In this post, we’ll return to the Kaggle data containing information on Pitchfork music reviews. I used this dataset to cluster music genres. In the current post, I will use R and the tidytext package (and philosophy) to examine the text of the music reviews. Specifically, the goal of the analysis described in this post will be to track the course of positive and negative sentiment use across the length of the review texts.

**The Data**

The data are available on the [Kaggle website](https://www.kaggle.com/nolanbconaway/pitchfork-data). I did an extensive data munging exercise to clean and prepare the data for analysis. As we will later be interested in examining sentiment use in the different genres, I excluded reviews with more than 1 genre from the scope of this analysis, leaving us with 12,147 review texts. In this post we will focus on 3 columns in our dataset: one column containing a unique review identifier, one column with the review text, and one column that contains the genre of the album being reviewed (with the following options: *electronic*, *experimental*, *folk/country*, *global*, *jazz*, *metal*, *pop/rnb*, *rap* and *rock*).

**The tidytext Approach**

In order to prepare our data for analysis, we must turn it into a tidy dataset. The basic idea behind the tidytext framework is that we represent our data with 1 line per token (a sub-division of a longer text, typically but not always a single word), keeping track of important meta-data (e.g. the id number of the review the word appears in) in additional columns.

I have always used the “[bag of words](https://en.wikipedia.org/wiki/Bag-of-words_model)” framework, in which each text is kept in a single line of the dataset. This traditional approach is very handy when doing, for example, [predictive analysis](http://methodmatters.blogspot.com/2017/06/analyzing-wine-data-in-python-part-3.html) with text. However, the tidytext philosophy lets us think about and analyze our data in a slightly different way.

Predictive Analysis with Text

**The Data**

I considered all of the text from official studio albums for each artist (list created by consulting the respective artist Wikipedia pages). This analysis was undertaken in the beginning of November 2017, and all albums released at that point were used. For Nas, the following albums were included: **Illmatic**, **It Was Written**, **I Am…**, **Nastradamus**, **Stillmatic**, **The Lost Tapes**, **God’s Son**, **Street’s Disciple**, **Hip Hop is Dead**, **Untitled**, and **Life Is Good**.

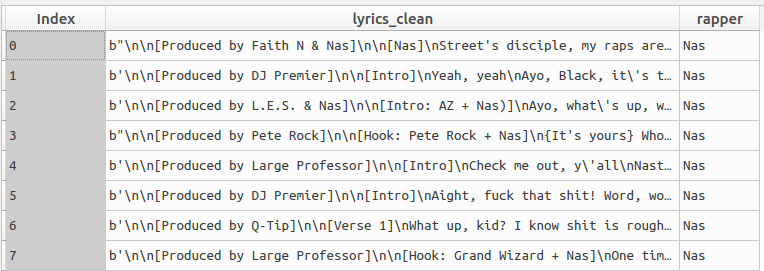
I included the following DOOM albums: **Operation: Doomsday.**, **Take Me to Your Leader**, **Vaudeville Villain**, **Madvillainy**, **Venomous Villain**, **MM.. FOOD**, **The Mouse and The Mask**, **Born Like This**, **Key to the Kuffs**, and **NehruvianDOOM**. Note that albums of instrumentals and work released prior to the adoption of the DOOM persona are not considered, while albums created with collaborators (e.g. **Madvillainy**, **The Mouse and the Mask**, **Key to the Kuffs**, etc.) are included.

The lyrics were scraped from genius.com, a website that allows users to transcribe and annotate song lyrics. Although I won’t go into the details in this blog post, I used the Python module Beautiful Soup to scrape the lyrics. Beautiful Soup works really well, is relatively easy to use, and made the task of obtaining these data very straightforward.

The lyric dataset contains 1 line for each song. For the purposes of the present analysis, we will focus on two main columns in our dataset. The first is the column containing the song lyrics, called “*lyrics\_clean*” (though, as we’ll see below, this field still requires some serious cleaning before being ready for analysis). The second is the column called “*rapper*” which can take on two values: “Nas” or “DOOM.”

My scraping program returned 340 songs in total; our original raw dataset therefore contains this many rows.

The head of this dataset can be seen below:



**Data Pre-Processing & Exploratory Analysis**

**Text Cleaning**

As can be seen in the above screenshot, the text is pretty messy. There are carriage returns (indicated by “*\n*”), information about the producers (e.g. “*Faith N & Nas*” in the first line), the artist (e.g. “*Nas*,” “*Grand Wizard + Nas*”), and the song structure (“*Intro*,” “*Hook*,” etc.).

In order to clean up the text, we’ll adapt a function we’ve used in previous posts. This function removes the carriage returns in the text, removes the text between brackets (e.g. [Intro]), removes non-letters and numbers, converts the text to lower case, removes stopwords, stems the text, and removes any remaining words with only 1 character. Note that we have to slightly adapt our stemming algorithm, otherwise it stems “Nas” (one of the artists in our dataset) to “na.”

Our function looks like this:

# import the necessary libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import re

from nltk.corpus import stopwords

from nltk.stem.porter import \*

stemmer = PorterStemmer()

# regex to remove unnecessary text from the lyrics such as [Intro], [Chorus]

pattern = re.compile("\[(.\*?)\]")

# define our text cleaning function

def text\_to\_words(raw\_text):

#

# 1. remove the extra carriage returns in the text

no\_carriage = raw\_text.replace('\\n', ' ')

#

# 2. remove text between brackets []

no\_brackets = re.sub(pattern, "", no\_carriage)

#

# 3. Remove non-letters and numbers

letters\_only = re.sub("[^a-zA-Z]", " ", no\_brackets)

#

# 4. Convert to lower case, split into individual words

words = letters\_only.lower().split()

#

# 5. define stop words. make it a set (faster)

stops = set(stopwords.words("english"))

#

# 6. Remove stop words

meaningful\_words = [w for w in words if not w in stops] #returns a list

#

# 7. Stem words

# exception for "Nas" which is stemmed to "Na" by default

# big-ups to:

# https://stackoverflow.com/questions/24517722/how-to-stop-nltk-stemmer-from-removing-the-trailing-e

# https://stackoverflow.com/questions/17321138/one-line-list-comprehension-if-else-variants

singles = [stemmer.stem(word) if word.lower()!= 'nas' else

word for word in meaningful\_words]

#

# 8. remove words with length of less than 2

remaining\_words = [x for x in singles if not len(x) < 2]

#

# 9. Join the words back into one string separated by space,

# and return the result.

remaining\_words\_joined = " ".join(remaining\_words)

#

# 10. return the remaining words in a single joined string

return(remaining\_words\_joined)

Which we apply to our dataset (called *nas\_doom*) with a list comprehension:

# apply the function to our lyrics column

nas\_doom['modeling\_text\_data'] = [text\_to\_words(text) for text in nas\_doom.lyrics\_clean]

We can look at the first text in our raw data, which begins like this (I can’t share all the lyrics because I don’t have the rights to them):

"\\n\\n[Produced by Faith N & Nas]\\n\\n[Nas]\\nStreet\'s disciple, my raps are trifle\\nI shoot slugs from my brain just like a rifle"

And the data cleaned by the function, which begins like this:

"street discipl rap trifl shoot slug brain like rifl"

We appear to have successfully removed the mess and retained only the essential parts of the text!

**Exploring Word Counts by Artist**

Before proceeding to the modeling, let’s first explore the word counts for the songs in our dataset. We can create a count of the words in the cleaned texts with the function presented below. When exploring the word counts, I noticed that some were quite low. This happened with instrumental tracks for which there are no or very few lyrics. I therefore removed these songs from the data. The following code counts the words of the cleaned texts and removes rows for which the word count is 10 or fewer (there were 6 such songs):

# define a function to count the number of cleaned words

def word\_count(text):

words = text.split()

wc = len(words)

return(wc)

# apply the function to the text data

# create a column with the wordcount value

# in our master dataset

nas\_doom['wordcount'] = [word\_count(text) for text in nas\_doom.modeling\_text\_data]

# remove songs with less than 11 words after cleaning

# and reset the index

nas\_doom = nas\_doom[nas\_doom['wordcount']>10].reset\_index(drop = True)

With these rows removed, our dataset contains 334 songs: 185 by Nas and 149 by DOOM. Let’s use seaborn to examine the distributions of word counts for the two artists:

# plot the wordcount distributions for the

# respective artists

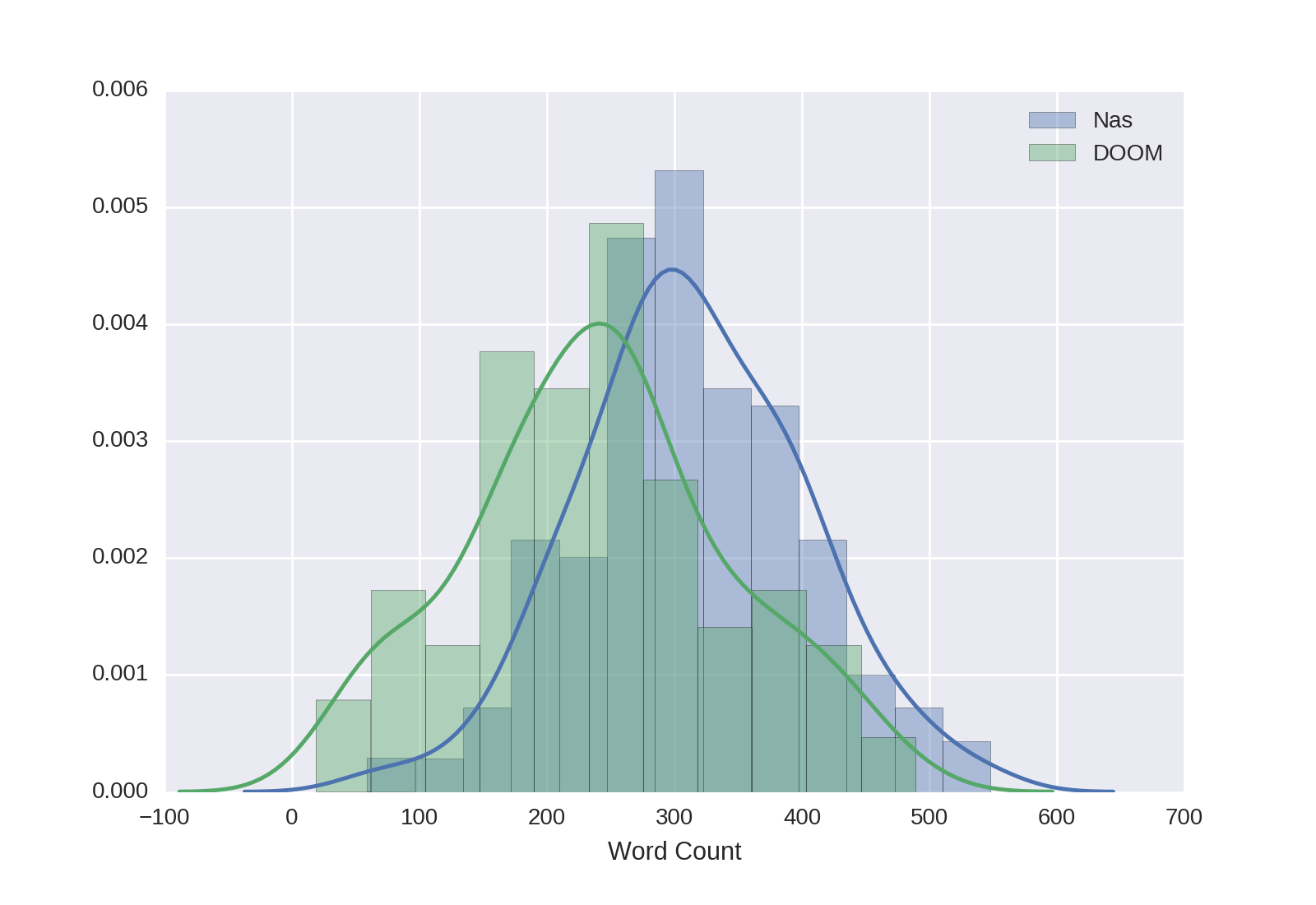
ax = sns.distplot(nas\_doom[nas\_doom.rapper == 'Nas']['wordcount'], label="Nas")

ax = sns.distplot(nas\_doom[nas\_doom.rapper == 'DOOM']['wordcount'], label="DOOM")

ax.set(xlabel='Word Count')

plt.legend()

Which yields the following plot:



It is clear that Nas songs tend to have more words than DOOM songs. The mean (cleaned) word count for Nas is 313.14 (*SD* = 88.34) while the mean for DOOM is 240.75 (*SD* = 103.11).

**Most Frequent Words**

As a final exploratory analysis, let’s examine the most frequently-occurring words in our cleaned data. We can use sklearn’s word count vectorizer to count the occurrences of each word, and visualize the results using seaborn.

The code below calculates the frequency of each word (called ‘unigrams’ in the natural language processing world) and two-word combination (e.g. ‘bigrams’). It then stores the unigrams/bigrams and their respective frequencies in a dataframe, and plots the 15 most frequent ones.

# let's look at frequent words in the corpus

# using sklearn's count vectorizer

# import the count vectorizer function

from sklearn.feature\_extraction.text import CountVectorizer

# define the vectorizer, including unigrams and bigrams

# with the code: ngram\_range=(1, 2)

vectorizer\_frequency = CountVectorizer(input = 'content', ngram\_range=(1, 2),

min\_df = 25, binary = False)

# apply the vectorizer to the processed texts

counts = vectorizer\_frequency.fit\_transform(nas\_doom['modeling\_text\_data'])

# turn the dtm matrix to a numpy array to sum the columns

counts\_array = counts.toarray()

# sum up the counts of each word

dist = np.sum(counts\_array, axis=0)

# extract the names of the features

vocab = vectorizer\_frequency.get\_feature\_names()

# make it a dataframe

topwords = pd.DataFrame(dist, vocab, columns = ["Word\_Count"])

# add 'word' as a column

topwords = topwords.reset\_index()

# sort the words by frequency

topwords = topwords.sort\_values(by = 'Word\_Count',

ascending=False)

# define the color palette

mypal = sns.light\_palette(sns.xkcd\_rgb["mulberry"], n\_colors = 15, reverse = True)

# plot the top words and set the axis labels

topwords\_plot = sns.barplot(y = 'index', x ="Word\_Count",

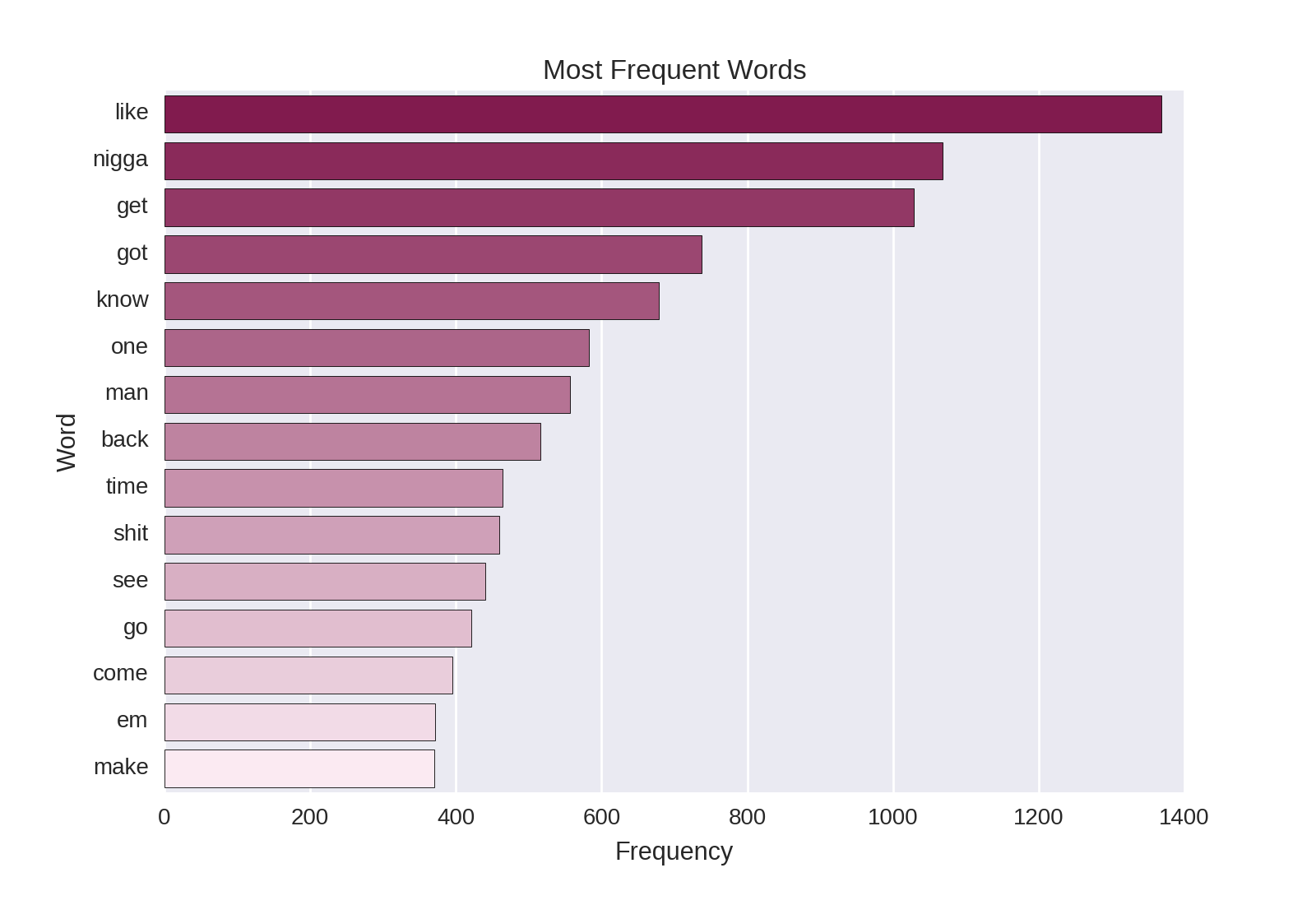
data=topwords[0:15], orient ='h', palette = mypal)

topwords\_plot.set\_title('Most Frequent Words')

topwords\_plot.set(xlabel='Frequency')

topwords\_plot.set(ylabel='Word')

Which yields the following plot:



“Like” is the most frequently occurring word in our data, appearing 1368 times (on average 4.02 times per song). In my experience, comparing objects, people or actions is quite frequent in rap lyrics (indeed, the first line from Nas’ Illmatic quoted above -*Street’s disciple, my raps are trifle / I shoot slugs from my brain just like a rifle*- is a great example). I note that others have also pointed out that simile use is quite common in hip-hop.

Other notable frequently-occurring words include “get” and “got.” Clearly, obtaining other things is an important subject in many of these songs. It was interesting to see that “time” (or times - remember that our stemming converts both to “time”) occurred so frequently in these data. Some examples include: *time to grind*, *hard times*, *tough times*, *took more time to write in my book of rhymes*, *triple that times three*, *doing time* [serving a prison sentence], and *have an iller rhyme, at least by Miller Time*.

**Modeling**

**Document-Term Matrix and Train-Test Split**

Before we can begin modeling, we need to represent our text data in a matrix format. I go over this in more detail a previous blog post; in short we create a document-term matrix (documents in the rows, words in the columns) and include a column for every word in our dataset. We will extract binary indicators - if a word is present in a song, the relevant column for the given row takes on a value of 1; otherwise it takes on a value of zero. We only retain words which appear in 25 or more songs.

We use sklearn’s count vectorizer, specifying we want binary indicators, unigrams and bigrams (as we did above), and then split the data into training and test samples with the following code:

# we will use binary indicators for the words, and include bigrams

# we'll only keep unigrams/bigrams that appear in at least 25 songs

from sklearn.feature\_extraction.text import CountVectorizer

vectorizer\_count = CountVectorizer(input = 'content', ngram\_range=(1, 2),

min\_df = 25, binary = True)

# apply our vectorizer to the processed texts

predictive\_features = vectorizer\_count.fit\_transform(nas\_doom['modeling\_text\_data'])

# split the data into train and test sets

from sklearn.cross\_validation import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(predictive\_features,

nas\_doom["rapper"], test\_size=0.30,

random\_state=5)

We can see how many different features were extracted using this method with the following code:

print(predictive\_features.shape)

With our data and the parameters supplied above, we extract 589 different text features.

**Random Forest**

We will first create a random forest model with our training data to predict the rapper, using the words from the song lyrics as predictive features. We will use a similar approach for building the random forest model as we did in a previous post. We then extract the feature importances from our model and plot them using seaborn. The following code accomplishes all of this:

# Prepare and run Random Forest

# import the random forest package

from sklearn.ensemble import RandomForestClassifier

# create the random forest object

# specify we want 500 trees

rforest = RandomForestClassifier(n\_estimators = 500, n\_jobs=-1)

# fit the training data and create the decision trees

rforest\_model = rforest.fit(X\_train,y\_train)

# extract feature importances

df\_featimport = pd.DataFrame([i for i in zip(vectorizer\_count.get\_feature\_names(),

rforest\_model.feature\_importances\_)],

columns=["features","importance"])

# create a palette using XKCD colors

# https://xkcd.com/color/rgb/

mypal = sns.light\_palette(sns.xkcd\_rgb["bottle green"], n\_colors = 15,

reverse = True)

# plot the top 15 features

ax = sns.barplot(x="importance", y="features",

data=df\_featimport.sort('importance', ascending=False)[0:15],

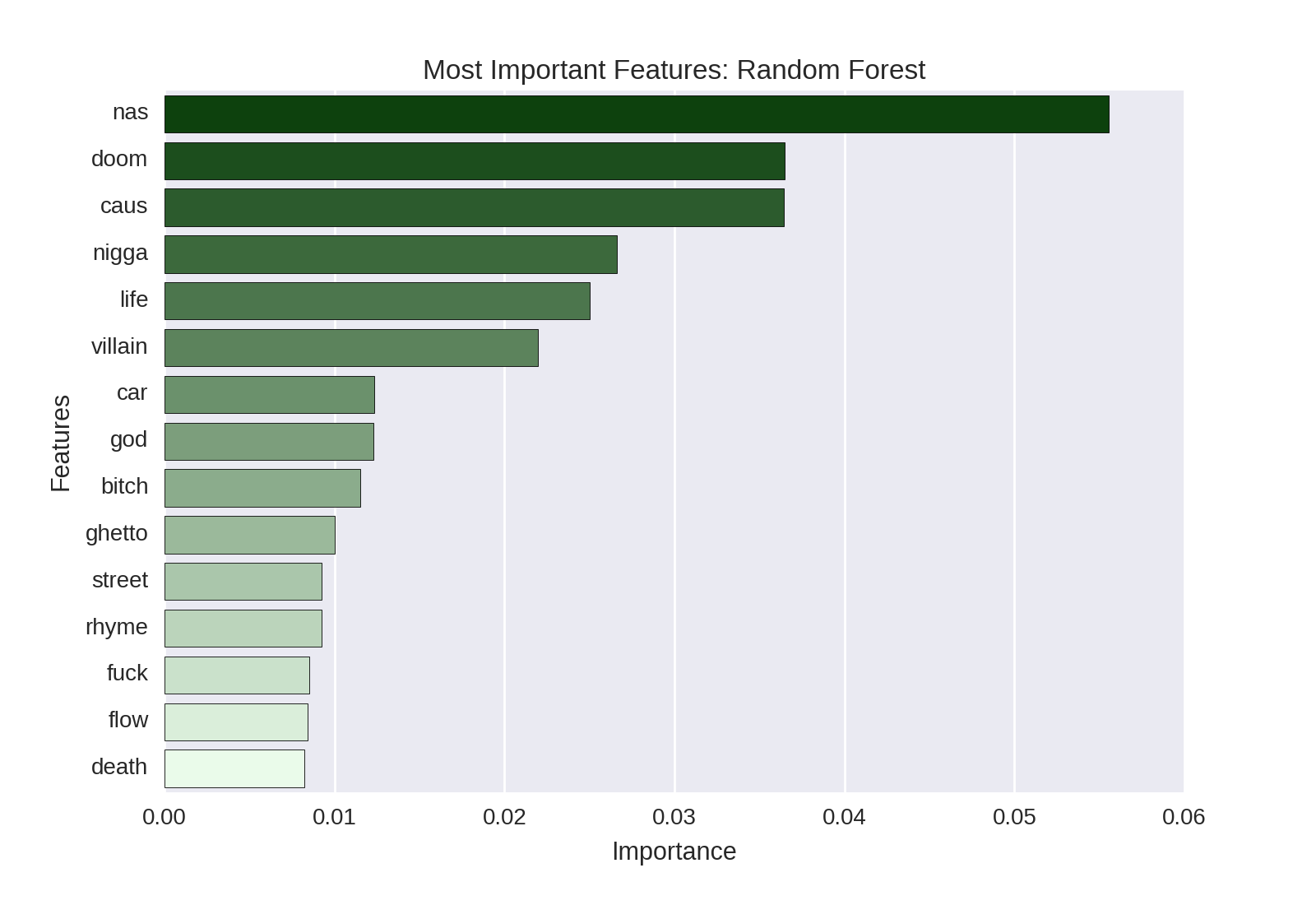
label="Total", palette = mypal)

# set axis labels

ax.set\_title('Most Important Features: Random Forest')

ax.set(xlabel='Importance', ylabel='Features')

And gives us the following plot of the feature importances:



We can see that the rappers’ names are (of course) very predictive: *Nas*, *DOOM*, and *Villian* (a moniker sometimes used by DOOM) are among the most-predictive features in our text data. References to urban life (often typical of Nas’ work, e.g. *ghetto* and *street*) also feature prominently.

Let’s use our model to predict the rappers of the songs in our hold-out test data and create a confusion matrix:

# predict class on test data

preds\_class\_rf = rforest\_model.predict(X\_test)

# confusion matrix

pd.crosstab(preds\_class\_rf, y\_test)

Which gives us the following confusion matrix:

| **Actual Class:** | **DOOM** | **Nas** |
| --- | --- | --- |
| Predicted Class: |  |  |
| DOOM | 40 | 2 |
| Nas | 4 | 55 |

On the whole, we do quite well! There are 2 songs in our test data that are by Nas but which our model predicts are by DOOM. There are 4 songs in our test data which are by DOOM but which our model predicts are by Nas.

**LASSO Regression**

We will now create a LASSO regression model to predict the rapper, given the song lyrics. We model using the same training and test sets as we did for the random forest model above. (For more details about LASSO regression please see this previous post).

We then extract the features with the largest positive and negative penalized coefficients and plot them. The following code accomplishes all of this:

# LASSO logistic regression

# import the module

from sklearn.linear\_model import LogisticRegressionCV

# define the lasso model

log\_model = LogisticRegressionCV(penalty='l1', solver='liblinear', cv=5,

random\_state= 42)

# and fit it to the training data

log\_model.fit(X\_train, y\_train)

# make a dataframe with the features and coefficient values

coefficients = pd.concat([pd.DataFrame(vectorizer\_count.get\_feature\_names(),

columns = ['features']),pd.DataFrame(np.transpose(log\_model.coef\_),

columns = ['penalized\_coefficients'])], axis = 1)

# remove coefficients with values of zero

# and those with an absolute value of less than 1.6

coefficients\_trimmed = coefficients[(coefficients.penalized\_coefficients != 0) &

(abs(coefficients.penalized\_coefficients) > 1.6)]

# sort and reset index

coefficients\_trimmed = coefficients\_trimmed.sort\_values('penalized\_coefficients',

ascending=False).reset\_index(drop = True)

# plot the coefficients, specifying color for words more predictive

# of Nas or DOOM, respectively

# https://stackoverflow.com/questions/31074758/how-to-set-a-different-color-to-the-largest-bar-in-a-seaborn-barplot

clrs = ['darkolivegreen' if (x < 0) else 'orangered' for x in

coefficients\_trimmed.penalized\_coefficients ]

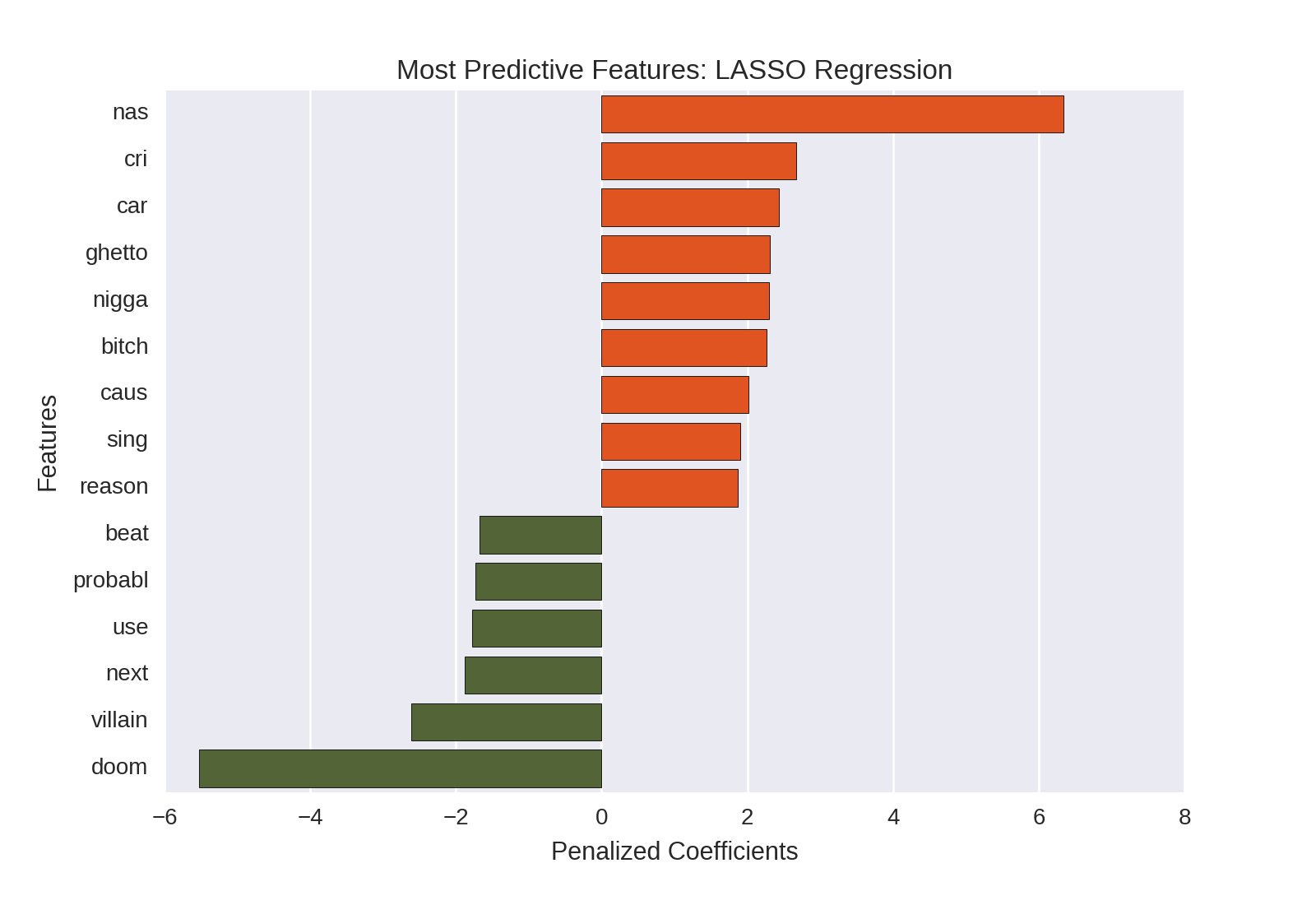
ax = sns.barplot(x="penalized\_coefficients", y="features",

data=coefficients\_trimmed, palette = clrs)

ax.set\_title('Most Predictive Features: LASSO Regression')

ax.set(xlabel='Penalized Coefficients', ylabel='Features')

And returns the following plot of the largest positive and negative penalized coefficients:



Unlike the random forest feature importances, the penalized regression coefficients give us a sense of the *direction* of given feature; e.g. whether it is positively or negatively predictive of Nas (vs. DOOM) being the rapper. The LASSO regression coefficients are therefore more informative as they allow us to understand the nature of the relationship between a given predictor and the outcome variable. With the random forest feature importances shown above, we simply know that a feature is predictive, but not in which direction.

Note that I have colored the positive coefficients (e.g. features which, if they are present, indicate that Nas is the likely rapper) in orange and the negative coefficients (e.g. features which, if present, indicate that DOOM is the likely rapper) in green. This makes it quick and easy to see which words are more predictive of Nas vs. DOOM, respectively. I would argue that the LASSO regression and the plot of the penalized coefficients provide more insight into our problem than the random forest and the resulting feature importances plot.

We can use the LASSO regression model to predict the rapper for each song in our hold-out test dataset and create a confusion matrix with the following code:

# predict class on test data

preds\_class\_log = log\_model.predict(X\_test)

# confusion matrix

pd.crosstab(preds\_class\_log, y\_test)

Which produces the following confusion matrix:

| **Actual Class:** | **DOOM** | **Nas** |
| --- | --- | --- |
| Predicted Class: |  |  |
| DOOM | 41 | 4 |
| Nas | 3 | 53 |

The results seem comparable to the random forest model above!

**Model Comparison**

Let’s compare the model performance in a more structured way via ROC curves.

We first calculate the predicted probabilities for both models, and then create and visualize the ROC curves with the following code (adapted from that used in previous posts):

# pretty ROC curves

from sklearn import metrics

from sklearn.metrics import \*

# predict random forest probabilities on test data

preds\_prob\_rf = rforest\_model.predict\_proba(X\_test)

# predict lasso regression probabilities on test data

preds\_prob\_log = log\_model.predict\_proba(X\_test)

# create the ROC curve plot:

fpr, tpr, thresholds = metrics.roc\_curve(pd.get\_dummies(y\_test).ix[:, 1],

preds\_prob\_rf[:,1])

auc1 = auc(fpr,tpr)

plt.plot(fpr, tpr,label="AUC Random Forest:{0:0.2f}".format(auc1),

color='red', linewidth=2)

fpr, tpr, thresholds = metrics.roc\_curve(pd.get\_dummies(y\_test).ix[:, 1],

preds\_prob\_log[:,1])

auc1 = auc(fpr,tpr)

plt.plot(fpr, tpr,label="AUC LASSO Logistic Regression:{0:0.2f}".format(auc1),

color='blue', linewidth=2)

plt.plot([0, 1], [0, 1], 'k--', lw=1)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

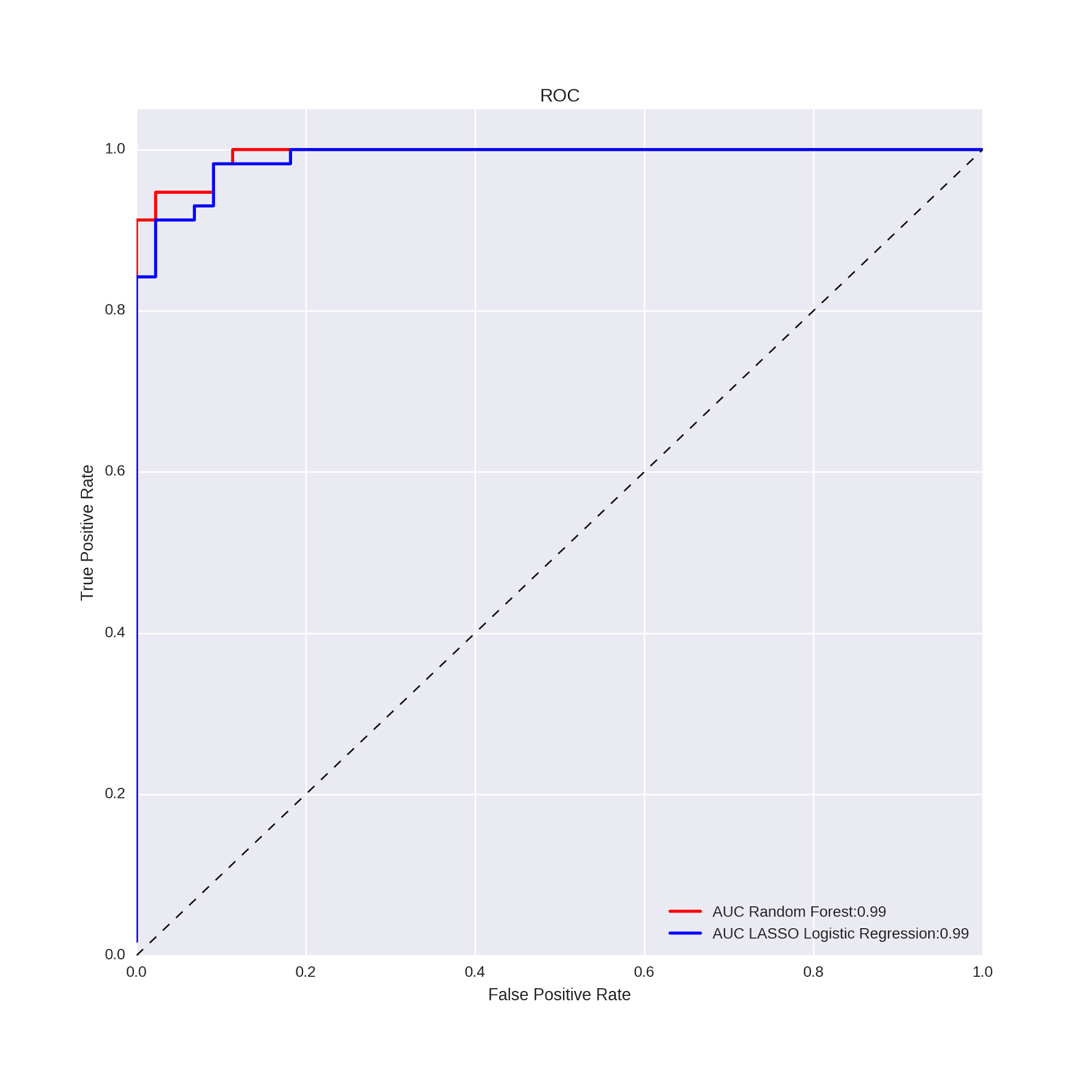
plt.ylabel('True Positive Rate')

plt.title('ROC')

plt.grid(True)

plt.legend(loc="lower right")

Which gives us the following plot:



The random forest and the LASSO regression model have identical ROC values!

**Substantive Interpretation - What Have We Learned From This Analysis?**

Finally, what does this analysis teach us about the language and thematic content of songs on the Nas and DOOM albums?

We first see that name-checks (e.g. rappers mentioning themselves in the lyrics) are the best predictors of the song artist. Indeed, *Nas*, *DOOM* and *Villian* are the most predictive features in this analysis.

Second, Nas’ depiction of the difficulties of urban life (a theme which characterized his ground-breaking debut album *Illmatic* and one which is viewed as an important subject in all of his subsequent work) comes across quite clearly. This theme, exemplified by the term *ghetto*, is clearly a lyrical differentiator between Nas and DOOM, with Nas using the term much more frequently.

Third, Nas’ use of what some might call “offensive language” is a prominent linguistic differentiator. Indeed, *b-tch* and what I’ll term *the n-word* are both among the top words that Nas uses more than DOOM. I’m not the right person to comment on the broader cultural and societal implications of this. I will say that, as someone who listens to a lot of hip-hop music, this type of language is quite often used, almost like a lyrical trope in some cases. Nas’ lyrics are clearly more characteristic of this style than are DOOM’s.

Fourth, we perhaps get a sense of the rappers’ thinking styles. DOOM seems more comfortable with uncertainty, using the term *probably* to indicate likelihood but not certainty (this way of thinking is near-and-dear to an old statistician’s heart). Here is an illustrative example of this use from the Vaudeville Villain album: *If these walls could talk, they’d probably still ignore me*. Nas, in contrast, is more likely to use the term *reason*. Examination of the lyrics in this dataset yields a number of examples of Nas seeking explanations for why things occur (e.g. *Only reason I’m here now is cause God chose me*), suggesting an importance to understanding process and structure in the world. Why are things the way they are? Nas wants to know; DOOM just says - “it’s probably like this.”

Fifth, there are two references to musical qualities. DOOM makes reference to the *beat*; an important musical quality in hip-hop and one for which DOOM (a prominent MC in his own right) is known. Nas, meanwhile, makes references to *singing*. This is an interesting and unconventional (although not entirely inappropriate) way to describe rapping, and suggests an importance placed on the human element of the musical performance in hip-hop (as opposed to the more mechanical or electronically-produced creation of beats). An illustrative example from Nas’ It Was Written album: *They use me wrong so I sing this song*.

One of the advantages of the tidytext approach is that it retains information about word order that is lost when using the bag of words approach. A nice illustration of using word order in quantitative text analysis .This analysis examines the balance of emotion words across the course of each of Jane Austen’s novels. I was inspired by this approach, and the current post is an adaptation of this idea, applied to Pitchfork music reviews.

**Data Preparation**

The head of our raw dataset (called *text\_df*), which serves as the input for our analysis, looks like this:\*

|  | **line** | **text** | **genre** |
| --- | --- | --- | --- |
| 1 | 1 | “Trip-hop” eventually became a ’90s punchline, a music-press shorthand for “overhyped hotel lounge music.”… | electronic |
| 2 | 2 | Eight years, five albums, and two EPs in, the New York-based outfit Krallice have long since shut up purists about their “hipster black metal.” … | metal |
| 3 | 3 | Minneapolis’ Uranium Club seem to revel in being aggressively obtuse… | rock |
| 4 | 4 | Kleenex began with a crash. It transpired one night not long after they’d formed, in Zurich of 1978… | rock |
| 5 | 5 | It is impossible to consider a given release by a footwork artist without confronting the long shadow cast by DJ Rashad’s catalog… | electronic |
| 6 | 6 | Rapper Simbi Ajikawo, who records as Little Simz, is by all measures on an upward trajectory… | rap |

The column “line” serves as an indicator of the review id. The column “text” contains the text of the review and the “genre” column indicates the genre.

Tidytext Vignette

### The Life-Changing Magic of Tidying Text

Using tidy data principles can make many text mining tasks easier, more effective, and consistent with tools already in wide use. Much of the infrastructure needed for text mining with tidy data frames already exists in packages like dplyr, broom, tidyr and ggplot2. In this package, we provide functions and supporting data sets to allow conversion of text to and from tidy formats, and to switch seamlessly between tidy tools and existing text mining packages. Check out our book to learn more about text mining using tidy data principles.

### A few first tidy text mining examples

The novels of Jane Austen can be so tidy! Let’s use the text of Jane Austen’s 6 completed, published novels from the janeaustenr package, and transform them into a tidy format. janeaustenr provides them as a one-row-per-line format:

library(janeaustenr)

library(dplyr)

library(stringr)

original\_books <- austen\_books() %>%

group\_by(book) %>%

mutate(line = row\_number(),

chapter = cumsum(str\_detect(text, regex("^chapter [\\divxlc]",

ignore\_case = TRUE)))) %>%

ungroup()

original\_books

## # A tibble: 73,422 × 4

## text book line chapter

## <chr> <fct> <int> <int>

## 1 "SENSE AND SENSIBILITY" Sense & Sensibility 1 0

## 2 "" Sense & Sensibility 2 0

## 3 "by Jane Austen" Sense & Sensibility 3 0

## 4 "" Sense & Sensibility 4 0

## 5 "(1811)" Sense & Sensibility 5 0

## 6 "" Sense & Sensibility 6 0

## 7 "" Sense & Sensibility 7 0

## 8 "" Sense & Sensibility 8 0

## 9 "" Sense & Sensibility 9 0

## 10 "CHAPTER 1" Sense & Sensibility 10 1

## # … with 73,412 more rows

To work with this as a tidy dataset, we need to restructure it as **one-token-per-row** format. The unnest\_tokens function is a way to convert a dataframe with a text column to be one-token-per-row:

library(tidytext)

tidy\_books <- original\_books %>%

unnest\_tokens(word, text)

tidy\_books

## # A tibble: 725,055 × 4

## book line chapter word

## <fct> <int> <int> <chr>

## 1 Sense & Sensibility 1 0 sense

## 2 Sense & Sensibility 1 0 and

## 3 Sense & Sensibility 1 0 sensibility

## 4 Sense & Sensibility 3 0 by

## 5 Sense & Sensibility 3 0 jane

## 6 Sense & Sensibility 3 0 austen

## 7 Sense & Sensibility 5 0 1811

## 8 Sense & Sensibility 10 1 chapter

## 9 Sense & Sensibility 10 1 1

## 10 Sense & Sensibility 13 1 the

## # … with 725,045 more rows

This function uses the tokenizers package to separate each line into words. The default tokenizing is for words, but other options include characters, ngrams, sentences, lines, paragraphs, or separation around a regex pattern.

Now that the data is in one-word-per-row format, we can manipulate it with tidy tools like dplyr. We can remove stop words (accessible in a tidy form with the function get\_stopwords()) with an anti\_join.

cleaned\_books <- tidy\_books %>%

anti\_join(get\_stopwords())

We can also use count to find the most common words in all the books as a whole.

cleaned\_books %>%

count(word, sort = TRUE)

## # A tibble: 14,375 × 2

## word n

## <chr> <int>

## 1 mr 3015

## 2 mrs 2446

## 3 must 2071

## 4 said 2041

## 5 much 1935

## 6 miss 1855

## 7 one 1831

## 8 well 1523

## 9 every 1456

## 10 think 1440

## # … with 14,365 more rows

Sentiment analysis can be done as an inner join. Sentiment lexicons are available via the get\_sentiments() function. Let’s look at the words with a positive score from the lexicon of Bing Liu and collaborators. What are the most common positive words in Emma?

positive <- get\_sentiments("bing") %>%

filter(sentiment == "positive")

tidy\_books %>%

filter(book == "Emma") %>%

semi\_join(positive) %>%

count(word, sort = TRUE)

## # A tibble: 668 × 2

## word n

## <chr> <int>

## 1 well 401

## 2 good 359

## 3 great 264

## 4 like 200

## 5 better 173

## 6 enough 129

## 7 happy 125

## 8 love 117

## 9 pleasure 115

## 10 right 92

## # … with 658 more rows

Or instead we could examine how sentiment changes during each novel. Let’s find a sentiment score for each word using the same lexicon, then count the number of positive and negative words in defined sections of each novel.

library(tidyr)

bing <- get\_sentiments("bing")

janeaustensentiment <- tidy\_books %>%

inner\_join(bing) %>%

count(book, index = line %/% 80, sentiment) %>%

spread(sentiment, n, fill = 0) %>%

mutate(sentiment = positive - negative)

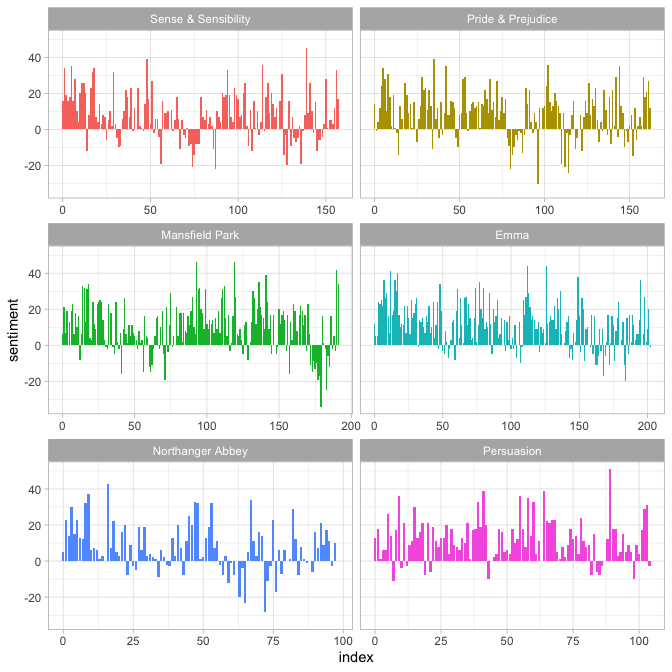
Now we can plot these sentiment scores across the plot trajectory of each novel.

library(ggplot2)

ggplot(janeaustensentiment, aes(index, sentiment, fill = book)) +

geom\_bar(stat = "identity", show.legend = FALSE) +

facet\_wrap(~book, ncol = 2, scales = "free\_x")



### Most common positive and negative words

One advantage of having the data frame with both sentiment and word is that we can analyze word counts that contribute to each sentiment.

bing\_word\_counts <- tidy\_books %>%

inner\_join(bing) %>%

count(word, sentiment, sort = TRUE)

bing\_word\_counts

## # A tibble: 2,585 × 3

## word sentiment n

## <chr> <chr> <int>

## 1 miss negative 1855

## 2 well positive 1523

## 3 good positive 1380

## 4 great positive 981

## 5 like positive 725

## 6 better positive 639

## 7 enough positive 613

## 8 happy positive 534

## 9 love positive 495

## 10 pleasure positive 462

## # … with 2,575 more rows

This can be shown visually, and we can pipe straight into ggplot2 because of the way we are consistently using tools built for handling tidy data frames.

bing\_word\_counts %>%

filter(n > 150) %>%

mutate(n = ifelse(sentiment == "negative", -n, n)) %>%

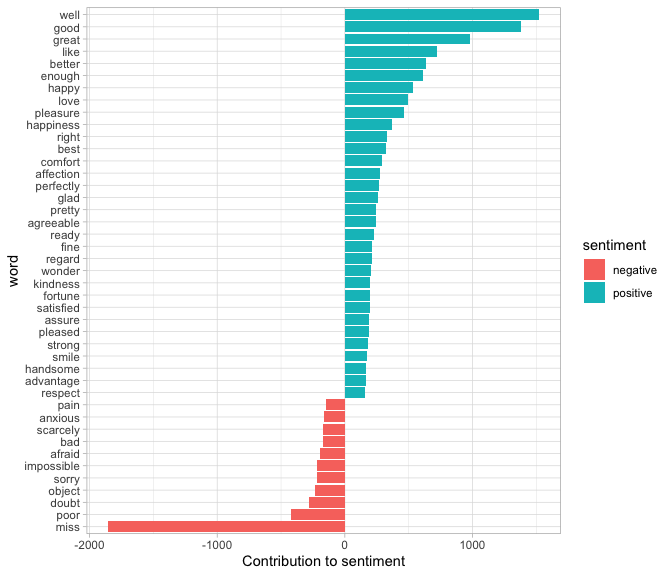
mutate(word = reorder(word, n)) %>%

ggplot(aes(word, n, fill = sentiment)) +

geom\_col() +

coord\_flip() +

labs(y = "Contribution to sentiment")



This lets us spot an anomaly in the sentiment analysis; the word “miss” is coded as negative but it is used as a title for young, unmarried women in Jane Austen’s works. If it were appropriate for our purposes, we could easily add “miss” to a custom stop-words list using bind\_rows.

### Wordclouds

We’ve seen that this tidy text mining approach works well with ggplot2, but having our data in a tidy format is useful for other plots as well.

For example, consider the wordcloud package. Let’s look at the most common words in Jane Austen’s works as a whole again.

library(wordcloud)

cleaned\_books %>%

count(word) %>%

with(wordcloud(word, n, max.words = 100))



In other functions, such as comparison.cloud, you may need to turn it into a matrix with reshape2’s acast. Let’s do the sentiment analysis to tag positive and negative words using an inner join, then find the most common positive and negative words. Until the step where we need to send the data to comparison.cloud, this can all be done with joins, piping, and dplyr because our data is in tidy format.

library(reshape2)

tidy\_books %>%

inner\_join(bing) %>%

count(word, sentiment, sort = TRUE) %>%

acast(word ~ sentiment, value.var = "n", fill = 0) %>%

comparison.cloud(colors = c("#F8766D", "#00BFC4"),

max.words = 100)



### Looking at units beyond just words

Lots of useful work can be done by tokenizing at the word level, but sometimes it is useful or necessary to look at different units of text. For example, some sentiment analysis algorithms look beyond only unigrams (i.e. single words) to try to understand the sentiment of a sentence as a whole. These algorithms try to understand that

I am not having a good day.

is a sad sentence, not a happy one, because of negation. The Stanford CoreNLP tools and the sentimentr R package are examples of such sentiment analysis algorithms. For these, we may want to tokenize text into sentences.

PandP\_sentences <- tibble(text = prideprejudice) %>%

unnest\_tokens(sentence, text, token = "sentences")

Let’s look at just one.

PandP\_sentences$sentence[2]

## [1] "by jane austen"

The sentence tokenizing does seem to have a bit of trouble with UTF-8 encoded text, especially with sections of dialogue; it does much better with punctuation in ASCII.

Another option in unnest\_tokens is to split into tokens using a regex pattern. We could use this, for example, to split the text of Jane Austen’s novels into a data frame by chapter.

austen\_chapters <- austen\_books() %>%

group\_by(book) %>%

unnest\_tokens(chapter, text, token = "regex", pattern = "Chapter|CHAPTER [\\dIVXLC]") %>%

ungroup()

austen\_chapters %>%

group\_by(book) %>%

summarise(chapters = n())

## # A tibble: 6 × 2

## book chapters

## <fct> <int>

## 1 Sense & Sensibility 51

## 2 Pride & Prejudice 62

## 3 Mansfield Park 49

## 4 Emma 56

## 5 Northanger Abbey 32

## 6 Persuasion 25

We have recovered the correct number of chapters in each novel (plus an “extra” row for each novel title). In this data frame, each row corresponds to one chapter.

Near the beginning of this vignette, we used a similar regex to find where all the chapters were in Austen’s novels for a tidy data frame organized by one-word-per-row. We can use tidy text analysis to ask questions such as what are the most negative chapters in each of Jane Austen’s novels? First, let’s get the list of negative words from the Bing lexicon. Second, let’s make a dataframe of how many words are in each chapter so we can normalize for the length of chapters. Then, let’s find the number of negative words in each chapter and divide by the total words in each chapter. Which chapter has the highest proportion of negative words?

bingnegative <- get\_sentiments("bing") %>%

filter(sentiment == "negative")

wordcounts <- tidy\_books %>%

group\_by(book, chapter) %>%

summarize(words = n())

tidy\_books %>%

semi\_join(bingnegative) %>%

group\_by(book, chapter) %>%

summarize(negativewords = n()) %>%

left\_join(wordcounts, by = c("book", "chapter")) %>%

mutate(ratio = negativewords/words) %>%

filter(chapter != 0) %>%

top\_n(1)

## # A tibble: 6 × 5

## # Groups: book [6]

## book chapter negativewords words ratio

## <fct> <int> <int> <int> <dbl>

## 1 Sense & Sensibility 43 161 3405 0.0473

## 2 Pride & Prejudice 34 111 2104 0.0528

## 3 Mansfield Park 46 173 3685 0.0469

## 4 Emma 15 151 3340 0.0452

## 5 Northanger Abbey 21 149 2982 0.0500

## 6 Persuasion 4 62 1807 0.0343

These are the chapters with the most negative words in each book, normalized for number of words in the chapter. What is happening in these chapters? In Chapter 43 of Sense and Sensibility Marianne is seriously ill, near death, and in Chapter 34 of Pride and Prejudice Mr. Darcy proposes for the first time (so badly!). Chapter 46 of Mansfield Park is almost the end, when everyone learns of Henry’s scandalous adultery, Chapter 15 of Emma is when horrifying Mr. Elton proposes, and in Chapter 21 of Northanger Abbey Catherine is deep in her Gothic faux fantasy of murder, etc. Chapter 4 of Persuasion is when the reader gets the full flashback of Anne refusing Captain Wentworth and how sad she was and what a terrible mistake she realized it to be.

We will first turn our raw data into a tidy text dataframe:

# load the packages we'll be using  
library(plyr); library(dplyr)  
library(tidytext)  
library(tidyr)  
library(ggplot2)  
# unnest to one line   
# https://cran.r-project.org/web/packages/tidytext/vignettes/tidytext.html  
tidy\_reviews <- text\_df %>% unnest\_tokens(word, text)

Our data have now been transformed into a tidy format (only first 10 rows shown):

|  | **line** | **genre** | **word** |
| --- | --- | --- | --- |
| 1 | 1 | electronic | trip |
| 2 | 1 | electronic | hop |
| 3 | 1 | electronic | eventually |
| 4 | 1 | electronic | became |
| 5 | 1 | electronic | a |
| 6 | 1 | electronic | 90s |
| 7 | 1 | electronic | punchline |
| 8 | 1 | electronic | a |
| 9 | 1 | electronic | music |
| 10 | 1 | electronic | press |

As shown above, our data now contains one word per line, and our meta-data (review id and genre) are contained in two additional columns. Note that, by default, the *unnest\_tokens* function removes punctuation and converts all letters to lower case as described below:

# **1 The tidy text format**

Using tidy data principles is a powerful way to make handling data easier and more effective, and this is no less true when it comes to dealing with text. As described by Hadley Wickham (Wickham 2014), tidy data has a specific structure:

* Each variable is a column
* Each observation is a row
* Each type of observational unit is a table

We thus define the tidy text format as being **a table with one-token-per-row.** A token is a meaningful unit of text, such as a word, that we are interested in using for analysis, and tokenization is the process of splitting text into tokens. This one-token-per-row structure is in contrast to the ways text is often stored in current analyses, perhaps as strings or in a document-term matrix. For tidy text mining, the **token** that is stored in each row is most often a single word, but can also be an n-gram, sentence, or paragraph. In the tidytext package, we provide functionality to tokenize by commonly used units of text like these and convert to a one-term-per-row format.

Tidy data sets allow manipulation with a standard set of “tidy” tools, including popular packages such as dplyr (Wickham and Francois 2016), tidyr (Wickham 2016), ggplot2 (Wickham 2009), and broom (Robinson 2017). By keeping the input and output in tidy tables, users can transition fluidly between these packages. We’ve found these tidy tools extend naturally to many text analyses and explorations.

At the same time, the tidytext package doesn’t expect a user to keep text data in a tidy form at all times during an analysis. The package includes functions to tidy() objects (see the broom package [Robinson et al cited above]) from popular text mining R packages such as tm (Feinerer, Hornik, and Meyer 2008) and quanteda (Benoit and Nulty 2016). This allows, for example, a workflow where importing, filtering, and processing is done using dplyr and other tidy tools, after which the data is converted into a document-term matrix for machine learning applications. The models can then be re-converted into a tidy form for interpretation and visualization with ggplot2.

## 1.1 Contrasting tidy text with other data structures

As we stated above, we define the tidy text format as being a table with **one-token-per-row.** Structuring text data in this way means that it conforms to tidy data principles and can be manipulated with a set of consistent tools. This is worth contrasting with the ways text is often stored in text mining approaches.

* **String**: Text can, of course, be stored as strings, i.e., character vectors, within R, and often text data is first read into memory in this form.
* **Corpus**: These types of objects typically contain raw strings annotated with additional metadata and details.
* **Document-term matrix**: This is a sparse matrix describing a collection (i.e., a corpus) of documents with one row for each document and one column for each term. The value in the matrix is typically word count or tf-idf (see Chapter 3).

Let’s hold off on exploring corpus and document-term matrix objects until Chapter 5, and get down to the basics of converting text to a tidy format.

## 1.2 The unnest\_tokens function

Emily Dickinson wrote some lovely text in her time.

text <- c("Because I could not stop for Death -",

"He kindly stopped for me -",

"The Carriage held but just Ourselves -",

"and Immortality")

text

#> [1] "Because I could not stop for Death -"

#> [2] "He kindly stopped for me -"

#> [3] "The Carriage held but just Ourselves -"

#> [4] "and Immortality"

This is a typical character vector that we might want to analyze. In order to turn it into a tidy text dataset, we first need to put it into a data frame.

library(dplyr)

text\_df <- tibble(line = 1:4, text = text)

text\_df

#> # A tibble: 4 × 2

#> line text

#> <int> <chr>

#> 1 1 Because I could not stop for Death -

#> 2 2 He kindly stopped for me -

#> 3 3 The Carriage held but just Ourselves -

#> 4 4 and Immortality

What does it mean that this data frame has printed out as a “tibble”? A tibble is a modern class of data frame within R, available in the dplyr and tibble packages, that has a convenient print method, will not convert strings to factors, and does not use row names. Tibbles are great for use with tidy tools.

Notice that this data frame containing text isn’t yet compatible with tidy text analysis, though. We can’t filter out words or count which occur most frequently, since each row is made up of multiple combined words. We need to convert this so that it has **one-token-per-document-per-row**.

A token is a meaningful unit of text, most often a word, that we are interested in using for further analysis, and tokenization is the process of splitting text into tokens.

In this first example, we only have one document (the poem), but we will explore examples with multiple documents soon.

Within our tidy text framework, we need to both break the text into individual tokens (a process called tokenization) and transform it to a tidy data structure. To do this, we use tidytext’s unnest\_tokens() function.

library(tidytext)

text\_df %>%

unnest\_tokens(word, text)

#> # A tibble: 20 × 2

#> line word

#> <int> <chr>

#> 1 1 because

#> 2 1 i

#> 3 1 could

#> 4 1 not

#> 5 1 stop

#> 6 1 for

#> 7 1 death

#> 8 2 he

#> 9 2 kindly

#> 10 2 stopped

#> # … with 10 more rows

The two basic arguments to unnest\_tokens used here are column names. First we have the output column name that will be created as the text is unnested into it (word, in this case), and then the input column that the text comes from (text, in this case). Remember that text\_df above has a column called text that contains the data of interest.

After using unnest\_tokens, we’ve split each row so that there is one token (word) in each row of the new data frame; the default tokenization in unnest\_tokens() is for single words, as shown here. Also notice:

* Other columns, such as the line number each word came from, are retained.
* Punctuation has been stripped.
* By default, unnest\_tokens() converts the tokens to lowercase, which makes them easier to compare or combine with other datasets. (Use the to\_lower = FALSE argument to turn off this behavior).

Having the text data in this format lets us manipulate, process, and visualize the text using the standard set of tidy tools, namely dplyr, tidyr, and ggplot2, as shown in Figure 1.1.

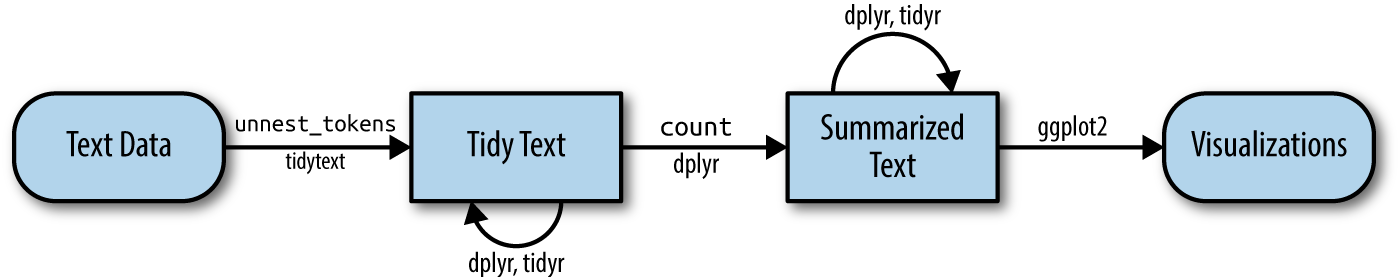


Figure 1.1: A flowchart of a typical text analysis using tidy data principles. This chapter shows how to summarize and visualize text using these tools.

## 1.3 Tidying the works

Let’s use the text of Jane Austen’s 6 completed, published novels from the janeaustenr package (Silge 2016), and transform them into a tidy format. The janeaustenr package provides these texts in a one-row-per-line format, where a line in this context is analogous to a literal printed line in a physical book. Let’s start with that, and also use mutate() to annotate a linenumber quantity to keep track of lines in the original format and a chapter (using a regex) to find where all the chapters are.

library(janeaustenr)

library(dplyr)

library(stringr)

original\_books <- austen\_books() %>%

group\_by(book) %>%

mutate(linenumber = row\_number(),

chapter = cumsum(str\_detect(text,

regex("^chapter [\\divxlc]",

ignore\_case = TRUE)))) %>%

ungroup()

original\_books

#> # A tibble: 73,422 × 4

#> text book linenumber chapter

#> <chr> <fct> <int> <int>

#> 1 "SENSE AND SENSIBILITY" Sense & Sensibility 1 0

#> 2 "" Sense & Sensibility 2 0

#> 3 "by Jane Austen" Sense & Sensibility 3 0

#> 4 "" Sense & Sensibility 4 0

#> 5 "(1811)" Sense & Sensibility 5 0

#> 6 "" Sense & Sensibility 6 0

#> 7 "" Sense & Sensibility 7 0

#> 8 "" Sense & Sensibility 8 0

#> 9 "" Sense & Sensibility 9 0

#> 10 "CHAPTER 1" Sense & Sensibility 10 1

#> # … with 73,412 more rows

To work with this as a tidy dataset, we need to restructure it in the **one-token-per-row** format, which as we saw earlier is done with the unnest\_tokens() function.

library(tidytext)

tidy\_books <- original\_books %>%

unnest\_tokens(word, text)

tidy\_books

#> # A tibble: 725,055 × 4

#> book linenumber chapter word

#> <fct> <int> <int> <chr>

#> 1 Sense & Sensibility 1 0 sense

#> 2 Sense & Sensibility 1 0 and

#> 3 Sense & Sensibility 1 0 sensibility

#> 4 Sense & Sensibility 3 0 by

#> 5 Sense & Sensibility 3 0 jane

#> 6 Sense & Sensibility 3 0 austen

#> 7 Sense & Sensibility 5 0 1811

#> 8 Sense & Sensibility 10 1 chapter

#> 9 Sense & Sensibility 10 1 1

#> 10 Sense & Sensibility 13 1 the

#> # … with 725,045 more rows

This function uses the tokenizers package to separate each line of text in the original data frame into tokens. The default tokenizing is for words, but other options include characters, n-grams, sentences, lines, paragraphs, or separation around a regex pattern.

Now that the data is in one-word-per-row format, we can manipulate it with tidy tools like dplyr. Often in text analysis, we will want to remove stop words; stop words are words that are not useful for an analysis, typically extremely common words such as “the”, “of”, “to”, and so forth in English. We can remove stop words (kept in the tidytext dataset stop\_words) with an anti\_join().

data(stop\_words)

tidy\_books <- tidy\_books %>%

anti\_join(stop\_words)

The stop\_words dataset in the tidytext package contains stop words from three lexicons. We can use them all together, as we have here, or filter() to only use one set of stop words if that is more appropriate for a certain analysis.

We can also use dplyr’s count() to find the most common words in all the books as a whole.

tidy\_books %>%

count(word, sort = TRUE)

#> # A tibble: 13,914 × 2

#> word n

#> <chr> <int>

#> 1 miss 1855

#> 2 time 1337

#> 3 fanny 862

#> 4 dear 822

#> 5 lady 817

#> 6 sir 806

#> 7 day 797

#> 8 emma 787

#> 9 sister 727

#> 10 house 699

#> # … with 13,904 more rows

Because we’ve been using tidy tools, our word counts are stored in a tidy data frame. This allows us to pipe this directly to the ggplot2 package, for example to create a visualization of the most common words (Figure 1.2).

library(ggplot2)

tidy\_books %>%

count(word, sort = TRUE) %>%

filter(n > 600) %>%

mutate(word = reorder(word, n)) %>%

ggplot(aes(n, word)) +

geom\_col() +

labs(y = NULL)

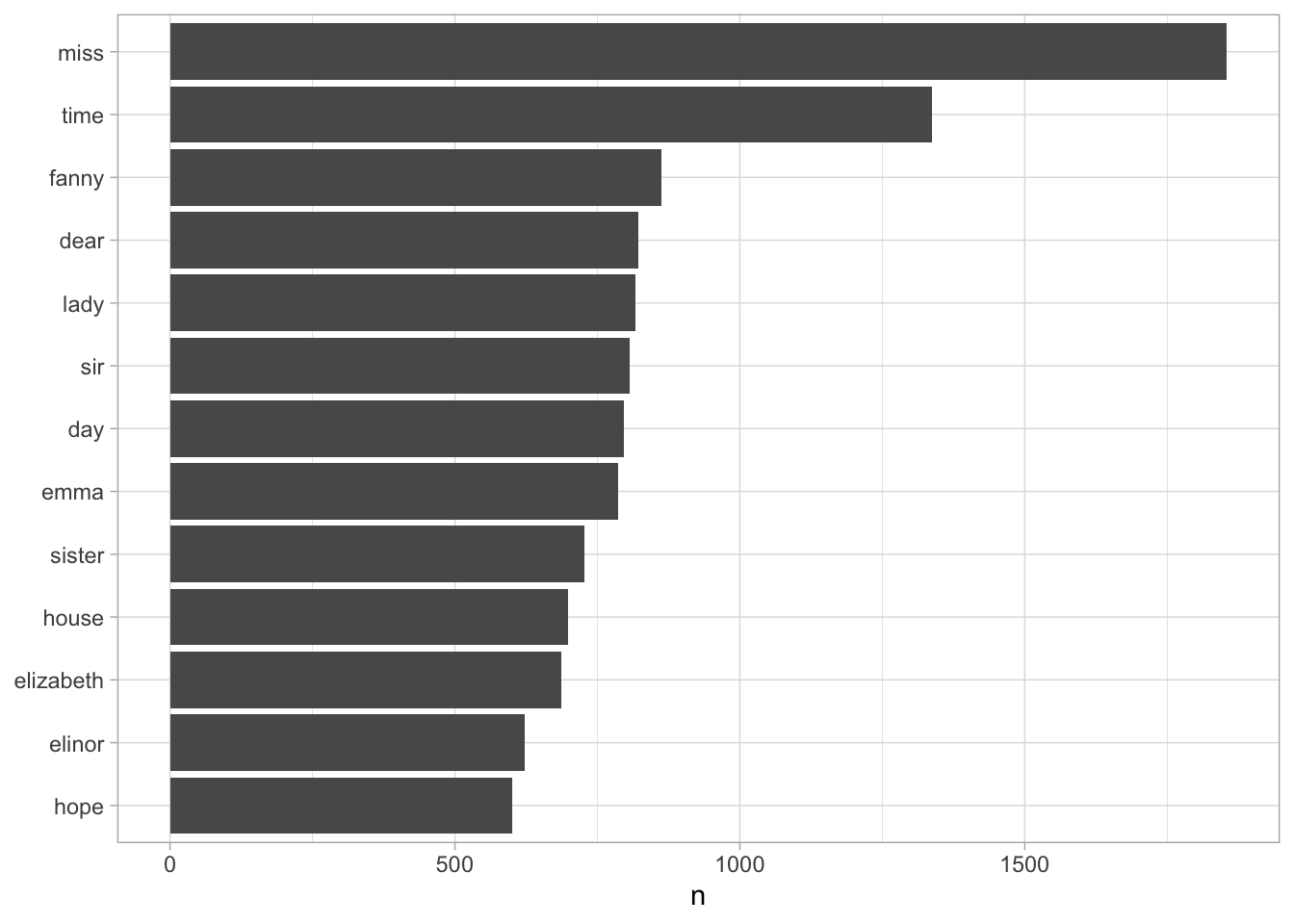


Figure 1.2: The most common words in Jane Austen’s novels

Note that the austen\_books() function started us with exactly the text we wanted to analyze, but in other cases we may need to perform cleaning of text data, such as removing copyright headers or formatting. You’ll see examples of this kind of pre-processing in the case study chapters, particularly Chapter 9.1.1.

## 1.4 The gutenbergr package

Now that we’ve used the janeaustenr package to explore tidying text, let’s introduce the gutenbergr package (Robinson 2016). The gutenbergr package provides access to the public domain works from the Project Gutenberg collection. The package includes tools both for downloading books (stripping out the unhelpful header/footer information), and a complete dataset of Project Gutenberg metadata that can be used to find works of interest. In this book, we will mostly use the function gutenberg\_download() that downloads one or more works from Project Gutenberg by ID, but you can also use other functions to explore metadata, pair Gutenberg ID with title, author, language, etc., or gather information about authors.

To learn more about gutenbergr, check out the package’s documentation at rOpenSci, where it is one of rOpenSci’s packages for data access.

## 1.5 Word frequencies

A common task in text mining is to look at word frequencies, just like we have done above for Jane Austen’s novels, and to compare frequencies across different texts. We can do this intuitively and smoothly using tidy data principles. We already have Jane Austen’s works; let’s get two more sets of texts to compare to. First, let’s look at some science fiction and fantasy novels by H.G. Wells, who lived in the late 19th and early 20th centuries. Let’s get The Time Machine, The War of the Worlds, The Invisible Man, and The Island of Doctor Moreau. We can access these works using gutenberg\_download() and the Project Gutenberg ID numbers for each novel.

library(gutenbergr)

hgwells <- gutenberg\_download(c(35, 36, 5230, 159))

tidy\_hgwells <- hgwells %>%

unnest\_tokens(word, text) %>%

anti\_join(stop\_words)

Just for kicks, what are the most common words in these novels of H.G. Wells?

tidy\_hgwells %>%

count(word, sort = TRUE)

#> # A tibble: 11,769 × 2

#> word n

#> <chr> <int>

#> 1 time 454

#> 2 people 302

#> 3 door 260

#> 4 heard 249

#> 5 black 232

#> 6 stood 229

#> 7 white 222

#> 8 hand 218

#> 9 kemp 213

#> 10 eyes 210

#> # … with 11,759 more rows

Now let’s get some well-known works of the Brontë sisters, whose lives overlapped with Jane Austen’s somewhat but who wrote in a rather different style. Let’s get Jane Eyre, Wuthering Heights, The Tenant of Wildfell Hall, Villette, and Agnes Grey. We will again use the Project Gutenberg ID numbers for each novel and access the texts using gutenberg\_download().

bronte <- gutenberg\_download(c(1260, 768, 969, 9182, 767))

tidy\_bronte <- bronte %>%

unnest\_tokens(word, text) %>%

anti\_join(stop\_words)

What are the most common words in these novels of the Brontë sisters?

tidy\_bronte %>%

count(word, sort = TRUE)

#> # A tibble: 23,051 × 2

#> word n

#> <chr> <int>

#> 1 time 1065

#> 2 miss 855

#> 3 day 827

#> 4 hand 768

#> 5 eyes 713

#> 6 night 647

#> 7 heart 638

#> 8 looked 602

#> 9 door 592

#> 10 half 586

#> # … with 23,041 more rows

Interesting that “time”, “eyes”, and “hand” are in the top 10 for both H.G. Wells and the Brontë sisters.

Now, let’s calculate the frequency for each word for the works of Jane Austen, the Brontë sisters, and H.G. Wells by binding the data frames together. We can use pivot\_wider() and pivot\_longer() from tidyr to reshape our dataframe so that it is just what we need for plotting and comparing the three sets of novels.

library(tidyr)

frequency <- bind\_rows(mutate(tidy\_bronte, author = "Brontë Sisters"),

mutate(tidy\_hgwells, author = "H.G. Wells"),

mutate(tidy\_books, author = "Jane Austen")) %>%

mutate(word = str\_extract(word, "[a-z']+")) %>%

count(author, word) %>%

group\_by(author) %>%

mutate(proportion = n / sum(n)) %>%

select(-n) %>%

pivot\_wider(names\_from = author, values\_from = proportion) %>%

pivot\_longer(`Brontë Sisters`:`H.G. Wells`,

names\_to = "author", values\_to = "proportion")

frequency

#> # A tibble: 57,820 × 4

#> word `Jane Austen` author proportion

#> <chr> <dbl> <chr> <dbl>

#> 1 a 0.00000919 Brontë Sisters 0.0000319

#> 2 a 0.00000919 H.G. Wells 0.0000150

#> 3 a'most NA Brontë Sisters 0.0000159

#> 4 a'most NA H.G. Wells NA

#> 5 aback NA Brontë Sisters 0.00000398

#> 6 aback NA H.G. Wells 0.0000150

#> 7 abaht NA Brontë Sisters 0.00000398

#> 8 abaht NA H.G. Wells NA

#> 9 abandon NA Brontë Sisters 0.0000319

#> 10 abandon NA H.G. Wells 0.0000150

#> # … with 57,810 more rows

We use str\_extract() here because the UTF-8 encoded texts from Project Gutenberg have some examples of words with underscores around them to indicate emphasis (like italics). The tokenizer treated these as words, but we don’t want to count “\_any\_” separately from “any” as we saw in our initial data exploration before choosing to use str\_extract().

Now let’s plot (Figure 1.3).

library(scales)

# expect a warning about rows with missing values being removed

ggplot(frequency, aes(x = proportion, y = `Jane Austen`,

color = abs(`Jane Austen` - proportion))) +

geom\_abline(color = "gray40", lty = 2) +

geom\_jitter(alpha = 0.1, size = 2.5, width = 0.3, height = 0.3) +

geom\_text(aes(label = word), check\_overlap = TRUE, vjust = 1.5) +

scale\_x\_log10(labels = percent\_format()) +

scale\_y\_log10(labels = percent\_format()) +

scale\_color\_gradient(limits = c(0, 0.001),

low = "darkslategray4", high = "gray75") +

facet\_wrap(~author, ncol = 2) +

theme(legend.position="none") +

labs(y = "Jane Austen", x = NULL)



Figure 1.3: Comparing the word frequencies of Jane Austen, the Brontë sisters, and H.G. Wells

Words that are close to the line in these plots have similar frequencies in both sets of texts, for example, in both Austen and Brontë texts (“miss”, “time”, “day” at the upper frequency end) or in both Austen and Wells texts (“time”, “day”, “brother” at the high frequency end). Words that are far from the line are words that are found more in one set of texts than another. For example, in the Austen-Brontë panel, words like “elizabeth”, “emma”, and “fanny” (all proper nouns) are found in Austen’s texts but not much in the Brontë texts, while words like “arthur” and “dog” are found in the Brontë texts but not the Austen texts. In comparing H.G. Wells with Jane Austen, Wells uses words like “beast”, “guns”, “feet”, and “black” that Austen does not, while Austen uses words like “family”, “friend”, “letter”, and “dear” that Wells does not.

Overall, notice in Figure 1.3 that the words in the Austen-Brontë panel are closer to the zero-slope line than in the Austen-Wells panel. Also notice that the words extend to lower frequencies in the Austen-Brontë panel; there is empty space in the Austen-Wells panel at low frequency. These characteristics indicate that Austen and the Brontë sisters use more similar words than Austen and H.G. Wells. Also, we see that not all the words are found in all three sets of texts and there are fewer data points in the panel for Austen and H.G. Wells.

Let’s quantify how similar and different these sets of word frequencies are using a correlation test. How correlated are the word frequencies between Austen and the Brontë sisters, and between Austen and Wells?

cor.test(data = frequency[frequency$author == "Brontë Sisters",],

~ proportion + `Jane Austen`)

#>

#> Pearson's product-moment correlation

#>

#> data: proportion and Jane Austen

#> t = 119.64, df = 10404, p-value < 2.2e-16

#> alternative hypothesis: true correlation is not equal to 0

#> 95 percent confidence interval:

#> 0.7527837 0.7689611

#> sample estimates:

#> cor

#> 0.7609907

cor.test(data = frequency[frequency$author == "H.G. Wells",],

~ proportion + `Jane Austen`)

#>

#> Pearson's product-moment correlation

#>

#> data: proportion and Jane Austen

#> t = 36.441, df = 6053, p-value < 2.2e-16

#> alternative hypothesis: true correlation is not equal to 0

#> 95 percent confidence interval:

#> 0.4032820 0.4446006

#> sample estimates:

#> cor

#> 0.424162

Unnesting our text data gives us a narrow but extremely long dataframe. Specifically, our dataframe contains as many rows as there are words in the 12,147 reviews: 8,182,882 to be precise!

As we are interested in understanding the course of emotional valence throughout the texts, which I will call “*position\_in\_review\_0*.”

# add position within review text  
tidy\_reviews <- tidy\_reviews %>% group\_by(line) %>%   
 mutate(position\_in\_review\_0 = 1:n())

Our dataset, called *tidy\_reviews*, now looks like this (only first 10 rows shown):

|  | **line** | **genre** | **word** | **position\_in\_review\_0** |
| --- | --- | --- | --- | --- |
| 1 | 1 | electronic | trip | 1 |
| 2 | 1 | electronic | hop | 2 |
| 3 | 1 | electronic | eventually | 3 |
| 4 | 1 | electronic | became | 4 |
| 5 | 1 | electronic | a | 5 |
| 6 | 1 | electronic | 90s | 6 |
| 7 | 1 | electronic | punchline | 7 |
| 8 | 1 | electronic | a | 8 |
| 9 | 1 | electronic | music | 9 |
| 10 | 1 | electronic | press | 10 |

We can see that the data contain stopwords (words which occur frequently but contain no meaningful content such as “the”, “a” etc.). Before we continue, let’s remove these stopwords. In the tidytext approach, this is done with an anti-join of our tidy text dataframe against a tidy dataframe containing a list of stopwords. We then order the dataset by the review id (called *line*) and by the word order position (*position\_in\_review\_0*) in the raw data. We create a new word order variable, called *position\_in\_review*, which gives the position of each word in the cleaned data (without stopwords), while removing the *position\_in\_review\_0* variable created above.

# remove stop words  
# order the dataset by review id  
# and the order of the remaining words  
# in the original dataset  
cleaned\_reviews <- tidy\_reviews %>%  
 anti\_join(stop\_words) %>% arrange(line, position\_in\_review\_0)  
  
# add position of word within cleaned review  
# we also remove the first word order column  
# (position\_in\_review\_0) created above  
cleaned\_reviews <- cleaned\_reviews %>% group\_by(line) %>%   
 mutate(position\_in\_review = 1:n()) %>% select(-position\_in\_review\_0)

The data (called *cleaned\_reviews*) now look like this (only first 10 rows shown):

|  | **line** | **genre** | **word** | **position\_in\_review** |
| --- | --- | --- | --- | --- |
| 1 | 1 | electronic | trip | 1 |
| 2 | 1 | electronic | hop | 2 |
| 3 | 1 | electronic | eventually | 3 |
| 4 | 1 | electronic | 90s | 4 |
| 5 | 1 | electronic | punchline | 5 |
| 6 | 1 | electronic | music | 6 |
| 7 | 1 | electronic | press | 7 |
| 8 | 1 | electronic | shorthand | 8 |
| 9 | 1 | electronic | overhyped | 9 |
| 10 | 1 | electronic | hotel | 10 |

Our goal was to preserve the word order so that we can track the use of emotion words across the course of the review texts. The steps necessary to achieve this were somewhat involved, but we have reached our goal. We have retained the important words in the texts and, for each word, we have created a record of its position in the review.

As the reviews have differing numbers of words, we cannot simply compare the evolution of sentiment use across word number. Therefore, we will count the number of words in each review, and for each word, calculate its position in terms of its percentage in the words of the text. This will result in 101 different levels representing word position for each text (because we go from 0 to 1 in increments of .01). We will eventually aggregate the data to this level, making it possible to visualize the use of emotion words across the different percentages of the review texts.

# count the number of remaining words for each review  
# and calculate the percentage within each review  
# that each word falls in  
cleaned\_reviews <- cleaned\_reviews %>% group\_by(line) %>%   
 mutate(wordcount\_review = n(),   
 percentage\_in\_review = round(position\_in\_review/wordcount\_review,2))

Our data now look like this (only first 10 rows shown):

|  | **line** | **genre** | **word** | **position\_in\_review** | **wordcount\_review** | **percentage\_in\_review** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | electronic | trip | 1 | 746 | 0 |
| 2 | 1 | electronic | hop | 2 | 746 | 0 |
| 3 | 1 | electronic | eventually | 3 | 746 | 0 |
| 4 | 1 | electronic | 90s | 4 | 746 | 0.01 |
| 5 | 1 | electronic | punchline | 5 | 746 | 0.01 |
| 6 | 1 | electronic | music | 6 | 746 | 0.01 |
| 7 | 1 | electronic | press | 7 | 746 | 0.01 |
| 8 | 1 | electronic | shorthand | 8 | 746 | 0.01 |
| 9 | 1 | electronic | overhyped | 9 | 746 | 0.01 |
| 10 | 1 | electronic | hotel | 10 | 746 | 0.01 |

*Intermezzo: Coding Sentiment With Dictionaries*

There are a number of different ways of analyzing sentiment in text. One common approach is to use dictionaries, which contain pre-defined lists of words which are categorized as belonging to a particular type of higher-level characteristic we wish to understand (e.g. in the case of sentiment – positive, negative, or more fine-grained such as excitement, anxiety, etc.).

In this post, we will use a dictionary set included in the tidytext package and which is described in the tidytext book and vignette. Specifically, we will use the positive and negative sentiment words contained in the “bing” dictionaries. For more information about the bing dictionaries, consult the tidytext book or vignette, or type *?sentiments* in the R console (with the tidytext package loaded). One thing to keep in mind is that the bing dictionaries contain many more negative words than positive words:

# count of positive/negative sentiment  
# in bing dictionaries  
get\_sentiments("bing") %>%   
 count(sentiment)

Which returns:

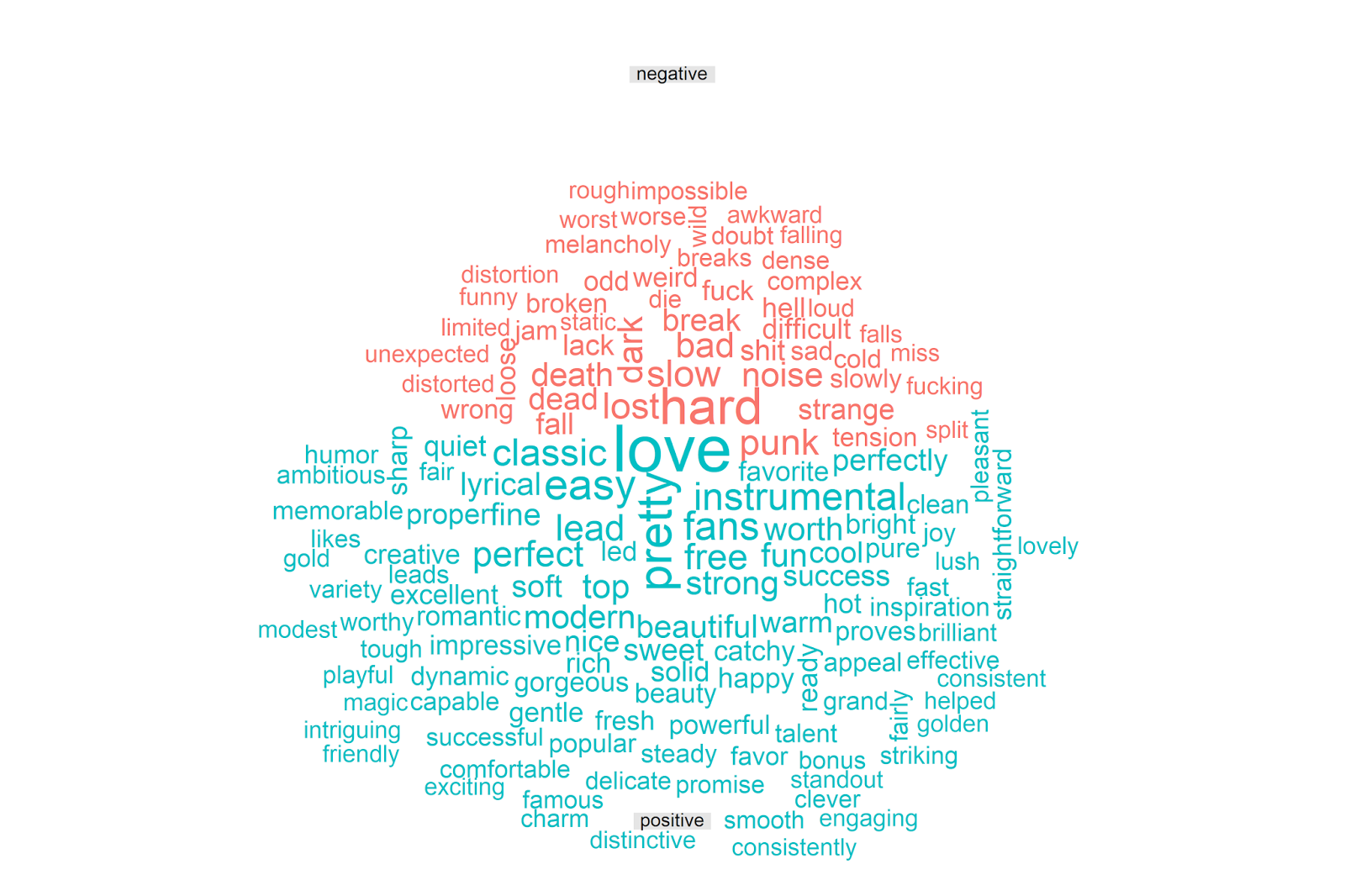
|  | **sentiment** | **n** |
| --- | --- | --- |
| 1 | negative | 4782 |
| 2 | positive | 2006 |

Indeed, there are more than twice as many negative than positive words in the bing dictionaries.

Let’s look at the most frequent positive and negative words from the bing dictionaries in our review data. We can make a word cloud with code directly adapted from the tidytext vignette to do this:

# plot most frequent positive/negative words  
# with wordcloud  
library(reshape2)  
library(wordcloud)  
  
cleaned\_reviews %>%  
 inner\_join(get\_sentiments("bing"), by = "word") %>%   
 count(word, sentiment, sort = **TRUE**) %>%   
 acast(word ~ sentiment, value.var = "n", fill = 0, fun.aggregate = length) %>%   
 comparison.cloud(colors = c("#F8766D", "#00BFC4"),  
 max.words = 150, title.size = 1, scale=c(3.5,1))

Which gives us the following plot:

[](https://i2.wp.com/3.bp.blogspot.com/-NglqNf5yEmw/Wmzt7ycOoUI/AAAAAAAAAZw/gKREAkemKCkShw2dzPH8-cGrSZ-Ja4SxQCLcBGAs/s1600/comp_cloud_pos_neg.png?ssl=1)

The vast majority of the words make sense to me, and seem to capture positive and negative things one could say about music in the context of an album review. It is amusing to see that “punk” is classified as negative, which makes sense in many contexts but is a bit off here. This is the downside of using general-purpose text classification dictionaries; overall they can perform quite well but they are by design not adapted for the specifics of every corpus. Despite these small imperfections, the above visualization makes clear that the bing dictionaries are picking up on meaningful indicators of sentiment in the Pitchfork reviews.

Note that, in the analysis below, I will treat positive and negative sentiment separately. The analysis presented in the tidytext vignette analyzes an overall sentiment score (e.g. sentiment = positive – negative). However, this seems strange to me for 2 reasons. First, as we saw above, there are twice as many negative (vs. positive) words in the bing corpus. Second, there is a large literature in psychology that suggests that positive and negative affect are orthogonal (independent); this logic underpins the measurement of affect in the widely used [PANAS questionnaire](https://en.wikipedia.org/wiki/Positive_and_Negative_Affect_Schedule), for example.

Let’s then apply these dictionaries to our data, in order to extract both positive and negative sentiment at each percentage of our review texts.

*Finishing the Munging*

We will conduct all of the necessary steps in a single dplyr chain. First, we merge in the sentiment dictionaries, retaining only the words classified as positive or negative. We then count the number of positive and negative words at each percentage in the reviews; this counting is done separately for each percentage of each review. We then aggregate the data by genre and percentage of the review text. Specifically, for each genre, we calculate the average number of positive and negative words that occur at each percentage in the review text.

pitchfork\_sentiment <- cleaned\_reviews %>%  
 # code the words according to the positive/negative bing sentiment dictionaries  
 inner\_join(get\_sentiments("bing"), by = "word") %>%  
 # count the number of pos/neg words at each percentage of each review  
 count(genre, index = percentage\_in\_review, sentiment) %>%  
 # put the pos/neg counts into their own columns  
 spread(sentiment, n, fill = 0) %>%  
 # for each genre, compute the average number of pos/neg words  
 # used at each percentage of the review texts  
 group\_by(genre, index) %>%  
 summarize(mean\_negative = round(mean(negative, na.rm = **TRUE**),2),  
 mean\_positive = round(mean(positive, na.rm = **TRUE**),2))

The resulting dataset, called *pitchfork\_sentiment*, looks like this (only first 10 rows shown):

|  | **genre** | **index** | **mean\_negative** | **mean\_positive** |
| --- | --- | --- | --- | --- |
| 1 | electronic | 0 | 0.67 | 0.35 |
| 2 | electronic | 0.01 | 0.61 | 0.54 |
| 3 | electronic | 0.02 | 0.61 | 0.59 |
| 4 | electronic | 0.03 | 0.62 | 0.54 |
| 5 | electronic | 0.04 | 0.64 | 0.53 |
| 6 | electronic | 0.05 | 0.56 | 0.59 |
| 7 | electronic | 0.06 | 0.62 | 0.55 |
| 8 | electronic | 0.07 | 0.64 | 0.56 |
| 9 | electronic | 0.08 | 0.63 | 0.56 |
| 10 | electronic | 0.09 | 0.58 | 0.58 |

Our data contains 9 genres, with 101 rows per genre (because we go from 0 to 1 in increments of .01), resulting in 909 total rows.

**Visualizing Emotional Valence Across the Album Reviews**

We will produce our plots using the excellent **ggplot2** package, which also is built according to the tidy data philosophy. From the tidy perspective, our data are problematic in that the values we want to visualize (*mean\_positive* and *mean\_negative*) are contained in two different columns. Therefore, in order to plot using ggplot2, we must transform our data from a wide to a long format, putting our observations in a single column, with an additional column containing the sentiment type (positive or negative).

We can achieve this using code taken directly from the *gather* help page from the tidyr package

# make the wide to long data  
# (from the "gather" help page in tidyr)  
long\_sentiment\_by\_genre <- gather(pitchfork\_sentiment, key,  
 value, -genre, -index)

Which gives us (only first 10 rows shown):

|  | **genre** | **index** | **key** | **value** |
| --- | --- | --- | --- | --- |
| 1 | electronic | 0 | mean\_negative | 0.67 |
| 2 | electronic | 0.01 | mean\_negative | 0.61 |
| 3 | electronic | 0.02 | mean\_negative | 0.61 |
| 4 | electronic | 0.03 | mean\_negative | 0.62 |
| 5 | electronic | 0.04 | mean\_negative | 0.64 |
| 6 | electronic | 0.05 | mean\_negative | 0.56 |
| 7 | electronic | 0.06 | mean\_negative | 0.62 |
| 8 | electronic | 0.07 | mean\_negative | 0.64 |
| 9 | electronic | 0.08 | mean\_negative | 0.63 |
| 10 | electronic | 0.09 | mean\_negative | 0.58 |

Our numeric data on both positive and negative sentiment are now contained in a single column (called *value*) while the sentiment type (called *key*) is contained in a separate column.

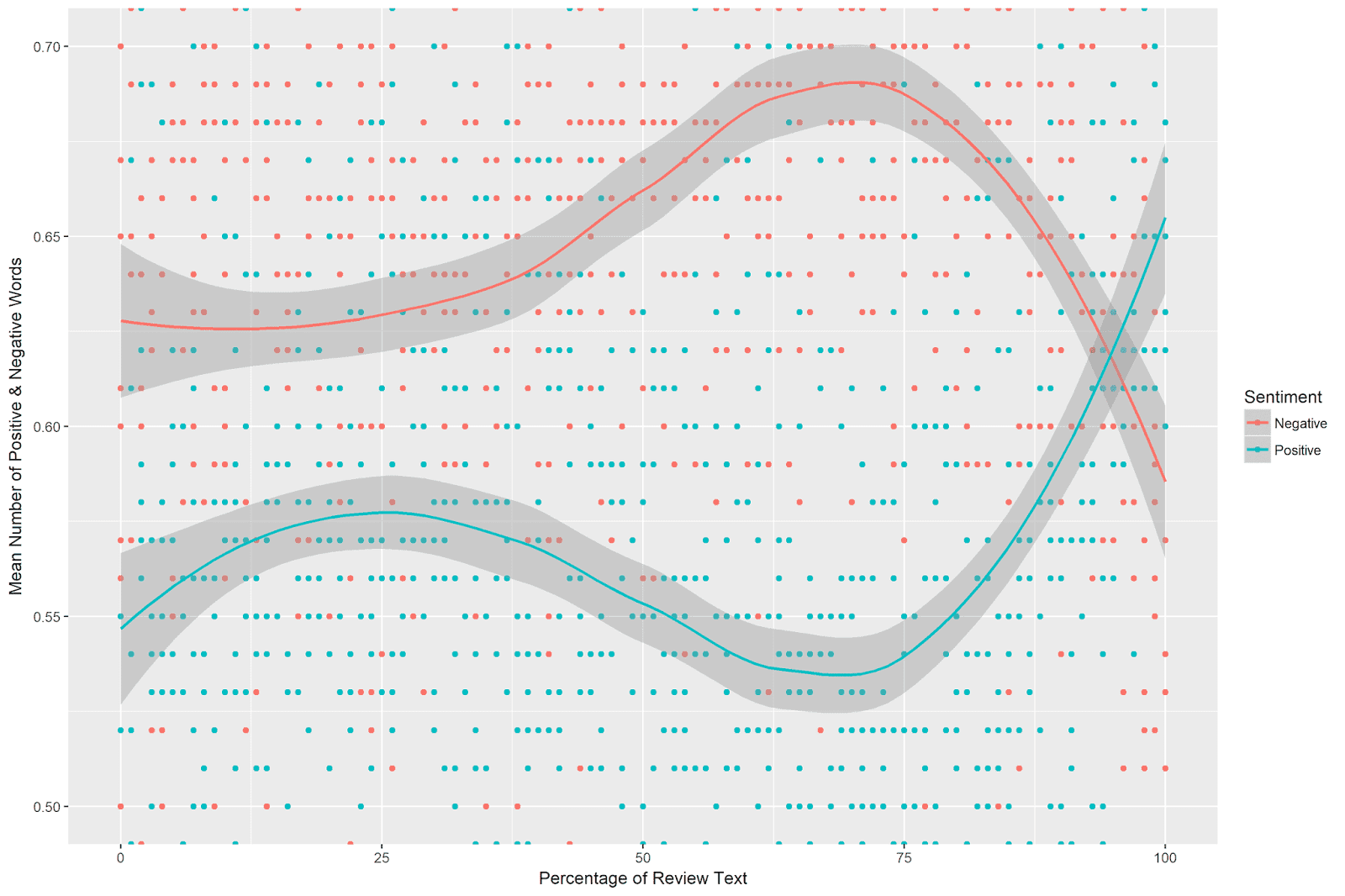
Our input dataframe (*pitchfork\_sentiment*) had 909 rows, and because each row had a value for both positive and negative sentiment, our long dataframe (called *long\_sentiment\_by\_genre*) has 1,818 rows.

*Overall Trends*

We are now ready to plot the average number of positive and negative sentiment words across the course of the reviews. We’ll first produce a plot of the overall data (not split by genre), using [loess regression](https://en.wikipedia.org/wiki/Local_regression) to visualize the overall trends per sentiment type. Note that I set the limits of the y axis to focus on the loess regression lines:

# plot the positive vs. negative sentiment across review percentage  
ggplot(long\_sentiment\_by\_genre, aes(index \* 100, value, color = key)) +  
 geom\_point() +  
 geom\_smooth(method="loess") +   
 coord\_cartesian(ylim = c(.5, .7)) +  
 labs(x = "Percentage of Review Text",   
 y = "Mean Number of Positive & Negative Words" ) +  
 scale\_color\_manual(name="Sentiment",  
 breaks=c("mean\_negative", "mean\_positive"),  
 labels=c("Negative", "Positive"),  
 values = c("#F8766D", "#00BFC4"))

Which produces the following plot:

[](https://i1.wp.com/4.bp.blogspot.com/-XDGO8DCZzB0/WnIhvLU3haI/AAAAAAAAAaA/eYRLQ9r_7KAzTG13jLES4r-1isFJMOsJACLcBGAs/s1600/overall_pos_neg.png?ssl=1)

This is quite interesting. There is a clear pattern of positive and negative sentiment use across the album reviews. Negative sentiment use is flat for around the first 45 percent of the review text, after which it increases, peaking just shy of the 75th percentile of the reviews. After peaking, the use of negative sentiment plummets sharply, ending lower than its starting point.

Positive sentiment, meanwhile, increases slightly at around 1/4 of the review text. It then decreases, reaching a low point at around 70%. From around the 75th percentile of the review texts, positive sentiment use increases sharply and ends far above its starting point.

When comparing the trends of positive and negative sentiment, there is a clear divergence just short of the 75th percentile of the album review texts. At this point, negative sentiment increases, while positive sentiment decreases. After the 75th percentile, these trends reverse, and the review texts end with more positive than negative sentiment.

*Trends by Genre*

We can also examine the course of positive and negative sentiment across the reviews for the different genres. This requires just a slight modification of the above code to use genre as a facet (note I again specify the range of the y-axis in the plot to highlight the trends):

# separate plots per genre  
ggplot(long\_sentiment\_by\_genre, aes(index, value, color = key)) +  
 geom\_point() +  
 geom\_smooth(method="loess") +   
 coord\_cartesian(ylim = c(.45, .8)) +  
 labs(x = "Percentage of Review Text", y = "Mean Count Positive & Negative Sentiment" ) +  
 scale\_color\_manual(name="Emotional\nValence",  
 breaks=c("mean\_negative", "mean\_positive"),  
 labels=c("Negative", "Positive"),  
 values = c("#F8766D", "#00BFC4")) +  
 facet\_wrap(~genre, ncol = 3, scales = "free\_x")

Which yields the following plot:

Ggplot – Facet

facet\_grid() forms a matrix of panels defined by row and column faceting variables. It is most useful when you have two discrete variables, and all combinations of the variables exist in the data. If you have only one variable with many levels, try [facet\_wrap()](https://ggplot2.tidyverse.org/reference/facet_wrap.html).

facet\_grid(

rows = NULL,

cols = NULL,

scales = "fixed",

space = "fixed",

shrink = TRUE,

labeller = "label\_value",

as.table = TRUE,

switch = NULL,

drop = TRUE,

margins = FALSE,

facets = NULL

)

## Arguments

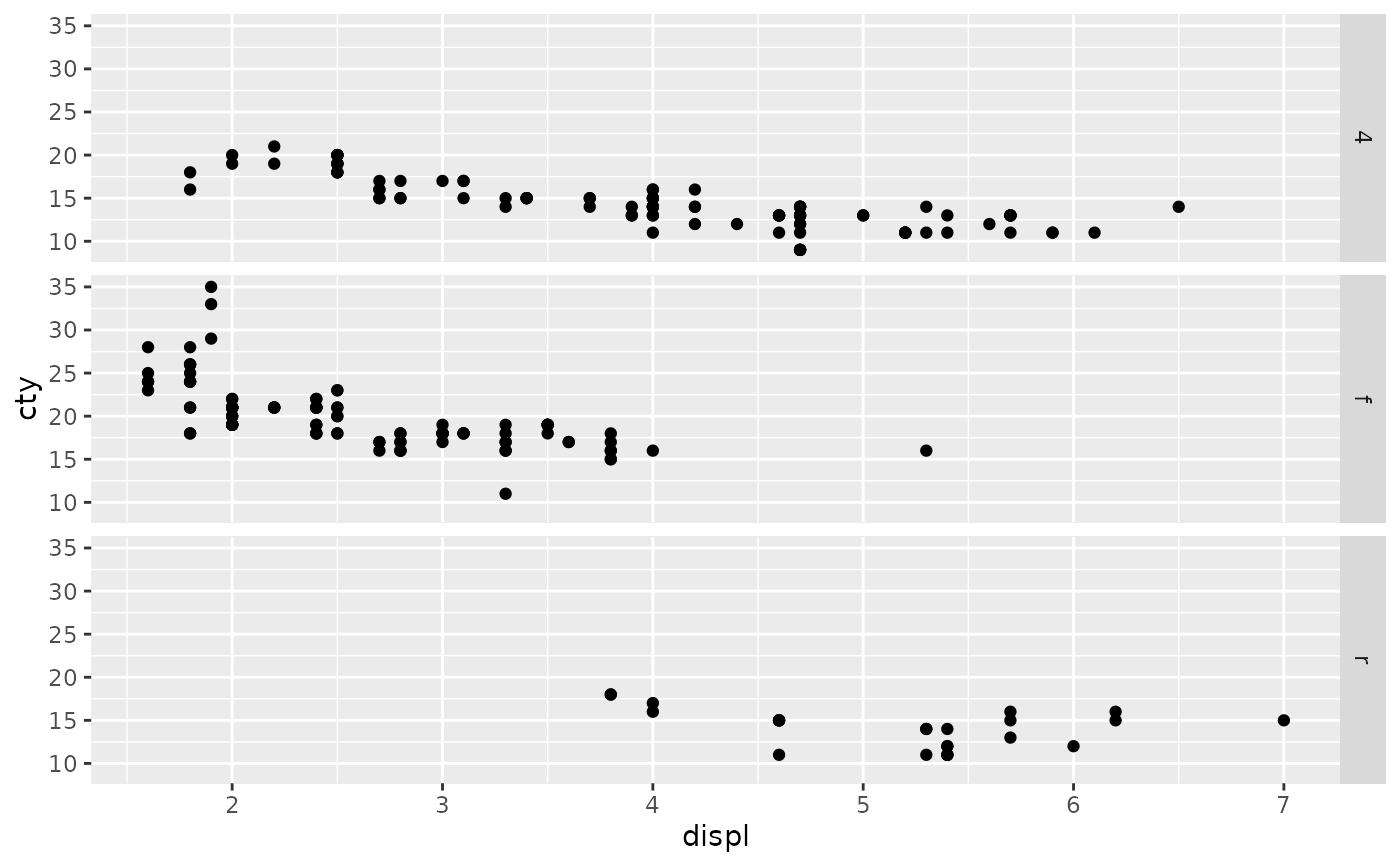
|  |  |
| --- | --- |
| **rows, cols** | A set of variables or expressions quoted by [vars()](https://ggplot2.tidyverse.org/reference/vars.html) and defining faceting groups on the rows or columns dimension. The variables can be named (the names are passed to labeller).  For compatibility with the classic interface, rows can also be a formula with the rows (of the tabular display) on the LHS and the columns (of the tabular display) on the RHS; the dot in the formula is used to indicate there should be no faceting on this dimension (either row or column). |
| **scales** | Are scales shared across all facets (the default, "fixed"), or do they vary across rows ("free\_x"), columns ("free\_y"), or both rows and columns ("free")? |
| **space** | If "fixed", the default, all panels have the same size. If "free\_y" their height will be proportional to the length of the y scale; if "free\_x" their width will be proportional to the length of the x scale; or if "free" both height and width will vary. This setting has no effect unless the appropriate scales also vary. |
| **shrink** | If TRUE, will shrink scales to fit output of statistics, not raw data. If FALSE, will be range of raw data before statistical summary. |
| **labeller** | A function that takes one data frame of labels and returns a list or data frame of character vectors. Each input column corresponds to one factor. Thus there will be more than one with vars(cyl, am). Each output column gets displayed as one separate line in the strip label. This function should inherit from the "labeller" S3 class for compatibility with [labeller()](https://ggplot2.tidyverse.org/reference/labeller.html). You can use different labeling functions for different kind of labels, for example use [label\_parsed()](https://ggplot2.tidyverse.org/reference/labellers.html) for formatting facet labels. [label\_value()](https://ggplot2.tidyverse.org/reference/labellers.html) is used by default, check it for more details and pointers to other options. |
| **as.table** | If TRUE, the default, the facets are laid out like a table with highest values at the bottom-right. If FALSE, the facets are laid out like a plot with the highest value at the top-right. |
| **switch** | By default, the labels are displayed on the top and right of the plot. If "x", the top labels will be displayed to the bottom. If "y", the right-hand side labels will be displayed to the left. Can also be set to "both". |
| **drop** | If TRUE, the default, all factor levels not used in the data will automatically be dropped. If FALSE, all factor levels will be shown, regardless of whether or not they appear in the data. |
| **margins** | Either a logical value or a character vector. Margins are additional facets which contain all the data for each of the possible values of the faceting variables. If FALSE, no additional facets are included (the default). If TRUE, margins are included for all faceting variables. If specified as a character vector, it is the names of variables for which margins are to be created. |
| **facets** | This argument is soft-deprecated, please use rows and cols instead. |

## Examples

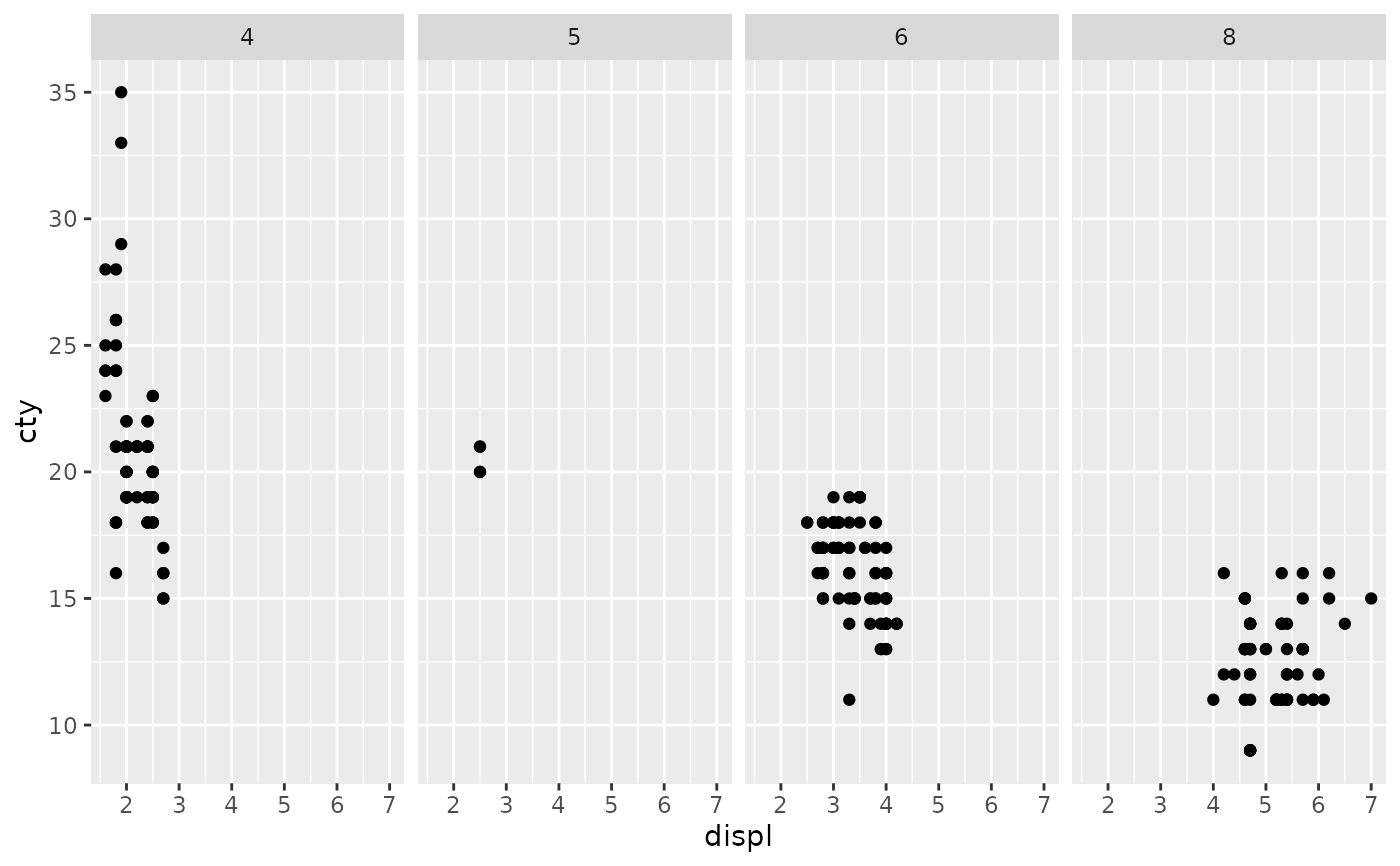
p <- [ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mpg, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(displ, cty)) + [geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)()

# Use vars() to supply variables from the dataset:

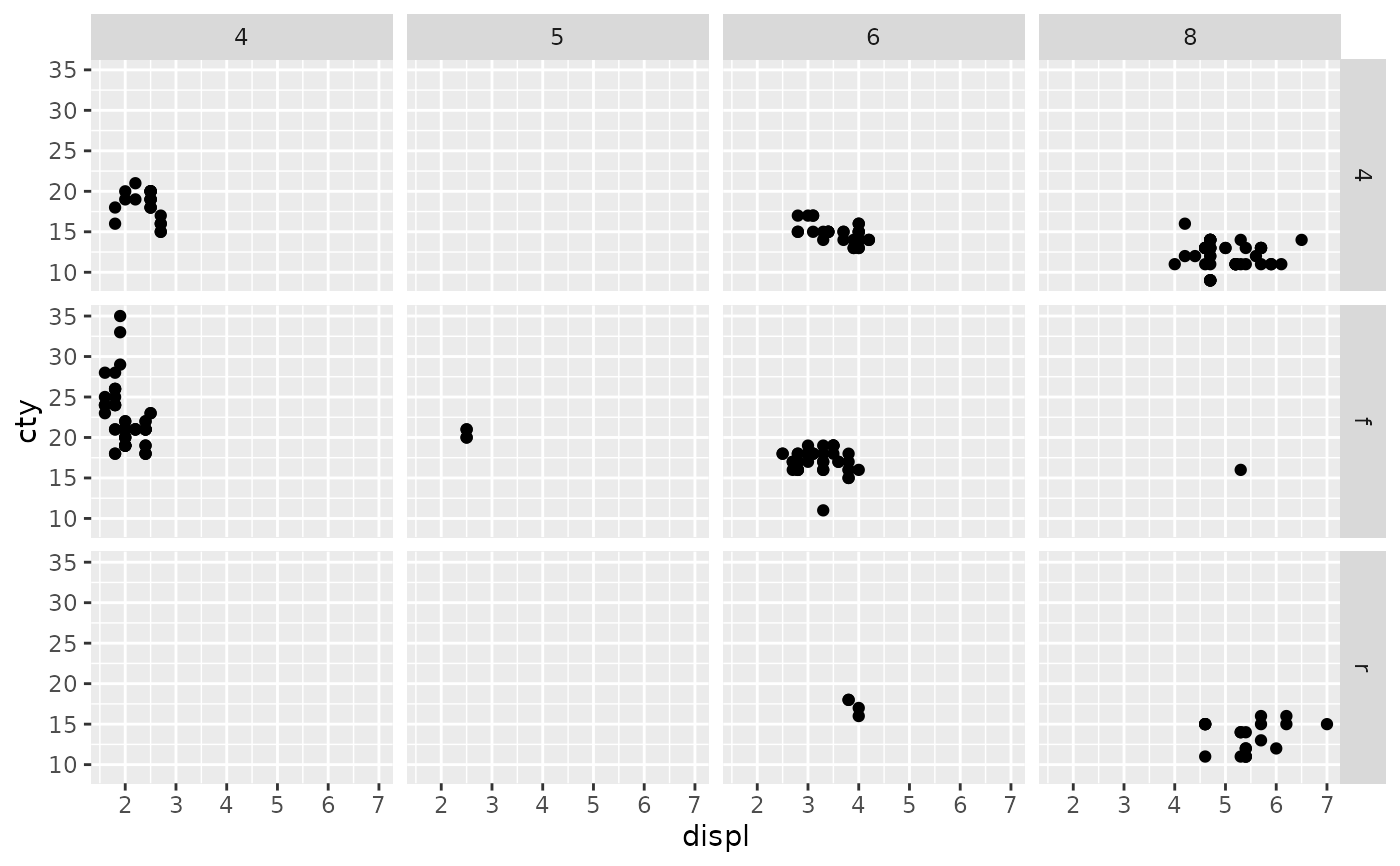
p + facet\_grid(rows = [vars](https://ggplot2.tidyverse.org/reference/vars.html)(drv))



p + facet\_grid(cols = [vars](https://ggplot2.tidyverse.org/reference/vars.html)(cyl))



p + facet\_grid([vars](https://ggplot2.tidyverse.org/reference/vars.html)(drv), [vars](https://ggplot2.tidyverse.org/reference/vars.html)(cyl))



# To change plot order of facet grid,

# change the order of variable levels with factor()

# If you combine a facetted dataset with a dataset that lacks those

# faceting variables, the data will be repeated across the missing

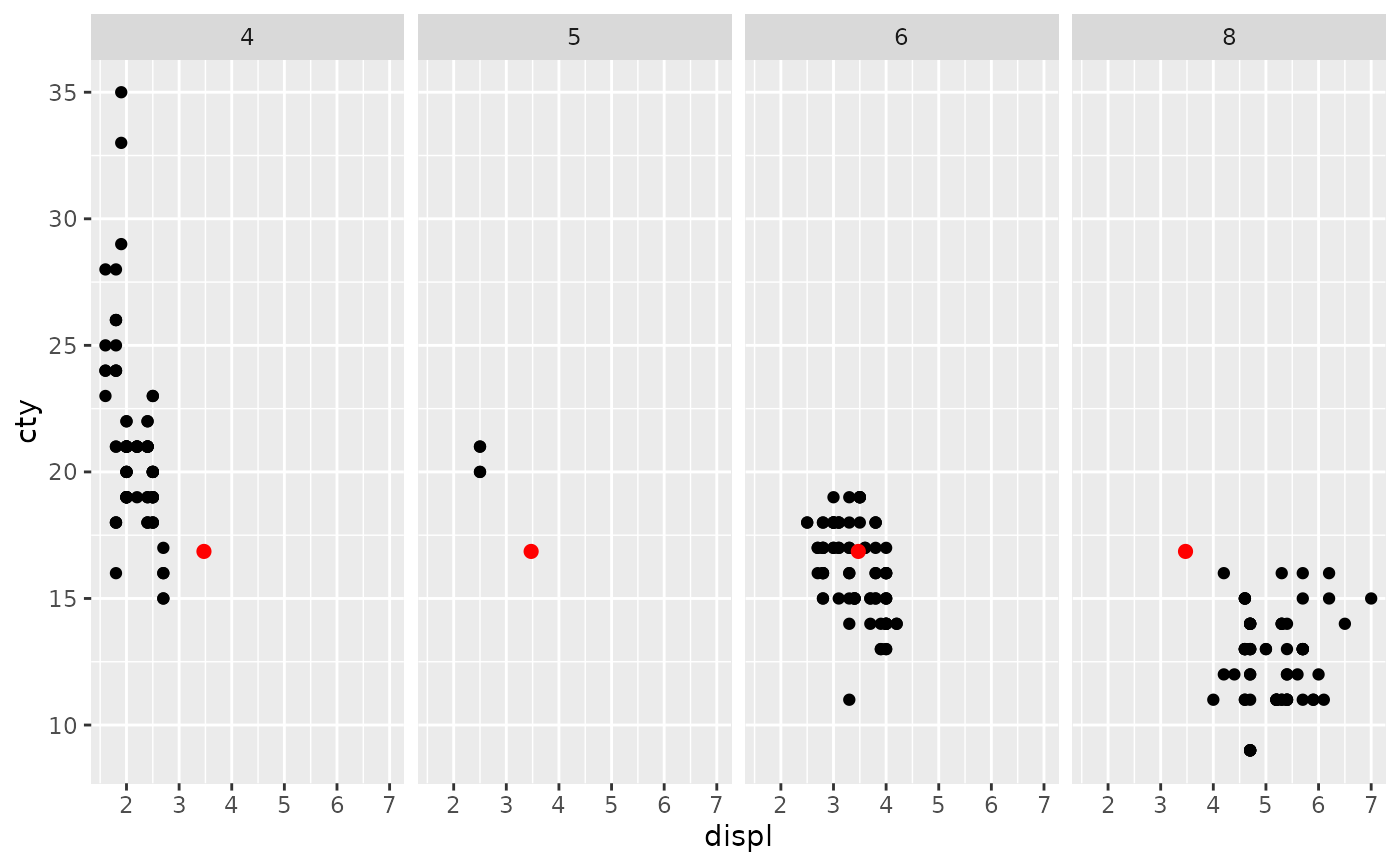
# combinations:

df <- [data.frame](https://rdrr.io/r/base/data.frame.html)(displ = [mean](https://rdrr.io/r/base/mean.html)(mpg$displ), cty = [mean](https://rdrr.io/r/base/mean.html)(mpg$cty))

p +

facet\_grid(cols = [vars](https://ggplot2.tidyverse.org/reference/vars.html)(cyl)) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)(data = df, colour = "red", size = 2)



# Free scales -------------------------------------------------------

# You can also choose whether the scales should be constant

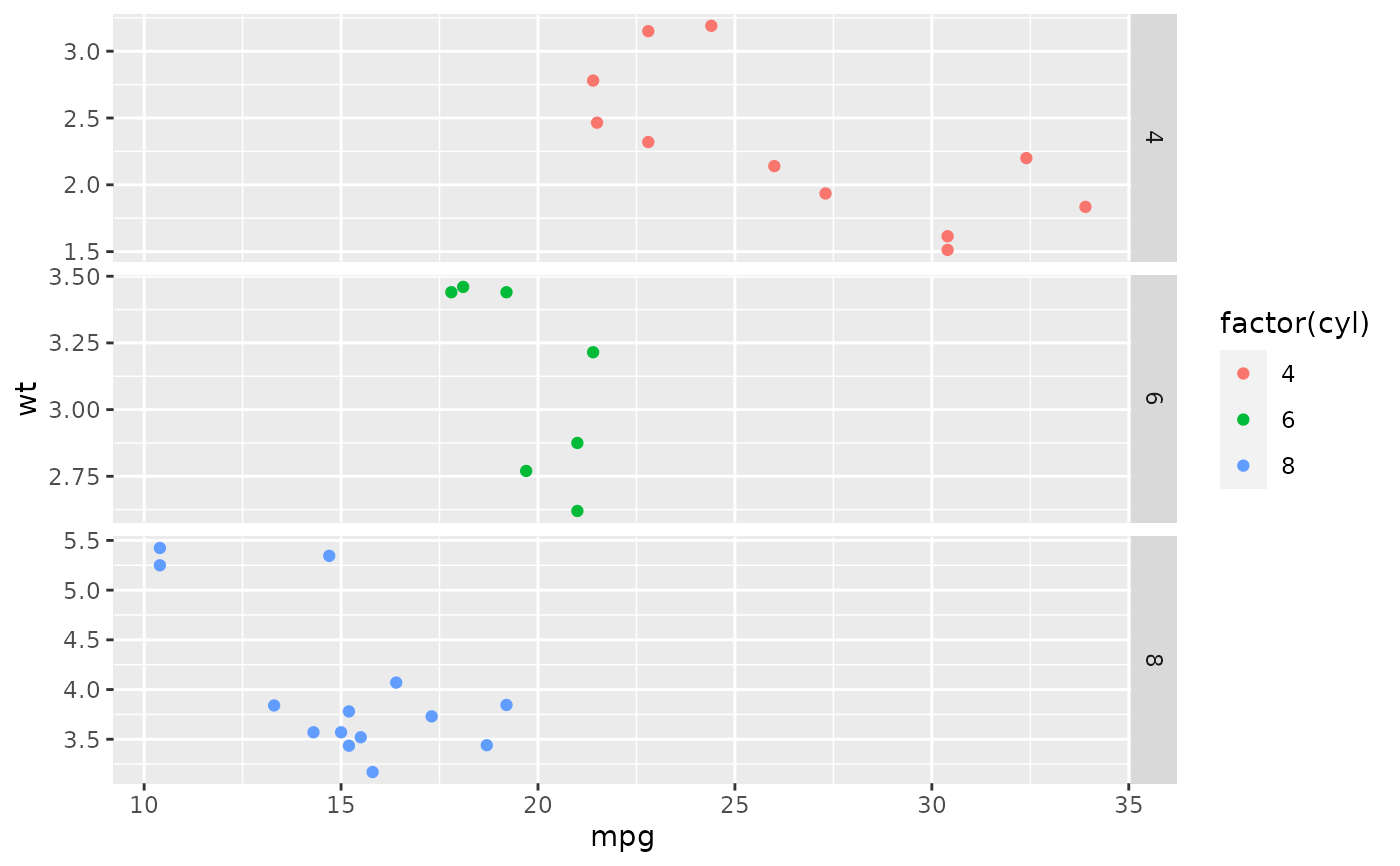
# across all panels (the default), or whether they should be allowed

# to vary

mt <- [ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(mpg, wt, colour = [factor](https://rdrr.io/r/base/factor.html)(cyl))) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)()

mt + facet\_grid([vars](https://ggplot2.tidyverse.org/reference/vars.html)(cyl), scales = "free")



# If scales and space are free, then the mapping between position

# and values in the data will be the same across all panels. This

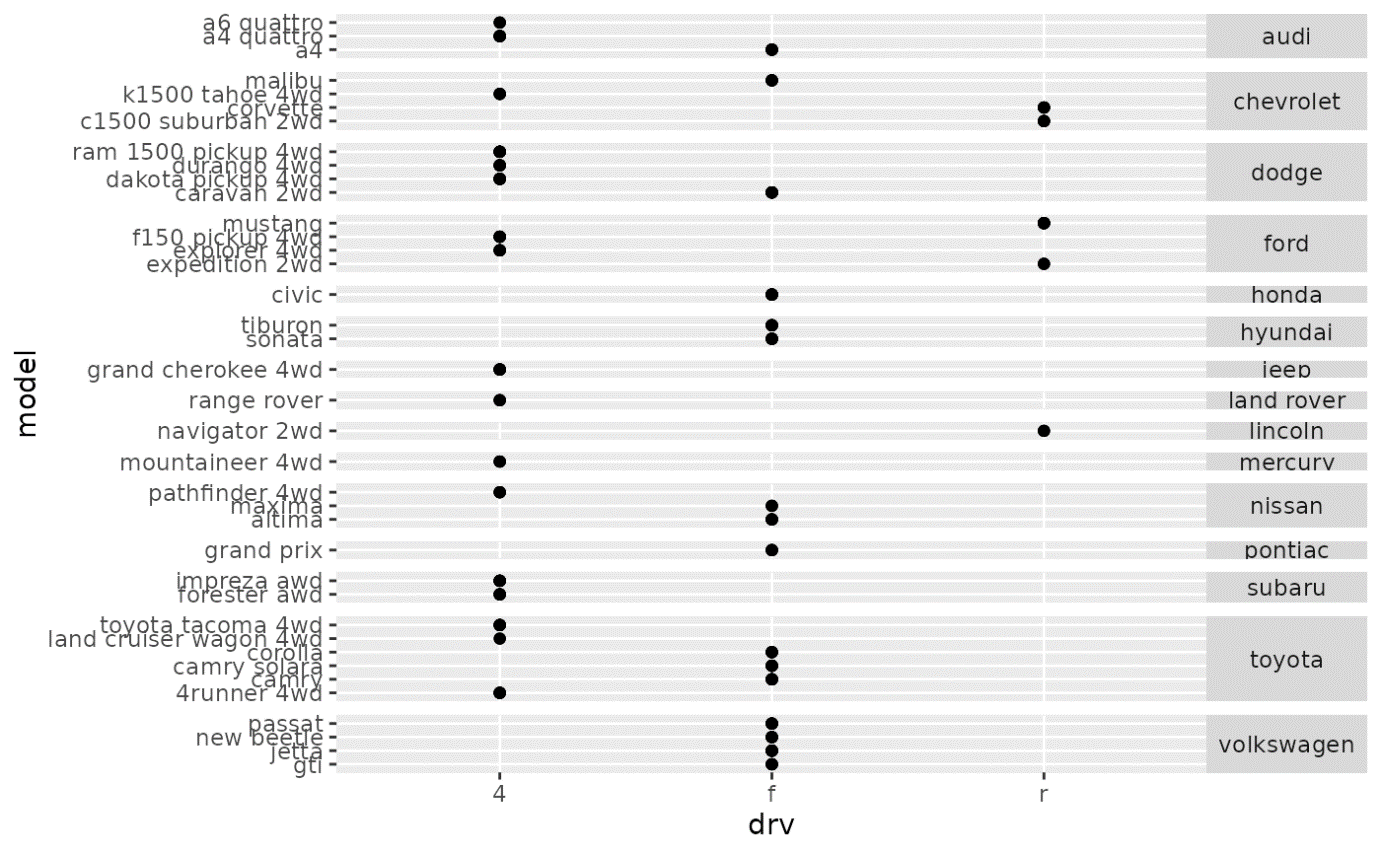
# is particularly useful for categorical axes

[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mpg, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(drv, model)) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)() +

facet\_grid(manufacturer ~ ., scales = "free", space = "free") +

[theme](https://ggplot2.tidyverse.org/reference/theme.html)(strip.text.y = [element\_text](https://ggplot2.tidyverse.org/reference/element.html)(angle = 0))



# Margins ----------------------------------------------------------

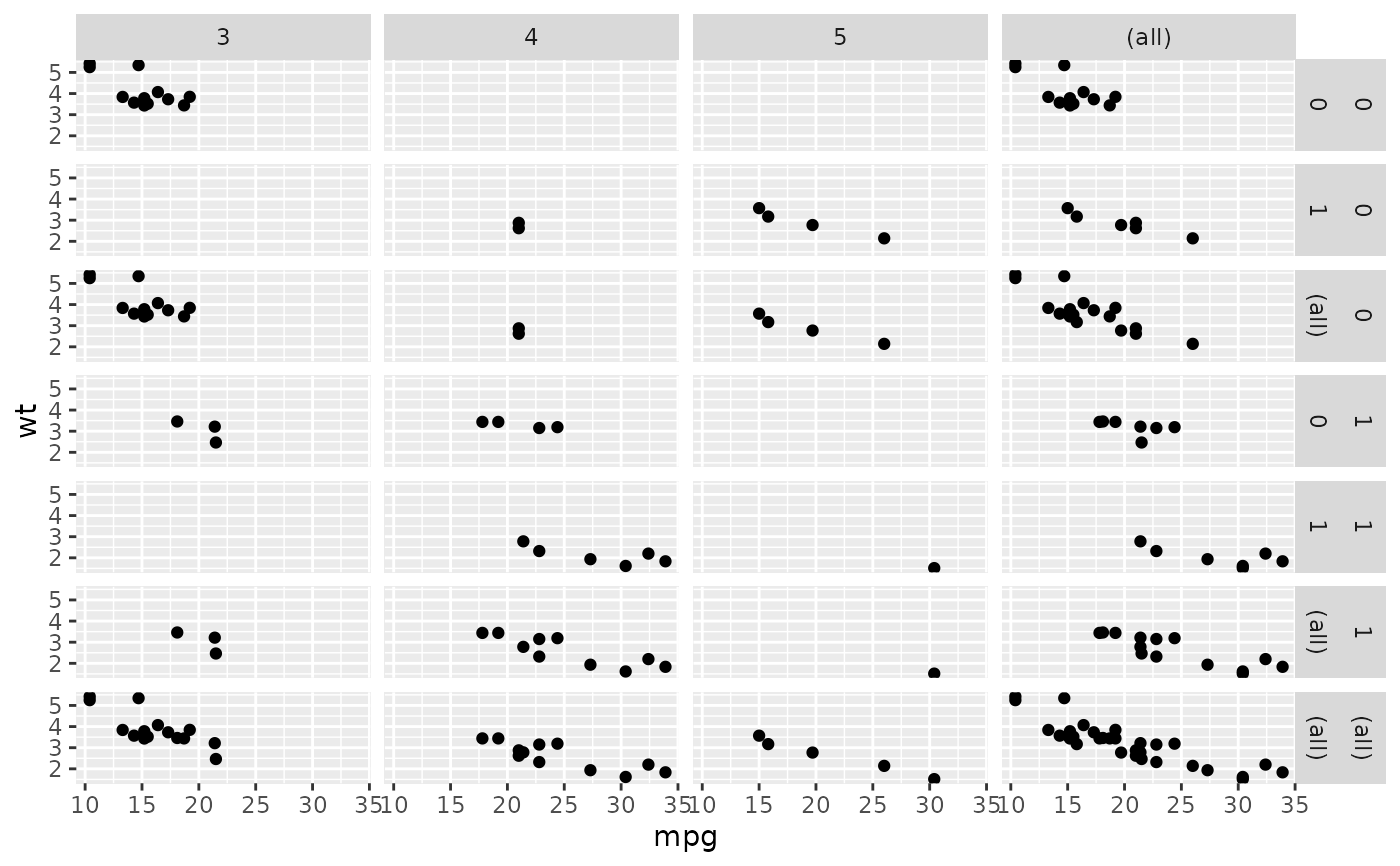
# \donttest{

# Margins can be specified logically (all yes or all no) or for specific

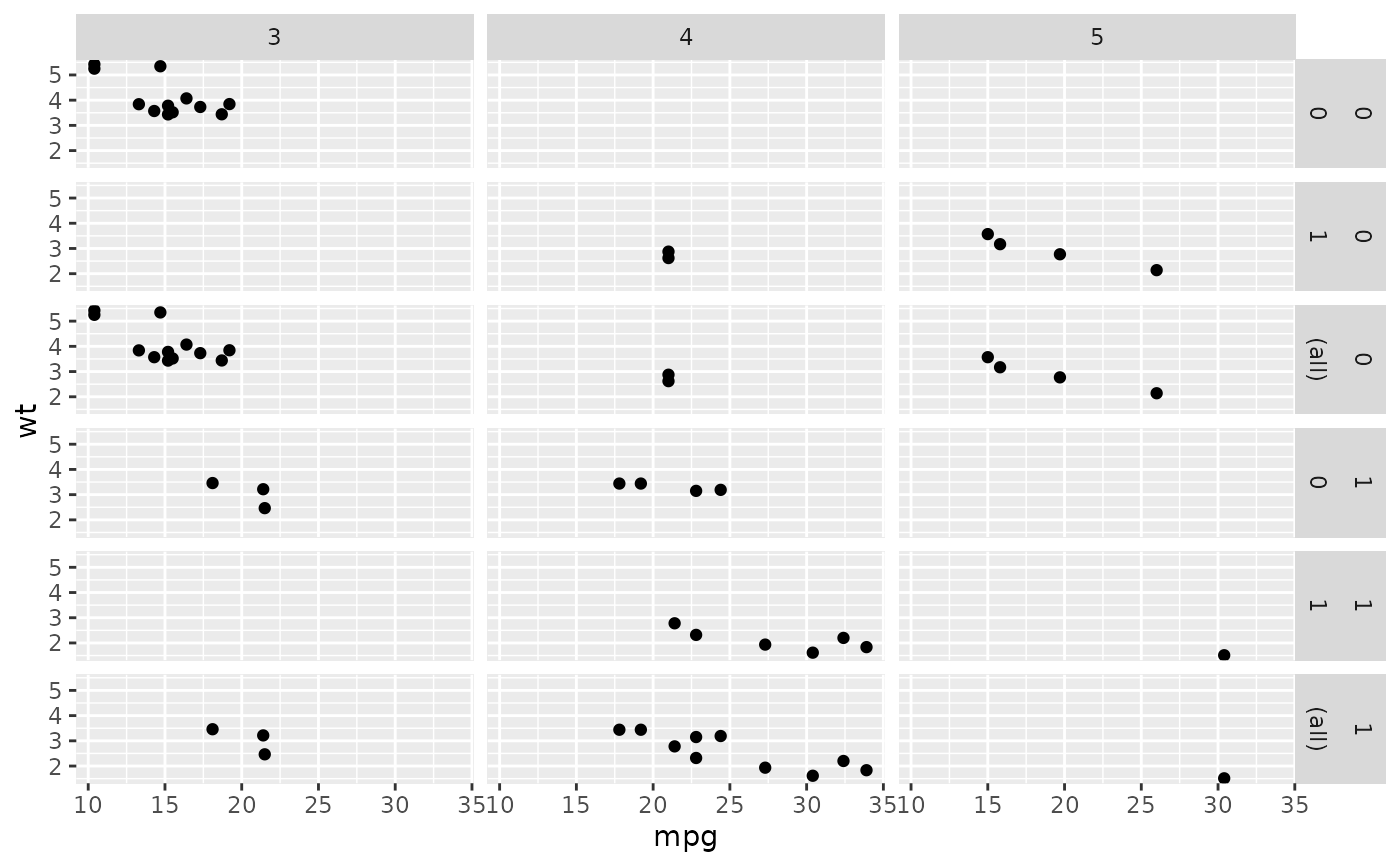
# variables as (character) variable names

mg <- [ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(x = mpg, y = wt)) + [geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)()

mg + facet\_grid(vs + am ~ gear, margins = TRUE)



mg + facet\_grid(vs + am ~ gear, margins = "am")

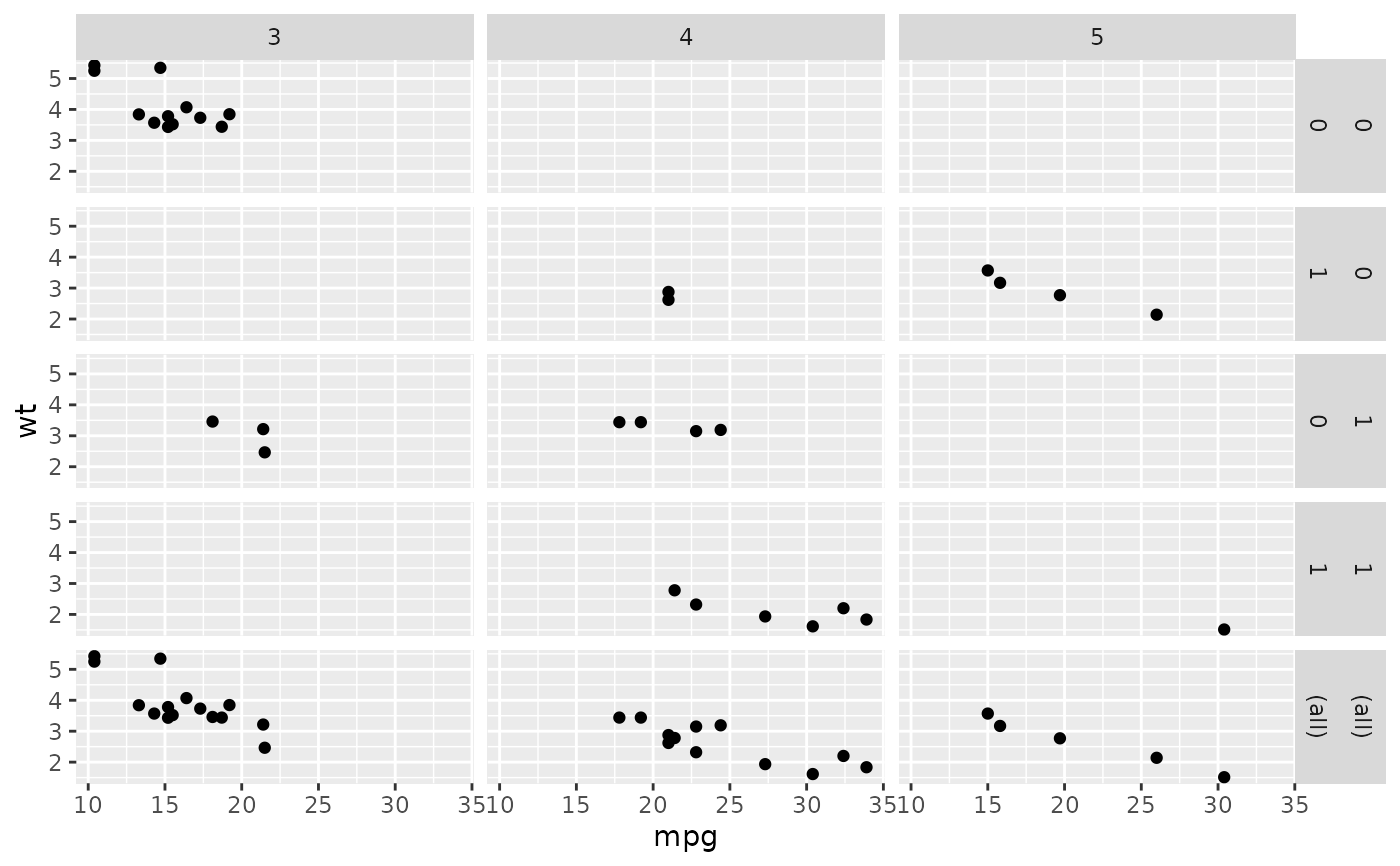


# when margins are made over "vs", since the facets for "am" vary

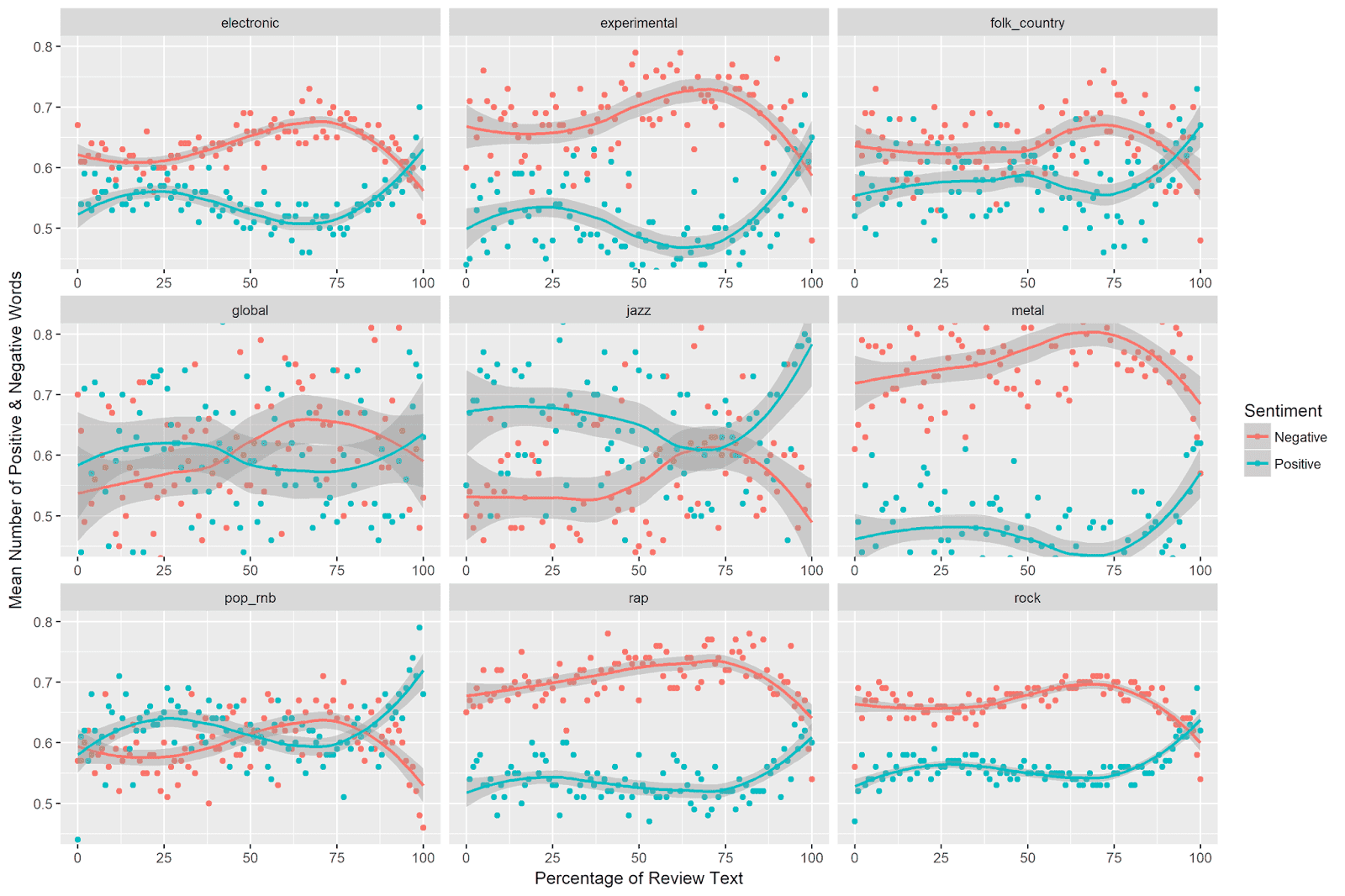
# within the values of "vs", the marginal facet for "vs" is also

# a margin over "am".

mg + facet\_grid(vs + am ~ gear, margins = "vs")



# }

[](https://i0.wp.com/4.bp.blogspot.com/-TZJydoWsEsA/WnNzZ3sDzmI/AAAAAAAAAac/v7Adnz1dWGUc6XzTncTuzpbcJCcHdIDzgCLcBGAs/s1600/pos_neg_by_genre.png?ssl=1)

The increase in negative sentiment and the decrease in positive sentiment just shy of 75% (and subsequent reversal of this trend) is evident across genres. The relative levels of positive and negative sentiment, however, differ across genres. Interestingly, jazz is the only genre for which positive sentiment use is consistently higher than negative sentiment use.

**Caveats and Limitations**

*Dictionary Considerations*

One striking pattern in the data was that negative sentiment use was consistently higher than positive sentiment use. Do the Pitchfork reviews really contain more negative than positive sentiment? I think that it’s important to keep in mind that the bing sentiment dictionaries contain twice as many negative as positive words. One alternative interpretation of the difference in mean levels of positive vs. negative sentiment, therefore, is that we have an easier time detecting negative sentiment (because we have many more negative words in our dictionary) than positive sentiment (which has half as many words). In sum, it’s hard to say from these data whether the Pitchfork music reviews are really more negative than positive overall.

*Effect Size*

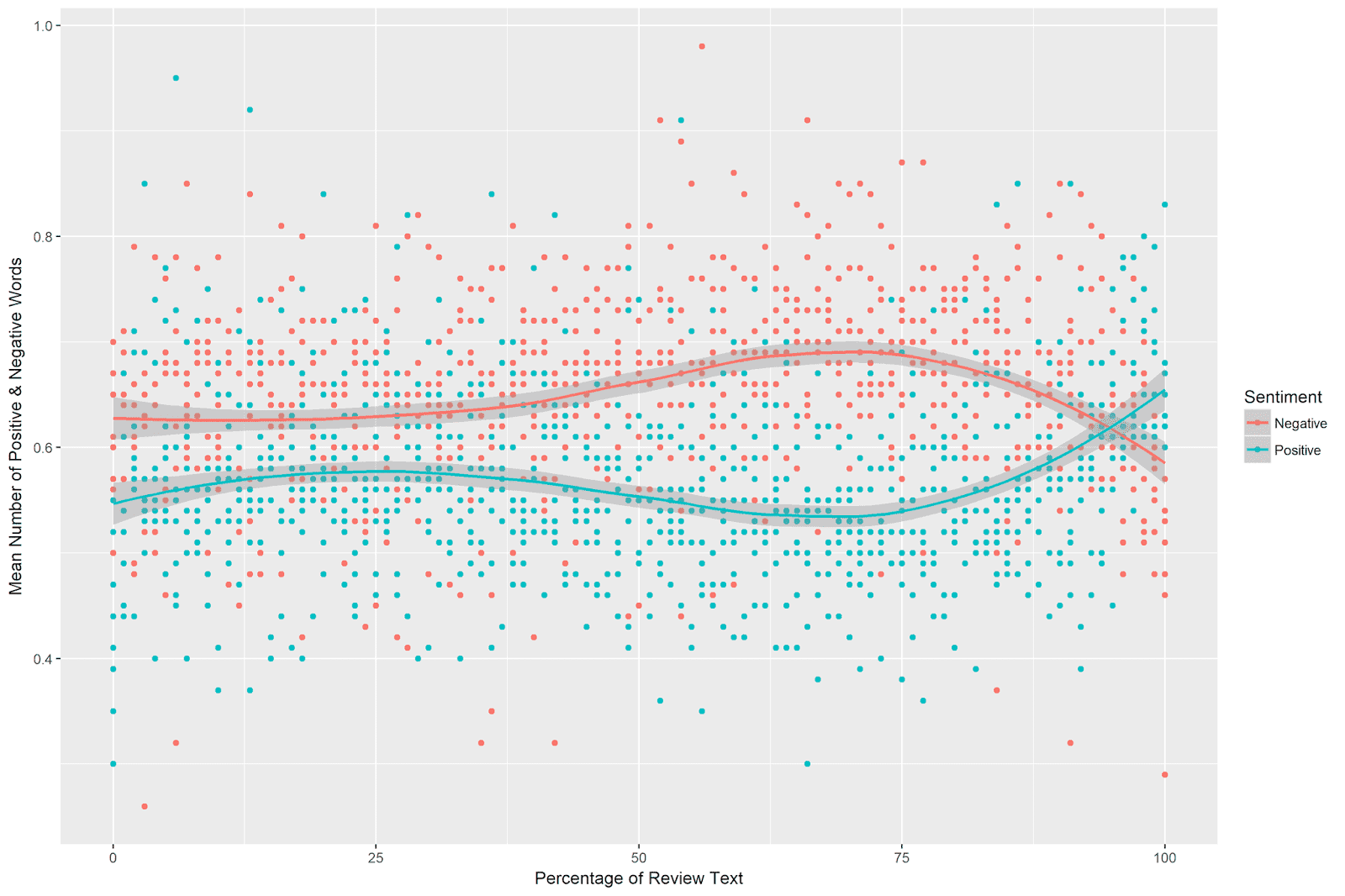
We saw in the above figures that positive and negative sentiment dipped and peaked across the review texts. How large are these decreases and increases in sentiment use? This question relates to the *effect size* of the observed trends in emotion use across the review texts.

One way to interpret observed effect sizes is by using domain knowledge, e.g. expertise accumulated through previous work in the domain. Unfortunately, we’re using a very specific metric here (mean number of words across percentages of a text), and I don’t know of any existing studies using this type of coding. This is the first time I myself have used this approach!

For classical statistical models, there are statistical definitions of effect size, but these do not apply to the type of local regression (loess) models we are using here.

One principal that I’ve heard mentioned a number of times is that an effect worth considering should be visible with the naked eye. The trends are quite striking in the above plots, but I’ve set the axes in such a way that the differences are highlighted. What happens if we plot the data, but allow the y-axis to be defined by the minimum and maximum observed values of mean positive and negative sentiment? (Note that this approach parallels the underlying logic of most measures of effect size, which compare observed differences scaled to some measure of variation in the data.)

The plot (obtained by using the above code with the *coord\_cartesian* command removed) looks like this:

[](https://i2.wp.com/3.bp.blogspot.com/-6SfWXYw04oM/WnXnnRPhxbI/AAAAAAAAAaw/U2nTiGB74CU8KbRj86rcGFyg1Xja9dHZQCLcBGAs/s1600/pos_neg_full_y_axis.png?ssl=1)

The patterns still looks clear to me. Even when shown across the observed values of mean sentiment, the divergence and subsequent reversal in sentiment use is easily observable.

**Conclusion**

In summary, in this post we examined the use of positive and negative sentiment words across the course of Pitchfork album reviews. We first turned our raw data (with one review per line) into a tidy text dataframe (with one word per line). We then removed stopwords and calculated the position of each word in terms of its percentage in the review text. Next, we calculated the mean positive and negative sentiment words at each percentage of the review texts for each genre separately. Finally, we visualized the overall trends of positive and negative sentiment across the review texts, and also examined these trends separately across genres.

There were clear patterns for positive and negative sentiment use. Positive sentiment reached its lowest point just short of the 75th percentile of the review texts, after which it sharply increased. Negative sentiment peaked just shy of the 75th percentile, after which it sharply decreased. In sum, it appears that Pitchfork music reviews increase in negativity and decrease in positivity around 75the percentile of the review texts, after which they become much less negative and much more positive (in essence ending on a positive note).