePipeline',)
2001
```

```
CONCURRENT_REQUESTS = 1
ENABLED = False
```

See Figure 6.

```
import re
class RoyalsPipeline(object):
    def process_item(self, item, spider):
        p = re.compile('\n')
        item['lyrics'] = p.sub('', item['lyrics'])
        return item
```

Figure 6

Pipelines.py File Used by the Royals Program

The function `process_item` is called on each item in each spider in the project. In Figure 6, `p` is a regular expression searching for the newline character `\n`. In each item, this string is substituted with the empty string, and the result is assigned back to the item before it is returned.

If the use of a pipeline is desired, it must be specified in the **settings.py** file, as described below.

Settings.py. This file is required and allows customization of the behavior of Scrapy [20]. By default, three settings exist: **BOT_NAME**, **SPIDER_MODULES**, and **NEWSPIDER_MODULE**. See Figure 7 for the full content of the **Royals settings.py** file.

```

BOT_NAME = 'royals'

SPIDER_MODULES = ['royals.spiders']
NEWSPIDER_MODULE = 'royals.spiders'
ITEM_PIPELINES = {'royals.pipelines.RoyalsPipeline':
200}
CONCURRENT_ITEMS = 1
CONCURRENT_REQUESTS = 1
COOKIES_ENABLED = False

```

Figure 7

Settings.py File Used by the Royals Program

BOT_NAME identifies the spider, and specifies how it is invoked from the command line [20]. **SPIDER_MODULES** lists the locations Scrapy will look for spiders for this project [20]. **NEWSPIDER_MODULE** specifies the location in which the *genspider* command will create new spiders [20]. **ITEM_PIPELINES** specifies which pipelines should be used. Each pipeline file in this list is followed by a number which indicates the order in which to use it in the case of multiple pipeline files. Many pipelines can be used in the same spider. In a multiple pipeline situation, pipelines are used sequentially from lowest to highest number. In Figure 7 this number

is 200. **CONCURRENT_ITEMS** and **CONCURRENT_REQUESTS** specify the maximum number of pipeline items to process in parallel and the maximum number of simultaneous requests performed by Scrapy, respectively [20]. The defaults for **CONCURRENT_ITEMS** and **CONCURRENT_REQUESTS** are higher than one, and when using the default settings, the song lyrics were returned out of order. This study is not concerned with including identifying metadata such as artist name and song title, but the lyrics from each song were needed together and in order, to provide context and to account for the few situations in which sentences crossed lyric boundaries.

Royals Design

The **Royals** program, named after one of the top songs of the summer of 2013, is a Scrapy spider written in Python 2.7 which takes the output from the **SongExtractor** program as input, scrapes *Metrolyrics* for the lyrics of each song, and outputs the lyrics into a text file. The **Royals** program can take any text file as input providing it conforms to the format in Figure 3. This allows a user to manually compile a list of songs and artists which are not necessarily in the *Billboard Hot 100* chart.

The main file of the spider is **royals_spider.py**. This file reads the input artist and song names and uses Scrapy to extract the lyrics. The other files in the program are **settings.py**, **items.py**, and **pipelines.py**, which were discussed above.

Before introducing the logic of `royals_spider.py`, a description of the organization of *Metrolyrics* should be attempted. The URLs leading to the song lyrics are formatted as in Figure 8.

<http://metrolyrics.com/song-name-lyrics-artist-name.html>

Figure 8

Format of Artist URL on *Metrolyrics*

For example, the link to the lyrics for *Royals* by Lorde is shown in Figure 9:

<http://www.metrolyrics.com/royals-lyrics-lorde.html>

Figure 9

Example Artist URL

During development, when a simple approach to finding the link to the song was desired, the program built the URLs directly. This was performed by iterating through each song and concatenating the URL and the artist and song names.

From the beginning, it was obvious this approach would not work for the final project, because artist and song names from *Billboard* do not always directly match the names on *Metrolyrics*. This is especially true for artist names when more than one artist is involved. For example, on *Billboard*, the artist for the song *Black Widow* was listed as *Iggy Azalea Featuring Rita Ora*, while on *Metrolyrics* only *Iggy Azalea* is listed, and the URL is <http://www.metrolyrics.com/black-widow-lyrics-iggy-azalea.html>.

Our first solution attempted involved searching for the artist or song name. To search for either on the Website, there is an internal search feature which generates results for both song and artist. The search engine is a fairly forgiving one, and partial matches are returned, which was exactly what was required for this task. Figure 10 shows the partial results of a search for *All About That Bass*.



Figure 10

Search Results for *All About That Bass* [5]

Ultimately, however, this approach proved infeasible because the search results were not directly coded in HTML. Instead, they were dynamically generated by JavaScript, presumably using the Website's internal APIs.

Our final solution was by no means neat, intuitive, or efficient, but it was accurate. *Metrolyrics* allows browsing for artist pages alphabetically. For example, Lorde's *Metrolyrics* page, which contains links to the pages for the lyrics for her

individual songs, can be found by navigating to the page listing all the artists beginning with the letter **L**. The URLs for these pages all have the same format, which can be seen in Figure 11 for the artists beginning with **L**.

http://www.metrolyrics.com/artists-l.html

Figure 11

Metrolyrics URL for a Listing of All Artists Beginning with L

This is how the individual lyric pages were found. The URL based on the artist last name is built, then that is searched for the URL to the specific artist, and finally that URL is searched for the song URL. This is described in more detail in the next section.

Royals Implementation

Royals_spider.py begins by opening and reading the input file which can be created by the **SongExtractor** program, and storing the data in a dictionary called **inputsongs**. The **inputsongs** data structure is identical to the songs data structure of the **SongExtractor** program, in that it is a dictionary in which the keys are song titles and the values are artist names. See Figure 2 for this structure. If the previously discussed enhancement were to be made to the **SongExtractor** program by changing this data structure to a list of tuples instead, then this data structure should also be modified in the same way. As it is currently, **inputsongs** does not lose any data, because if there were two songs with the same name, the **SongExtractor** program

would have chosen one by default and the data would have been lost at that point. This straightforward loop iterates through each song once, so the time complexity is $O(n)$, where n is the number of songs in the input file.

A song can have multiple artists. In these cases, there is usually a primary artist who is billed first on *Billboard*. In the observed data, *Metrolyrics* tended to list songs such as these using only the main artist. For this reason, before the **Royals** program stores the artists and songs into the **inputsongs** data structure, any part of the artist string after a comma or the word *featuring* was stripped.

Once **inputsongs** contains all the songs and artists from the input file, the **Royals** program builds and stores the URLs to the artist letter pages, such as <http://www.metrolyrics.com/artists-a.html>. Because there are only twenty-six letters in the alphabet, it was deemed more efficient to initially store all possible URLs rather than iterate through each artist, check the first letter of the artist name, then store the corresponding URL. Even though it is possible a few unnecessary links may be stored (even if the collection of artists does not contain any beginning with **x**, the link to the **x** artists will be stored), this is less effort than checking the first letter of every artist. These URLs were stored in a dictionary named **letterurl**, with the first letter as the key and the link as the value. Figure 12 shows an example of the entry for artists starting with the letter **a**.

After the `letterurl['a'] = http://metrolyrics.com/artists-a.html` must be searched to locate the pages of each in Figure 12. Artist page URLs take the form displayed in Figure 15. Example Entry of the Dictionary `letterurl`

There is an additional, twenty-seventh entry in the `letterurl` dictionary. *Metrolyrics* assigns any artist beginning with a non-alphabetic character, such as a number, using a **1**³, as in Figure 13.

For example, `http://www.metrolyrics.com/artists-1.html` is seen in Figure 13.

URL Leading to Artists Beginning with Non-Alphabetic Characters

This URL was also added to `letterurl`. The time complexity of this portion of code, because it was done twenty-seven times, is constant. Figure 14 shows the structure of `letterurl`.

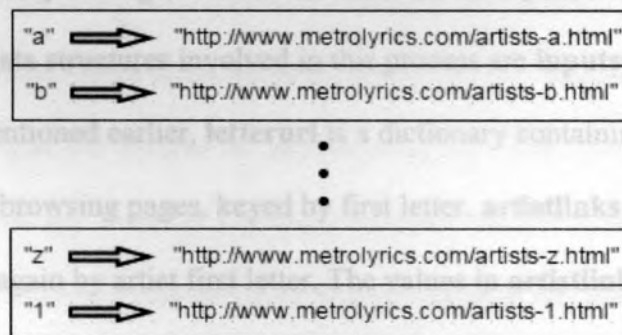


Figure 14

(key, value) `letterurl` Data Structure

³ This is the number one, not to be confused with the letter L.

After the twenty-seven URLs are built and stored, those pages must be searched to locate the pages of each individual artist. Artist page URLs take the form displayed in Figure 15.

<http://www.metrolyrics.com/artist-lyrics.html>

Figure 15

Format of Artist URLs on *Metrolyrics*

For example, to find the URL for the band Ace of Base, <http://www.metrolyrics.com/artists-a.html> is searched until the URL <http://www.metrolyrics.com/ace-of-base-lyrics.html> is found. The process of locating these artist pages is described below.

Data Structures

The links to artist pages are found by comparing the artist names from the input file, stored in **inputsongs**, to the links found on the alphabetical artist browsing pages. The main data structures involved in this process are **inputsongs**, **letterurl**, and **artistlinks**. As mentioned earlier, **letterurl** is a dictionary containing links to the alphabetical artist browsing pages, keyed by first letter. **artistlinks** is another dictionary, keyed again by artist first letter. The values in **artistlinks** are lists containing all links to artist pages beginning with that letter, as in Figure 16.

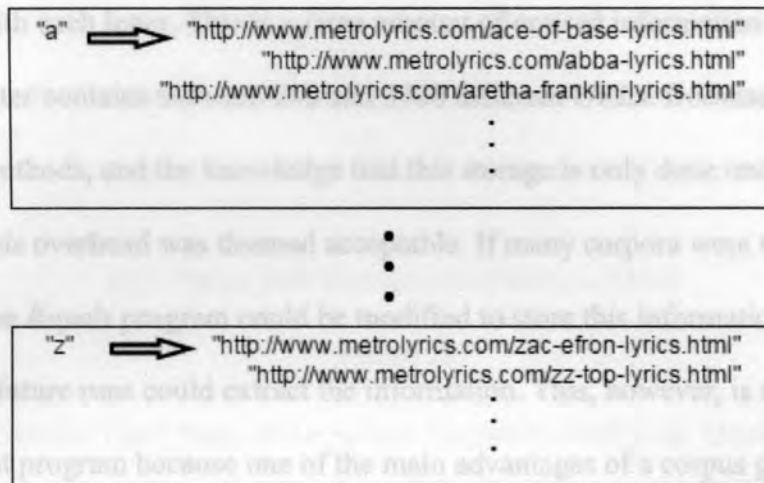


Figure 16

(key, value) artistlinks Data Structure

Unfortunately, the listing of artists on the first letter pages are not alphabetical. For this reason, the strategy of sorting **inputsongs** alphabetically by artist, then proceeding through the letter pages doing a one-by-one comparison will not work. Additionally, because of the number of artists with similar names, and the variation between how multiple artists are represented on the two Websites, finding the first match is not sufficient. Often times, there is no exact match. Instead, the best match is required. For example, the artist of the popular song *Say Something* is listed in *Billboard* as *A Great Big World & Christina Aguilera*. However, searching for that exact artist name in the *Metrolyrics* URLs will not result in any matches.

Because of these two issues with searching one-by-one until an exact match is found between the input artists and the links on the alphabetical pages, we adopted another strategy. The dictionary **artistlinks** is used to store all URLs of artist pages

beginning with each letter. This is a large amount of unused information being stored. Each first letter contains between 200 and 3700 different URLs. Because of the issues with other methods, and the knowledge that this storage is only done once per corpus generated, this overhead was deemed acceptable. If many corpora were to be generated, the *Royals* program could be modified to store this information in a file from which future runs could extract the information. This, however, is not a function of the current program because one of the main advantages of a corpus generator is the ability to have an up-to-date corpus after every run. Storing the artist information in this way could quickly lead to out-of-date information, most notably for new artists that are added to the *Metrolyrics* database, but also in the case of *Metrolyrics* modifying the URLs for any artists.

Locating artist page details. The program iterates through each song in **inputsongs**. First, it checks the first letter of the artist, and if the set of URLs for artists starting with that letter has not yet been stored in **artistlinks**, they are stored at this time. This is done by using the requests module to load the HTML of the page containing links to artists starting with that letter. Only forty artists are shown on this first page. A link labeled next must be used to access the following forty, and so on. The format of these next URLs is shown below in Figure 17.

<http://www.metrolyrics.com/artists-letter-page#.html>

Figure 17

Format of Pages beyond First for Alphabetical Artist List on *Metrolyrics*

Results 1-40 are on the first, non-numbered page. The page numbering starts at one for results 41-80, and continues until there are no more results. The example in Figure 18 shows the page with results 81-120 of artists starting with the letter **m**.

<http://www.metrolyrics.com/artists-m-2.html>

Figure 18

URL to the Third Page of the Artists Beginning with **m** on *Metrolyrics*

On the first page containing the first forty artists is also a number of total artists. This number is extracted and used to calculate the number of pages. Each of these pages is loaded, and XPath expressions are used to select all the individual artist links from each page and store them in the **artistlinks** dictionary. There is one entry per artist, so the space complexity of this information is linear with respect to the number of artists. This code is shown in Figure 19.

The approach taken to match the input artist with the URL is word-by-word, breadth-first. The *Rayala* program starts by attempting to match the first word of the artist with the first word in the artist URL, and if multiple matches are found, then attempts to match the first two words, and so on. A hyphen separator was used

```

# Stores links for all artists on the first page in artistlinks
artistlinks[firstletter] = tree.xpath('//tr/td/a/@href')

# Iterates through each page after the first page
# and appends the artist links to artistlinks
for page in range(1, numpages):
    # Builds the urls for the following pages
    # Ex: http://www.metrolyrics.com/artists-j-3.html
    pageurl = urlbeginning + firstletter + "-" + str(page) + urlending

    # Load the page
    letterpage = requests.get(pageurl)
    tree = html.fromstring(letterpage.text)

    # Append the links from the page to the array of links
    # already stored for that letter in artistlinks
    artistlinks[firstletter] = artistlinks[firstletter] +
tree.xpath('//tr/td/a/@href')

```

Figure 19

Python Code Storing All the Artist Links for Artists Beginning with firstletter

Once all the links for that letter are stored, they must be searched for the artist match links containing *metrolyrics*. This was made possible by using *metrolyrics*. To currently in question. As a reminder, the process of searching all the artists of a certain first letter for a match is done iteratively over each input song. To start this process, the artist name is split on whitespace and stored as an array. Then certain punctuation not used by *Metrolyrics* is stripped, leaving just the artist name as an array, one word per element, all lower case. With the artist name in this format, the matching can begin.

The approach taken to match the input artist with the URL is word-by-word, breadth-first. The **Royals** program starts by attempting to match the first word of the artist with the first word in the artist URL, and if multiple matches are found, then attempts to match the first two words, and so on. A hyphen separator was used

between words as this is also how the links are built. A **numwords** variable keeps track of the number of words currently being matched. This number starts at one, unless the first word is *the*, in which case it starts at two. This was done to avoid unnecessary matching of any artist starting with *the*. Python does not have **do...while** logic, so this matching is placed in a **while True** block, and once one and only one match is found, a **break** statement ends the loop.

One of the challenges encountered during the creation of this spider was that occasionally multiple artist links would match even when every word from the input artist was being used. We found two reasons for this. Originally, the program had been only matching *firstword-secondword*. This led to issues with artists that were prefixes of other artists. One example is Jessie J. Originally, this program was attempting to match links containing *jessie-j*. This was matching both Jessie J and Jessie James. To solve this problem, a hyphen was appended to the end of the string to match, because the links always end in *-lyrics.html*. When the string *jessie-j-* was searched for, only Jessie J matched.

Another version of the prefix problem was encountered in the event that some artists are whole word prefixes of other artists. In this case, the first few words were exactly the same. This happened with Taylor Swift and Taylor Swift Cover Band. The solution employed by the **Royals** program is if every word in the input artist is being searched for, and there are multiple matches, then use the shortest match. See Figure 20.

```

# If we have used all the words in the artist name, but
# more than one link matched
# This is used when one artist name is a subset of another
# Ex: taylor swift vs. taylor swift cover band
if (numwords == len(artistname)) and (len(matchedlinks) > 1):
    # Find and use the shortest.
    shortest = matchedlinks[0]
    for matchedlink in matchedlinks:
        if len(matchedlink) < len(shortest):
            shortest = matchedlink

matchedlinks = []
matchedlinks.append(shortest)

```

Figure 20

Python Code Preventing Prefix Problem

The time complexity of the matching of artist names to links on *Metrolyrics* is different in the best and worst case scenarios. In the best case, the artist link is found after exactly one full pass through all artists starting with that letter. This is done for each song, so the best case complexity is $O(nm)$, where n is the number of songs and m is the number of artists with the same first letter. Even if the artist is not found on the first few passes, the subsequent iterations only compare to the potential matches found in the previous iteration. Therefore any further iterations will be significantly reduced in amount of comparisons; enough in practice to become negligible and thus declare the complexity $O(nm)$ in all cases.

There are n songs stored, and m links, so the space complexity of this portion of the **Royals** program is $O(n + m)$.

Once exactly one matching link is found for the artist, the song name and artist URL are saved in a list of tuples called **artistlinksongnames** which will be accessed

in the following section of code. The structure for **artistlinksongnames** can be seen below in Figure 21. The links themselves were not enough; each link also had to be associated with the song names for two reasons. First, as previously stated, multiple artists may have songs with the same name. Second, if the song name were not stored, later each artist's page would have to be searched for a song name matching a song from the input, which would require significantly more work than is necessary.

"the sign", "http://www.metrolyrics.com/ace-of-base-lyrics.html"
"royals", "http://www.metrolyrics.com/lorde-lyrics.html"
"anaconda", "http://www.metrolyrics.com/nicki-minaj-lyrics.html"

•
•
•

Figure 21

artistlinksongnames Data Structure

After each artist's link is found, then these pages must be searched for the artists' links to the lyrics of each individual song. In the previous section of code, a list of artist names was searched through, attempting to match artists from the input file. Now, a list of song names from a particular artist is searched to match the song name from the input file.

Each song in **artistlinksongnames** is iterated through. First, the artist's page is loaded. Then, similarly to the search for artist names, the song is split on whitespace and stored as a list. Punctuation is stripped out of the search string, because the

Metrolyrics URLs do not contain punctuation other than the hyphens separating words.

The main difference between this search and the search for artist is that this search is first performed for all words in the song title, and the number of words in the search string is iteratively decreased by one until a matching song is found. These decisions were made in an attempt to decrease runtime. During the artist search, as many as 3700 links may be compared to the search string. Therefore it is more efficient to narrow down the search quickly, rather than run the risk of searching all artists many times before getting any matches. When examining songs by one artist, there are typically far fewer songs, and there is less variation in the representation of song names once stripped of punctuation, so the likelihood of an exact match on the first iteration is high. The time complexity of this section is $O(nq)$, where n is the number of songs (total, from the input file), and q is the average number of songs on each artist's page. Because even the Beatles only had between two and three hundred songs, as the number of songs in the corpus gets large, q will become negligible and the time complexity reduces to $O(n)$.

The space complexity is smaller than the time complexity, because the program is storing n songs and q song links. Therefore, the space complexity is $O(n + q)$, which, like the time complexity above, will reduce to $O(n)$.

When a match between the song provided by the input file and the URLs for the song lyrics is found, the URL is appended to the list `urls`. After all the songs have been searched for, `urls` contains the final list of URLs from which to scrape song

lyrics. These URLs become the `start_urls` for the class `LyricSpider`, which is the portion of the **Royals** program that actually scrapes *Metrolyrics* for the requested song lyrics.

`LyricSpider` is very simple, containing only three pieces of data; the name, domains allowed, and `start_urls`. Its parse function uses XPath expressions to select every line from the lyrics, and returns each line. Figure 22 shows the full code of `LyricSpider`. The lyrics are then written in an output file. This file becomes the corpus after some manual processing.

```
class LyricSpider(scrapy.Spider):
    name = "royals"
    allowed_domains = ["metrolyrics.com"]
    start_urls = urls

    def parse(self, response):
        for sel in response.xpath('//p[@class="verse"]/text()').extract():
            item = RoyalsItem()
            item['lyrics'] = sel
            yield item
```

Figure 22

Full Code for the Royals Spider

Complexity. Most of the work is done before the actual spider scrapes the information. The time complexity of the spider itself is $O(nl)$, where n is the number of URLs, or songs, and l is the number of lines in each song. Of the preprocessing tasks, the one with the highest time complexity was when the program iterated through each song and compared each song with each link in the page of artists with that

particular first letter. This, again, was complexity $O(nm)$, where m is the number of artist links starting with each letter.

THE SONG LYRIC CORPUS

The *Song Lyric Corpus* contains songs from 100 weeks of the *Billboard Hot 100* Chart [11]. This corpus was generated on November 11, 2014, so it contains most of the top 100 songs from the weeks of 12/15/2012 to 11/15/2014. Once the metadata was manually stripped from the corpus, there remained a total of 40440 lines and 280591 words.

Future Work

The corpus generation could be improved by implementing the handling of special cases, as previously described, such as if two artists have a song with the same name. A graphical user interface would greatly enhance the usability of both the **SongExtractor** and **Royals** programs. The addition of options to specify a specific time period would allow for studies of pop songs of specific eras.

Adding genre-specifying options would allow for the corpus to become much more specialized. Of course, if a user wanted only specific songs in their corpus, they can manually compile the input file for the **Royals** program, and skip using the **SongExtractor** program. This allows for as much customization as *Metrolyrics* supports. But adding an option to specify genre would allow, for example, a hip-hop genre corpus to be generated. This would require more research, however, because the

Billboard Hot 100 chart does not keep genre information, so the identification of songs by genre would have to be obtained elsewhere.

Chapter III

COMPARING VERB-PARTICLE CONSTRUCTIONS IN CORPORA OF VARYING FORMALITIES

MOTIVATION

Verb-Particle Construction Definition

Multi-word expressions (MWEs) pose a serious problem for natural language processing tasks such as machine translation and semantic analysis. MWEs are "idiosyncratic interpretations that cross word boundaries (or spaces)" [21]. They can typically be thought of as one unit of meaning. Examples of multiword expressions include *in short*, *kick the bucket*, *San Francisco*, *attorney general*, and *look up* [21]. Sag, Baldwin, Bond, Copestake, and Flickinger highlighted the importance of researching methods to computationally handle MWEs [21].

Verb-Particle Constructions (VPCs) are a specific type of MWE consisting of a verb and one or more *particles*. A particle is a word which has no meaning by itself, but must be paired with one or more other words to impart meaning. VPCs, as with other MWEs, have meanings that are more than the sum of their parts. For example, one can *look up* a building, and one can *look up* a word [21]. The former is a case of a verb followed by a prepositional phrase. The latter, however, means typically to consult a reference such as a dictionary or index to find a particular word

Chapter III

Machine Translation Example

COMPARING VERB-PARTICLE CONSTRUCTIONS IN CORPORA OF VARYING FORMALITIES

MOTIVATION

Verb-Particle Construction Definition

Multi-word expressions (MWEs) pose a serious problem for natural language processing tasks such as machine translation and semantic analysis. MWEs are “idiosyncratic interpretations that cross word boundaries (or spaces)” [21]. They can typically be thought of as one unit of meaning. Examples of multiword expressions include *in short*, *kick the bucket*, *San Francisco*, *attorney general*, and *look up* [21].

Sag, Baldwin, Bond, Copestake, and Flickinger highlighted the importance of researching methods to computationally handle MWEs [21].

Verb-Particle Constructions (VPCs) are a specific type of MWE consisting of a verb and one or more *particles*. A particle is a word which has no meaning by itself, but must be paired with one or more other words to impart meaning. VPCs, as with other MWEs, have meanings that are more than the sum of their parts. For example, one can *look up* a building, and one can *look up* a word [21]. The former is a case of a verb followed by a prepositional phrase. The latter, however, means typically to consult a reference such as a dictionary or index to find a particular word

VPCs are often, but not always, characterized by the ability to be split: one can *look a word up*, but not **look a tower up*⁴ [21].

Machine Translation Example

To illustrate one of the issues that VPCs specifically cause, namely machine translation, three sentences with the same meaning were translated from English to Spanish using Google Translate [22]. The first sentence contained the more formal syntactic phrase forms a sentence tree accounting for each word representing the word *submitted*. The second and third contained the VPC with the same meaning: *turned in*. The second sentence keeps the words in the VPC together, while the third splits the words. The Spanish output was then translated back into English and compared with the original English output. Table 2 compares the original to the resulting sentences. The sentences containing the VPCs do not translate as well, especially the sentence with the split verb/particle.

Table 2

Results of Translating Sentences with and without VPCs

Original Input	Output after English/Spanish/English Translation
She submitted a paper.	She presented a paper.
She turned in a paper.	She had a role.
She turned a paper in.	It became an article in.

⁴ In linguistic analysis, an asterisk represents an ungrammatical phrase.

Literature Review

In 2002, Baldwin and Villavicencio proposed a method of VPC extraction using three different approaches: a *part-of-speech tagger*, a *chunk parser*, and a *syntactic parser* [23]. A part-of-speech tagger assigns the part-of speech classification to each word in its input, such as noun or verb. A chunk parser separates the text into non-overlapping regions, usually based on noun, verb, or prepositional phrases, and a syntactic parser forms a sentence tree accounting for each word representing the syntax hierarchy. In 2005, Baldwin used the output of the previous paper and combined it with valence information to extract VPCs [24]. In 2007, Kim and Baldwin estimated VPC compositionality using distributional and semantic similarity of the head verb [25]. In 2006 and 2010, the same authors use a parser and word senses of the head verbs and nouns using WordNet to distinguish between VPCs and verbs followed by prepositions [26], [27]. A classifier was then developed using a memory based learner, TiMBL [28].

In 2014, Smith uses alignment in a bilingual corpus of English and Spanish movie and television subtitles [29], proposing that because VPCs tend to be less formal than their non-VPC counterparts, a spoken text corpus will have a higher density of VPCs than a more formal corpus. He also expects that these VPCs will often be slang or profanities, and not found in other corpora.

Goals of Verb-Particle Construction Analysis

This study of VPCs focuses on the examination and analysis of the number of VPCs in an informal corpus, with the hypothesis that there is a positive correlation between the amount of VPCs and the informality of the corpus. A corpus with a large number of VPCs may be useful in future attempts of identification.

The analysis of the verb-particle constructions of a song lyric corpus undertaken in this study has three goals:

1. The identification of verb-particle constructions in the corpus.
2. A comparison of number of occurrences of VPCs between the *Song Lyric Corpus* and a corpus of more formal writing. The *Brown Editorials Corpus* is used.
3. The identification of new or interesting VPCs.

VPC IDENTIFICATION

NLTK

The Natural Language Processing Toolkit (NLTK) [30] is an open source library written in Python designed to perform natural language processing tasks such as tokenization, part-of-speech tagging, and parsing. The NLTK was created in 2001 at the University of Pennsylvania and provides interfaces to over 50 lexical resources and corpora, including the *Brown Corpus* and *Penn Treebank*. Bird, Klein, and Loper note in their book *Natural Language Processing with Python* [30] that the NLTK is

simple, consistent, extensible, and modular, but “while the toolkit is efficient enough to support meaningful tasks, it is not highly optimized for runtime performance”.

Installing NLTK is simple. NLTK requires Python versions 2.6-2.7 or 3.2+.

Typically users install both NLTK and NLTK Data, which contains various corpora

and grammars [31]. This research also required NumPy [32], a Python package designed for scientific computing. This is used in NLTK’s part-of-speech taggers.

Method Overview

We took the following steps to identify and analyze verb-particle constructions in the *Song Lyric Corpus*:

1. Manual processing was performed on the *Song Lyric Corpus* to enable use of Python’s Natural Language Processing Toolkit.
2. The NLTK was used to tokenize the corpus.
3. A part-of-speech tagger was trained and used to tag the parts of speech of the corpus.
4. The same tagger was applied to the editorials sections of the *Brown Corpus*.
5. Manual examination of ten percent of the song corpus confirmed the POS tagging of VPCs and ensured all VPCs were found.
6. The same manual examination was performed on 50% of the tagged Editorials section of the *Brown Corpus*.

7. VPC frequency is compared, along with number of correct/incorrect tags.

8. A small number of new VPCs were identified.

Manual Corpus Processing

The output from the **Royals** program required processing before it was usable for analysis. Each line from the lyrics was wrapped in quotes with the keyword *lyrics*. This had to be stripped. Additionally, some metadata in the lyrics provided by *Metrolyrics* is conveyed in square brackets. This data includes text such as *Verse 3*, or *Bridge*, or the name of the singer of the following lyrics in multi-artist songs. We deleted this excess data. Repetitions of lines are sometimes conveyed by printing the line once, then appending (*x#*), where *#* is the number of desired repetitions. We also stripped these. The lines were not expanded, since multiple redundant lines will not aid in linguistic analysis.

The song lyrics often do not contain periods. Sentences boundaries are either not marked, or marked by newlines. Song lyrics also tend to contain run-on sentences. A random sampling of songs was examined to determine how to designate sentence boundaries. The majority of the lyrics are organized with one sentence per line, so we appended a period to the end of each line. This undoubtedly introduced some extra periods, but the analysis involves manually examining the part-of-speech tags just to account for errors such as this.

Figure 23

Example Tokenized Sentence "Keep me up till the sun is high."

Automated Corpus Modifications:

Tokenization and Tagging

Tokenization. *Tokenization* is “the task of cutting a string into identifiable linguistic units that constitute a piece of language data” [30]. This must be done before any automated linguistic analysis can occur. NLTK provides a default sentence tokenizer `sent_tokenize()` which separates the string into sentences using an algorithm which takes into account typical sentence punctuation such as '.' or '?' and also accounts for items such as abbreviations [31]. At first, this tokenizer did not work properly. The corpus was constructed so that each line was one sentence. Therefore it was missing the usual space after a sentence boundary. We manually appended spaces at the end of each line, and successfully re-applied the tokenizer.

Already broken into sentences, the corpus then had to be further tokenized into individual tokens. A token is not necessarily a word; it is a piece of language data. Most of the tokens are, in fact, words. There are exceptions, however. The word *don't* becomes two tokens to account for their separate meanings: *do* and *n't*. We used NLTK's default word tokenizer, `word_tokenize()`, on the sentences in the sentence-tokenized corpus. This produced a fully tokenized corpus. See Figure 23 for an example of a tokenized sentence from the song *Young Girls* [33].

```
['Keep', 'me', 'up', 'till', 'the', 'sun', 'is', 'high', '.']
```

Figure 23

Example Tokenized Sentence “Keep me up till the sun is high.”

This example is representative of the problems posed by this analysis. *Keep up* is a verb-particle construction, which the analysis is attempting to identify. Also, the word *till* is misspelled. It should be the abbreviation of *until*: *'til*. This will illustrate how the analysis works with imperfect data. In the following section, this sentence will be shown with its part-of-speech tags.

The full code for the tokenization of the corpus is shown below in Figure 24.

```
# Opens the corpus
file = open('11-11-8-30pm\\royalsitemsstripped.txt', 'r')

# Reads the file
# Adds a space at the end of each line so the sentence
tokenizer
# will work correctly. Also discards a newline because the
corpus
# contains a newline after every sentence, and the sentence
# tokenizer requires the entire file in one string.
data = file.read().replace('\n', ' ')

# Tokenize the sentences
full_sentences = nltk.sent_tokenize(data)

# Tokenize the words
sentences = [nltk.word_tokenize(sent) for sent in
full_sentences]
```

Figure 24

Python Code Tokenizing the Corpus

Tagging. In order to perform any linguistic analysis involving word meanings, the words in the data must be classified into their parts-of-speech, such as noun, verb, or preposition. This process is called *part-of-speech* (POS) *tagging*, or just tagging.

Ideally, a corpus would be tagged by hand, using native speakers of the corpus language. This, however, is time-consuming and unrealistic. Luckily, many automated POS taggers are available. NLTK has a number of taggers available. We trained a trigram tagger `t0` using NLTK's `NgramTagger` class and the *Brown Corpus*.

When assigning a tag to a word, a unigram tagger assigns the tag that is most statistically likely for that token. For example, it may encounter the token *wind*, and its statistics tells it that the noun is more common than the verb, so the word will be assigned the tag of *NN*, the Brown tag for singular noun. Generally these statistics are gathered by training the tagger on a set of data. An *n*-gram tagger works similarly, but uses statistics for the current word and the part-of-speech tags of the preceding $n - 1$ tokens [30]. A bigram (2-gram) tagger may therefore encounter the token *wind*, and see that the previous token was tagged *TO*, which in the Brown tagset stands for the infinitival *to*.⁵ This bigram tagger would then conclude that, when positioned after the infinitival *to*, the current tag is most likely a verb. A trigram tagger, such as `t3`, assigns the most likely tag based on the current word and the previous two words and tags.

When training a tagger, NLTK permits the use of a backoff tagger. This is essentially the backup if the main tagger fails to find statistics for a particular positioning. The trigram tagger used had a chain of backoff taggers. The trigram tagger's backoff was the bigram tagger, the bigram tagger's backoff was the unigram tagger, and the unigram tagger's backoff was the default tagger, which simply tags

⁵ In English, the infinitive form of a verb is the basic dictionary form, with no modifications for tense or person. The infinitival *to* refers to the particle *to* that precedes the infinitive form, such as in *to eat*.

everything as a noun. We trained the tagger on the News section of the *Brown Corpus*.

See Figure 25 for the code training the tagger.

```

brown_tagged_sents =
brown.tagged_sents(categories='news')

t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(brown_tagged_sents, backoff=t0)
t2 = nltk.BigramTagger(brown_tagged_sents, backoff=t1)
t3 = nltk.TrigramTagger(brown_tagged_sents, backoff=t2)

```

Figure 25

Training the Tagger

After training, we used the tagger on the fully tokenized corpus. The resulting tagged corpus was output into a file by our program, sentence by sentence, along with the untagged, untokenized sentence for easier reading. The tagging was done in only one line of code. See Figure 26.

```

# Tag the corpus
sentences = [t3.tag(sent) for sent in sentences]

```

Figure 26

Tagging the Corpus

Figure 27 shows one of the lyrics of the final part-of-speech tagged corpus. This is the same sentence as in Figure 23.

```
Keep me up till the sun is high.
[('Keep', 'NN'), ('me', 'u'PPO'), ('up', 'u'RP'), ('till', 'NN'),
 ('the', 'u'AT'), ('sun', 'NN'), ('is', 'u'BEZ'), ('high', 'u'JJ'), ('.', 'u'.')]

```

Figure 27

A Tagged Sentence

The tagged sentence is shown in the form of tuples. Each token has a tuple, in which the token is the first element, and the POS tag is the second. These particular tags are from the Brown tagset. Details of all the tag types can be found in the *Brown Corpus Manual* [2]. The tags important to this analysis are *IN* (prepositions) and *RP* (adverbs and particles). Note in the sentence in Figure 27 has mis-tagged *till* as a *NN* (noun). This illustrates the difficulties of working with inaccurate data.

There are other taggers available, and other corpora on which to train them. Various evaluation methods exist for taggers, but the goal of this tagger is to expedite manual analysis, not to maximize the tagging accuracy, so no evaluation is done.

The Brown Corpus

The NLTK provides the *Brown Corpus* in both tagged and untagged versions. The tagger *t3* was trained on a tagged version of the News section of the *Brown Corpus*. This same tagger was used to tag both the *Song Lyric Corpus* and also the untagged version of the Editorials section of the *Brown Corpus*. This allows a comparison of the tagger's performance on each corpus. Figure 28 shows the code using *t3* to tag the Editorials section of the *Brown Corpus*.

```

brown_sents = brown.sents(categories='editorial')
brown_sents = [t3.tag(sent) for sent in brown_sents]

```

Figure 28

Code Using Tagger t3 to Tag *Brown Editorial Corpus*Manual Examination

After the corpus has been tokenized and tagged, it is ready for manual analysis. We chose a sample size of ten percent to manually examine. This portion of the corpus has 4013 sentences and 28319 words. A similarly sized section of the *Brown Corpus* was desirable for comparison. The sentences in song lyrics tend to be short, especially compared with a corpus of formal editorials. For this reason, a similar word count, instead of sentence count, was desired. We selected fifty percent of the Editorials section of the *Brown Corpus* for a total of 30258 words.

The manual examination of these corpora is based on the methods used by Kim and Baldwin [27]. Two tags are examined: *RP* – ‘adverb, particle’ and *IN* – ‘preposition’. If tagged correctly, the particle of all VPCs should be tagged with *RP* and none should be tagged with *IN*. However, the identification of VPCs is difficult and it is expected the some VPCs will erroneously receive the *IN* tag. Additionally, the *RP* tag also includes adverbs, so it cannot be trusted to identify VPCs in all cases.

First we examined each instance of the *RP* tag, and a determined whether or not the tagged token was actually a VPC. After that, we examined each *IN* tag and made the same determination. All determinations were marked in the text for counting

and reference purposes. This was done for both the *Song Lyric Corpus* and the *Brown Editorials Corpus*.

FINDINGS

Comparison of VPC Occurrences

Table 3 contains the number of sentences, words, tags, and VPCs in each corpus. The POS tagger is tagged at least 10% of *IN* tags incorrectly. More analysis would need to be performed to get a true measure of the tagger's accuracy regarding prepositions, but that is not the focus of this study. The purpose of the POS tagger is was to greatly narrow down the number of tokens to be considered, thereby a mutual evaluation possible within the time frame.

Table 3

Findings of VPC Comparison

	Song Lyric Corpus	Brown Editorial Corpus
Sentence Count	4013	1498
Word Count	28319	30258
Words per Sentence	7.06	20.20
RP Tags	442	131
RP Tags that are not VPCs	105	50
IN Tags	1778	2984
IN Tags that Tag VPCs	223	29
VPCs	560	110
VPCs per Sentence	.14	.07
Probability a Word is in a VPC	.040	.007

Table 3 shows that the sentence count varies greatly due to the shorter sentence length in the *Song Lyric Corpus*. The average lyric only has seven words, while a sentence from the *Brown Editorial Corpus* has twenty. As mentioned earlier, this was why word count, and not sentence count, was chosen to divide the corpora.

IN tags are meant to mark prepositions, so any VPCs in which the particle is marked with *IN* is tagged incorrectly. In the *Song Lyric Corpus*, approximately 12.5% of the *IN* tags were inaccurately attached to the particles of VPCs. This was the case for 9.7% of the *IN* tags for the *Brown Editorials Corpus*. The POS tagger **t3** tagged at least 10% of *IN* tags incorrectly. More analysis would need to be performed to get a true measure of the tagger's accuracy regarding prepositions, but that is not the focus of this study. The purpose of the POS tagger **t3** was to greatly narrow down the number of tokens to be considered, therefore making a manual evaluation possible within the timeframe.

The difference in the number of VPCs in the two corpora is significantly large. The *Song Lyric Corpus* contains over five times the amount of VPCs than the *Brown Editorials Corpus*. In the *Song Lyric Corpus*, a word has approximately a 4% chance to be contained in a verb-particle construction, while in the *Brown Editorial Corpus*, a word has only a .7% chance. This is assuming a VPC consists of two words, but it should be noted that this is not always the case. Some verb-particle constructions contain three words, such as *brush up on* [21].

These results indicate that, due to the high occurrences of VPC, future studies may benefit from using a corpus of song lyrics.

Future Work

Applying an automated method of VPC extraction on a corpus of song lyrics could give valuable insight into potential improvements to automated VPC identification and extraction methods. Most notably, it would be useful to apply the

New VPCs Klee and Baldwin [25], [26] in a corpus of informal speech, using a parser and the Smith [29] defines a new VPC as one which does not appear in either Dictionary.com [34] or The Free Dictionary [35]. We identified the VPCs matching these criteria and they can be seen in Table 4 below.

Table 4

Newly Identified Verb-Particle Constructions

snatch out*	snatch the fucking track out
pour up*	Where my double cup, time to pour it up
howl out	howled out with joy
silverware out	silverware a nigga out
double on down	double on down like it's gonna make you free
wage on	The tug of war wages on
walk on by	They can walk on by
come on over	So come on over baby
bite up	While I bite the beat up

In addition to the two Websites that Smith used, we also searched Urban Dictionary [36], a site that provides definitions of slang, for each new VPC. Only the VPCs marked with * were listed on Urban Dictionary. Table 4 seems to suggest that three-word VPCs are not well documented. While phrases like *bite up* seem new, the phrases *walk on by* and *come on over* are familiar.

Future Work

Applying an automated method of VPC extraction on a corpus of song lyrics could give valuable insight into potential improvements to automated VPC identification and extraction methods. Most notably, it would be useful to apply the

methods of Kim and Baldwin [25], [26] to a corpus of informal speech, using a parser and the semantic meaning of the head verb to train a VPC classifier. Other media of informal speech would be worth investigation as well, such as Smith's idea of identifying VPCs in a subtitles corpus [29].

CONCLUSION

We created two programs to automatically generate song lyric corpora. This corpus generator has the potential to save future researchers hours of manual corpus compilation. Because it is a corpus generator, and not a corpus itself, sharing these programs has no potential of copyright issues, as all software used was open source. Additionally, the generation of a corpus, rather than the re-use, allows for corpora to remain modern. New songs are released every week, and now a corpus can be generated at any time that includes the top songs from the current week.

Using the song lyric corpus generator, we made the *Song Lyric Corpus*, consisting of approximately 800 of the top US songs from the last 100 weeks. We then tokenized and tagged both the *Song Lyric Corpus* and a corpus of more formal speech, the *Brown Editorials Corpus*, and counted the verb-particle constructions. We found that the *Song Lyric Corpus* we generated contained over five times as many VPCs as the existing *Brown Editorials Corpus*, indicating that a corpus of song lyrics is a promising medium with which to research verb-particle constructions.

Chapter IV

CONCLUSION

We created two programs to automatically generate song lyric corpora. This corpus generator has the potential to save future researchers hours of manual corpus compilation. Because it is a corpus generator, and not a corpus itself, sharing these programs has no potential of copyright issues, as all software used was open source. Additionally, the generation of a corpus, rather than the re-use, allows for corpora to remain modern. New songs are released every week, and now a corpus can be generated at any time that includes the top songs from the current week.

Using the song lyric corpus generator, we made the *Song Lyric Corpus*, consisting of approximately 800 of the top US songs from the last 100 weeks. We then tokenized and tagged both the *Song Lyric Corpus* and a corpus of more formal speech, the *Brown Editorials Corpus*, and counted the verb-particle constructions. We found that the *Song Lyric Corpus* we generated contained over five times as many VPCs as the existing *Brown Editorials Corpus*, indicating that a corpus of song lyrics is a promising medium with which to research verb-particle constructions.

REFERENCES

- [1] "Corpus," *Online Etymology Dictionary*, November 21, 2014, <http://dictionary.reference.com/browse/corpus>.
- [2] *Brown Corpus Manual*, November 21, 2014, <http://clo.uni-norwich.ac.uk/brown/bcm.html>.
- [3] *The British National Corpus*, version 3 (BNC XML Edition), 2007. Distributed by Oxford University Computing Services on behalf of the BNC Consortium, <http://www.natcorp.ox.ac.uk/>.
- [4] "Billboard-music charts," *Billboard* Photo Gallery & Free Video, November 21, 2014, <http://www.billboard.com/>.
- [5] Song Lyrics | MetroLyrics, November 21, 2014, <http://www.metrolyrics.com/>.
- [6] V. Werner, "Love is all around: A corpus-based study of pop lyrics," *Corpora*, vol. 7, no. 1, pp. 19-50, 2012.
- [7] G. Herk and U. Mieschater, "I can look through muddy water: Analyzing earlier African American English in blues lyrics (BLUR)," *English World-Wide*, pp. 205-209, 2005.
- [8] R. Kreyer and J. Mukherjee, "The style of pop song lyrics: A corpus-linguistic pilot study," *Anglia-Zeitschrift Für Englische Philologie*, vol. 125, no. 1, 2007, pp. 1-20.
- [9] J. Falk, "We will rock you: A diachronic corpus-based analysis of linguistic features in rock lyrics," *Bachelor Thesis*, 2012.
- [10] N. Katznelson, J. Gelman, K. Lindblom, and M. Caput, "American song lyrics: A corpus-based research project featuring twenty years in rock, pop, country, and hip-hop."

REFERENCES

- [1] "Corpus," *Online Etymology Dictionary*, November 21, 2014, <http://dictionary.reference.com/browse/corpus>.
- [2] *Brown Corpus Manual*, November 21, 2014, <http://clu.uni.no/icame/brown/bcm.html>.
- [3] *The British National Corpus*, version 3 (BNC XML Edition). 2007. Distributed by Oxford University Computing Services on behalf of the BNC Consortium, <http://www.natcorp.ox.ac.uk/>.
- [4] "Billboard–music charts," *Music News*, Artist Photo Gallery & Free Video, November 21, 2014, <http://www.billboard.com/>.
- [5] Song Lyrics | MetroLyrics, November 21, 2014, <http://www.metrolyrics.com/>.
- [6] V. Werner, "Love is all around: A corpus-based study of pop lyrics," *Corpora*, vol. 7, no. 1, pp. 19-50, 2012.
- [7] G. Herk and U. Miethaner, "I can look through muddy water: Analyzing earlier African American English in blues lyrics (BLUR)," *English World-Wide*, pp. 205-209, 2005.
- [8] R. Kreyer and J. Mukherjee, "The style of pop song lyrics: A corpus-linguistic pilot study," *Anglia–Zeitschrift Für Englische Philologie*, vol. 125, no. 1, 2007.
- [9] J. Falk, "We will rock you: A diachronic corpus-based analysis of linguistic features in rock lyrics," *Bachelor Thesis*, 2012.
- [10] N. Katznelson, J. Gelman, K. Lindblom, and M. Caput, "American song lyrics: A corpus-based research project featuring twenty years in rock, pop, country, and hip-hop."

- [11] "Music: Top 100 songs | Billboard Hot 100 Chart," November 21, 2014, <http://www.billboard.com/charts/hot-100>.
- [12] "Welcome to Python.org," November 21, 2014, <https://www.python.org/>.
- [13] "XPath tutorial," November 21, 2014, <http://www.w3schools.com/xpath/>.
- [14] "Meet Scrapy," November 21, 2014, <http://scrapy.org/>.
- [15] "Installation guide," November 21, 2014, <http://doc.scrapy.org/en/latest/intro/install.html>.
- [16] "Pip 1.5.6: Python Package Index," November 21, 2014, <https://pypi.python.org/pypi/pip>.
- [17] "Lxml-XML and HTML with Python," November 21, 2014, <http://lxml.de/>.
- [18] "Scrapy tutorial," November 21, 2014, <http://doc.scrapy.org/en/latest/intro/tutorial.html>.
- [19] Lorde, "Royals," On *Pure Heroine* [CD], Morningside, New Zealand: Golden Age, 2013.
- [20] "Settings," November 21, 2014, <http://doc.scrapy.org/en/latest/topics/settings.html>.
- [21] I. Sag, T. Baldwin, R. Bond, A. Copestake, and D. Flickinger, "Multiword expressions: A pain in the neck for NLP," in *Proceedings of the 3rd International Conference on Intelligent Text Processing and Computational Linguistics*, 2002, pp. 1-15.
- [22] "Google translate," November 21, 2014, <http://translate.google.com/>.
- [23] T. Baldwin and A. Villavicencio, "Extracting the unextractable: A case study on verb-particles," *Proceedings of the 6th Conference on Natural Language Learning*, 2002, pp. 98-104.

- [24] T. Baldwin, "The deep lexical acquisition of English verb-particle constructions," *Computer Speech and Language, Special Issue on Multiword Expressions*, vol. 19, no. 4, pp. 398-414, 2005.
- [25] S. Kim and T. Baldwin, "Detecting compositionality of English verb-particle constructions using semantic similarity," *Proceedings of PACLING 2007*, 2007, pp. 40-48.
- [26] T. Baldwin and S. Kim, "How to pick out token instances of English verb-particle constructions," *Journal of Language Resources and Evaluation*, no. 44, pp. 97-113, 2010.
- [27] S. Kim and T. Baldwin, "Automatic identification of English verb particle constructions using linguistic features," *Proceedings of the Third ACL-SIGSEM Workshop on Prepositions*, 2006, pp. 65-72.
- [28] W. Daelemans, J. Zavrel, K. Van der Sloot, and A. Van den Bosch, *TiMBL: Tilburg Memory Based Learner*, 2004.
- [29] A. Smith, "Breaking bad: Extraction of verb-particle constructions from a parallel subtitles corpus," *SIGLEX-MWE: Workshops on Multiword Expressions*, 2014.
- [30] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. Beijing: O'Reilly, 2009.
- [31] "Natural language toolkit," November 21, 2014, <http://www.nltk.org/>.
- [32] "NumPy," November 21, 2014, <http://www.numpy.org/>.
- [33] B. Mars, "Young girls," On *Unorthodox Jukebox* [CD], Atlantic, 2012.
- [34] *Dictionary.com*, November 21, 2014, <http://dictionary.reference.com/>.
- [35] "Dictionary, encyclopedia and thesaurus," November 21, 2014, <http://www.thefreedictionary.com/>.
- [36] *Urban Dictionary*, November 21, 2014, <http://www.urbandictionary.com/>.

APPENDICES

Source Code

SongExtractor.py

```

from urlparse import urlparse
import requests
from lxml import html
import re

NUM_PAGES = 100

file = open('songs.txt', 'w')
prevurl = "http://www.10000000.com/albums/100-100/"

# A dictionary in which the key is the song and the value is the artist
songs = {}

# Strips everything before and including the quote
p = re.compile('\"')

# Strips trailing whitespace
q = re.compile(' ')

striptabs = re.compile(' ')
stripnewlines = re.compile('\n')

# This function extracts the song and artist from the HTML
# and strips excess characters, then stores
# the song and artist in the songs dictionary
def extractSongArtist(item):
    song = item.xpath('//p/text()')
    artist = item.xpath('//p/text()')

    # Strip unnecessary characters in song and artist
    art = p.sub('', str(artist))
    art = q.sub('', art)
    art = striptabs.sub('', art)
    art = stripnewlines.sub('', art)
    so = p.sub('', str(song))
    so = q.sub('', so)
    art = art.lower()
    so = so.lower()

    if not so in songs:
        songs[so] = art

for week in range(1, NUM_PAGES):
    page = requests.get(prevurl)
    tree = html.fromstring(page.text)

    for item in tree.xpath('//p[@class="song-title"]'):
        extractSongArtist(item)

    prevurl = tree.xpath('//a[@class="next-page"]/@href')[0]
    prevurl = urlparse(prevurl).geturl()
    page = requests.get(prevurl)
    tree = html.fromstring(page.text)

    for item in tree.xpath('//p[@class="song-title"]'):
        extractSongArtist(item)

```

APPENDIX A

Source Code

SongExtractor.py

```

from urlparse import urljoin
import requests
from lxml import html
import re

NUM_WEEKS = 100

file = open('songs.txt', 'w')
prevurl = "http://www.billboard.com/charts/hot-100"

# A dictionary in which the key is the song and the value is the artist
songs = {}

# Strips everything before and including the quote
p = re.compile('^.*?(\'|\")')

# Strips trailing whitespace
q = re.compile('\s*(\'|\")$')

striptabs = re.compile('(\t)+')
stripnewlines = re.compile('\n')

# This function extracts the song and artist
# and strips excess characters, then stores
# the song and artist in the songs dictionary
def extractSongArtist(item):
    song = item.xpath('h2/text()')
    artist = item.xpath('h3/a/text()')

    # Strip unnecessary characters in song and artist
    art = p.sub('', str(artist))
    art = q.sub('', art)
    art = striptabs.sub('', art)
    art = stripnewlines.sub('', art)
    so = p.sub('', str(song))
    so = q.sub('', so)
    art = art.lower()
    so = so.lower()

    if not so in songs:
        songs[so] = art

for week in range(1, NUM_WEEKS):
    page = requests.get(prevurl)
    tree = html.fromstring(page.text)

    for item in tree.xpath('//div[@class="row-title"]'):
        extractSongArtist(item)

    prevurl = tree.xpath('//nav[@id="chart-nav"]/a[@class="prev"]/@href')[0]
    prevurl = urljoin("http://www.billboard.com", prevurl)
    page = requests.get(prevurl)
    tree = html.fromstring(page.text)

for item in tree.xpath('//div[@class="row-title"]'):
    extractSongArtist(item)

```

```

for song in songs:
    file.write(songs[song] + '|' + song + '\n')royals_spider.py
import scrapy
import string
import re
from string import ascii_lowercase
from royals.items import RoyalsItem
from lxml import html
import requests

input = open('songs.txt')
artistsSongs = input.read().splitlines()
inputsongs = {}

def findmatches(links, artistname, numwords):
    matchesfound = []
    for link in links:
        stringtomatch = artistname[0] + "-"
        for word in range(1, numwords):
            stringtomatch = stringtomatch + artistname[word] + "-"
        if re.search(stringtomatch, link):
            matchesfound.append(link)
    return matchesfound

for line in artistsSongs:
    stripast = re.compile('\+')
    line = stripast.sub('.', line)
    artsong = string.split(line, '|')
    song = artsong[1]

    # assigning artists and songs to a dictionary in which
    # the key is the song and the value is the artist.

    # If there are multiple artists, only use the first.
    # Accomplish this by only taking the artist before either
    # the first comma or the word "featuring"
    firstartist = artsong[0].split(',')
    firstartist = firstartist[0].split(' featuring')
    inputsongs[song] = firstartist[0]

urlbeginning = "http://www.metrolyrics.com/artists-"
urlending = ".html"
# Contains the urls to the artist pages alphabetically.
# letterurl['a'] = http://metrolyrics.com/artists-a.html
letterurl = {}
for letter in ascii_lowercase:
    letterurl[letter] = urlbeginning + letter + urlending

# Metrolyrics puts artists starting with a digit on the page
# metrolyrics.com/artists-1.html

letterurl['1'] = urlbeginning + '1' + urlending

urls = []

artistlinksongnames = []

```



```

# first two words. This is done to handle cases of multiple artists.
if artistname[0] is 'the':
    numwords = 2
else:
    numwords = 1
matchedlinks = artistlinks[firstletter]
while True:
    # Iterates through each link in the letter
    matchedlinks = findmatches(matchedlinks, artistname, numwords)

    # If we have used all the words in the artist name, but
    # more than one link matched
    # This is used when one artist name is a subset of another
    # Ex: taylor swift vs. taylor swift cover band
    if (numwords == len(artistname)) and (len(matchedlinks) > 1):
        # Find and use the shortest.
        shortest = matchedlinks[0]
        for matchedlink in matchedlinks:
            if len(matchedlink) < len(shortest):
                shortest = matchedlink

        matchedlinks = []
        matchedlinks.append(shortest)

    # If we found the link, leave the loop
    if len(matchedlinks) < 2: #matching last
        break

    matchedlinks = artistlinks[firstletter]
    numwords +=1
    # If we have matched two songs, start over, matching
    # one more word (back through the while loop)
if len(matchedlinks) == 1:
    artistlinksongnames.append((matchedlinks[0], song))

for tuple in artistlinksongnames:
    #tuple[0] contains links to artist page
    #tuple[1] contains song name in a string
    artistpage = requests.get(tuple[0])
    tree = html.fromstring(artistpage.text)
    # iterate through each song and find its link
    songbyword = tuple[1].split()
    for x in range(0, len(songbyword)):
        nopunct = re.compile('(,|\.|&|\'|\"|\(|\)|\{|\}|\[|\]|)')
        songbyword[x] = nopunct.sub('', songbyword[x])
    numwords = len(songbyword)
    matched = False
    while True:
        for songlink in tree.xpath('//td/a/@href'):
            #find the song it matches
            stringtomatch = songbyword[0] + "-"
            for word in range(1, numwords):
                stringtomatch = stringtomatch + songbyword[word] + "-"
            rawstring = stringtomatch.encode('string-escape')
            if re.search(rawstring, songlink):
                urls.append(songlink)
                matched = True
                break

```

```
if matched or numwords < 1:
    break
else:
    numwords -= 1

class LyricSpider(scrapy.Spider):
    name = "royals"
    allowed_domains = ["metrolyrics.com"]
    start_urls = urls

    def parse(self, response):
        for sel in response.xpath('//p[@class="verse"]/text()').extract():
            item = RoyalsItem()
            item['lyrics'] = sel

            yield item
```

APPENDIX B

Scrapy Project User Documentation

Purpose:

The *SongExtractor* program loads the *Billboard Hot 100* chart [4] and navigates the HTML to extract song titles and artist names from the most recent specified number of weeks. One hundred songs per week are extracted, discarding duplicate songs. The range of weeks ends at the current week and begins a user-specified number of weeks prior, resulting in the n most current weeks, where n is the specified number of weeks.

Prerequisites:

APPENDIX B

- Internet connection
- Python 2.6-2.7.8 *SongExtractor* User Documentation is available as a free download at <https://www.python.org/downloads/> [34].
- The *lxml* XML Toolkit is also required and available as a free download at <http://lxml.de/> [21].

Steps for use:

1. Ensure prerequisite software is installed.
2. To set the desired number of weeks for extraction of song and artist names:
 - a. Open *SongExtractor.py*.
 - b. In line 6, change the value of *NUM_WEEKS* to the number of weeks desired. The default value is 100.
 - c. Save

SongExtractor User Documentation

Purpose:

The *SongExtractor* program loads the *Billboard Hot 100* chart [4] and navigates the HTML to extract song titles and artist names from the most recent specified number of weeks. One hundred songs per week are extracted, discarding duplicate songs. The range of weeks ends at the current week and begins a user-specified number of weeks prior, resulting in the n most current weeks, where n is the specified number of weeks.

Prerequisites:

- Internet connection
- Python 2.6-2.7.8 is required for use of this program. It is available as a free download at <https://www.python.org/downloads/> [34].
- The *lxml* XML Toolkit is also required and available as a free download at <http://lxml.de/> [21].

Steps for use:

1. Ensure prerequisite software is installed.
2. To set the desired number of weeks for extraction of song and artist names:
 - a. Open *SongExtractor.py*.
 - b. In line 6, change the value of *NUM_WEEKS* to the number of weeks desired. The default value is 100.
 - c. Save.

3. Run the *SongExtractor* program.
 - a. Run a command prompt and navigate to the directory containing *SongExtractor.py*.
 - b. Enter the following command: `python SongExtractor.py`.

Output:

Once the *SongExtractor* program has finished, the output will be in the file *songs.txt* within the same directory as *SongExtractor.py*. The output is formatted with one song per line, first the song name, then the separator |, then the artist name. See below for example output, reproduced from Figure 3.

```
arctic monkeys|do i wanna know?  
b.o.b featuring t.i. & juicy j|we still in this b****  
big sean featuring lil wayne & jhene aiko|beware  
eric church|give me back my hometown  
cole swindell|hope you get lonely tonight
```

The output file *songs.txt* can be directly used as input to the *Royals* program described in Appendix C.

Royals User Documentation

Output

The *Royals* program collects song lyrics from *Internet* and stores them in an output file. This output file contains a corpus of song lyrics. The songs to be collected are specified in an input file, *songs.txt*.

Provided files

royals_spider.py, items.py, settings.py, pipelines.py, scrapy.cfg

Prerequisites

- Internet connection.
- Python 2.6-2.7.6 is required for use of this program. It is available as a free download at <http://www.python.org> [4].
- The *Beautiful Soup* Toolkit is also required and available as a free download at <http://www.crummy.com> [21].
- *Scrapy* is a web scraping software available at <http://scrapy.org> [22].
- A *songs.txt* file containing song titles and artist names in a specified format.

This file can either be produced by the program *SongExtractor* or made manually. If made manually, adhere to the same format as the output file described in Appendix B.

Steps for use

1. Save provided folder *royals* in the *Scrapy* directory.
2. Place *songs.txt* in the *Scrapy/royals* directory.

3. On a command prompt, *Royals* User Documentation directory.

4. Enter the command: `scrapy crawl royals -o corpus.json`

Purpose:

The *Royals* program collects song lyrics from *Metrolyrics* and stores them in an output file. This output file contains a corpus of song lyrics. The songs to be collected are specified in an input file, *songs.txt*.

Included files:

royals_spider.py, *items.py*, *settings.py*, *pipelines.py*, *scrapy.cfg*

Prerequisites:

Internet connection.

- Internet connection
- Python 2.6-2.7.8 is required for use of this program. It is available as a free download at <https://www.python.org/downloads/> [34].
- The lxml XML Toolkit is also required and available as a free download at <http://lxml.de/> [21].
- Scrapy is a web scraping software available at <http://scrapy.org/> [23].
- A *songs.txt* file containing song titles and artist names in a specified format. This file can either be produced by the program *SongExtractor* or made manually. If made manually, adhere to the same format as the output file described in Appendix B

Steps for use:

1. Save provided folder *royals* in the *Scrapy* directory.
2. Place *songs.txt* in the *Scrapy/royals* directory.

3. On a command prompt, navigate to the *Scrapy/royals* directory.
4. Enter the command: *scrapy crawl royals -o corpus.json*
 - a. The file name after the `-o` in the command will contain the output. It can have any name with a *.json* extension.

Output:

Once the *Royals* program has finished, the output file specified on the command line will contain the lyrics of the songs from the input file. This process can take several minutes depending on the number of items in the input file and the Internet connection.