Fedspeak Documentation

Darren Chang

2020-07-11

Contents

Introduction

Here is the link to the Beige Book, a qualitative report that the Federal Reserve releases 8 times a year. In past decades, the Beige Book has been released as often as once a month. Each Federal Reserve district writes their own report and a National Summary is prepared by Research Associates at a district bank on a rotating basis.

Working

More information on the Beige Book and previous forecasting methods to come.

Text Mining

Text mining was completed in `Python 3.7` using `BeautifulSoup`. Setup by importing `Pandas`, `BeauitfulSoup`, and `Requests`.

```
## ---- setup
import pandas as pd
import numpy as np

import os #setwd
import time #timing

# scraping
from bs4 import BeautifulSoup
from contextlib import closing
import requests # website requests
from requests.exceptions import RequestException
from requests import get
```

I scrape the Minneapolis Fed's Beige Book Archive, which hosts `html` files for each Beige Book back to 1970. The links to each website to be scraped is in the `links.csv` file in this repo. Import it as a dataframe.

```
links = pd.read_csv('~/_econ/fedspeak/text mining/links.csv')
links.head(20)
```

```
##     year  month  ...      report      date
## 0   1970      5  ...  1970-05-at  5/1/1970
## 1   1970      5  ...  1970-05-bo  5/1/1970
## 2   1970      5  ...  1970-05-ch  5/1/1970
## 3   1970      5  ...  1970-05-cl  5/1/1970
## 4   1970      5  ...  1970-05-da  5/1/1970
## 5   1970      5  ...  1970-05-kc  5/1/1970
## 6   1970      5  ...  1970-05-mi  5/1/1970
## 7   1970      5  ...  1970-05-ny  5/1/1970
## 8   1970      5  ...  1970-05-ph  5/1/1970
## 9   1970      5  ...  1970-05-ri  5/1/1970
## 10  1970      5  ...  1970-05-sf  5/1/1970
## 11  1970      5  ...  1970-05-sl  5/1/1970
## 12  1970      5  ...  1970-05-su  5/1/1970
## 13  1970      6  ...  1970-06-at  6/1/1970
## 14  1970      6  ...  1970-06-bo  6/1/1970
## 15  1970      6  ...  1970-06-ch  6/1/1970
## 16  1970      6  ...  1970-06-cl  6/1/1970
## 17  1970      6  ...  1970-06-da  6/1/1970
## 18  1970      6  ...  1970-06-kc  6/1/1970
## 19  1970      6  ...  1970-06-mi  6/1/1970
```

```
##
## [20 rows x 6 columns]
```

```
links.tail(20)
```

```
##       year  month  ...     report      date
## 5674  2020      3  ...  2020-03-mi  3/1/2020
## 5675  2020      3  ...  2020-03-ny  3/1/2020
## 5676  2020      3  ...  2020-03-ph  3/1/2020
## 5677  2020      3  ...  2020-03-ri  3/1/2020
## 5678  2020      3  ...  2020-03-sf  3/1/2020
## 5679  2020      3  ...  2020-03-sl  3/1/2020
## 5680  2020      3  ...  2020-03-su  3/1/2020
## 5681  2020      4  ...  2020-04-at  4/1/2020
## 5682  2020      4  ...  2020-04-bo  4/1/2020
## 5683  2020      4  ...  2020-04-ch  4/1/2020
## 5684  2020      4  ...  2020-04-cl  4/1/2020
## 5685  2020      4  ...  2020-04-da  4/1/2020
## 5686  2020      4  ...  2020-04-kc  4/1/2020
## 5687  2020      4  ...  2020-04-mi  4/1/2020
## 5688  2020      4  ...  2020-04-ny  4/1/2020
## 5689  2020      4  ...  2020-04-ph  4/1/2020
## 5690  2020      4  ...  2020-04-ri  4/1/2020
## 5691  2020      4  ...  2020-04-sf  4/1/2020
## 5692  2020      4  ...  2020-04-sl  4/1/2020
## 5693  2020      4  ...  2020-04-su  4/1/2020
##
## [20 rows x 6 columns]
```

Next, I define our scraping functions that iterate through the links, opens the url, and returns all of the text with the `<p>` tag. Much of the code for this portion was taken from Real Python's guide to web scraping.

```
## ---- scraping
# functions for getting URLs, with error logging
def simple_get(url):
    """
    Attempts to get the content at `url` by making an HTTP GET request.
    If the content-type of response is some kind of HTML/XML, return the
    text content, otherwise return None.
    """
    try:
        with closing(get(url, stream = True)) as resp:
            if is_good_response(resp):
                soup = BeautifulSoup(resp.content, 'html.parser')
                results = soup.find_all('p')
                return results
            else:
                return None
    except RequestException as e:
        log_error('Error during requests to {0} : {1}'.format(url, str(e)))
        return None


def is_good_response(resp):
    """
    Returns True if the response seems to be HTML, False otherwise.
```

```
    """
    content_type = resp.headers['Content-Type'].lower()
    return (resp.status_code == 200
            and content_type is not None
            and content_type.find('html') > -1)

def log_error(e):
    """
    log errors
    """
    print(e)
```

We define another function to do the actual work of scraping. This returns a dataframe with the metadata about the website we're scraping (date, year, bank, url) merged with lists of lists of the actual text in each page.

```
# scraping a set of links
def scrape(links, #dataframe of urls and other info
):
    """
    function for scraping set of links to dataframe.
    returns data frame of raw text in lists of lists
    """
    links_use = links['url'].values.tolist() # extract urls as list
    fed_text_raw = pd.DataFrame() #empty df

    for url in links_use:
        text = simple_get(url)
        df = pd.DataFrame({'url': url, 'text': [text]})
        fed_text_raw = fed_text_raw.append(df, ignore_index = True)
    fed_text_raw = pd.DataFrame(fed_text_raw)
    fed_text_raw.columns = fed_text_raw.columns.str.strip() #strip column names

    return fed_text_raw
```

Finally, I scrape our links. The returned file was over 1GB in size when saved as a csv file and the process usually takes about 2.5 hours. I would recommend exporting it (to csv or whatever file storage format you prefer) to avoid scraping multiple times.

```
fed_text_raw = scrape(links)
```

Cleaning Data

We come to the unenviable task of cleaning the data, which represents a few million words from 50 years of Beige Book Reports. First, we strip the dataset of html tags and numbers. When using `BeautifulSoup`, this is often done with the `get_text()` command; however, we use `find_all` with the html paragraph tag to scrape our data, so `get_text` does not work for this case.

Preprocessing

The following function uses regex to replace characters, html tags, and other wacky spacing issues.

```
## ---- cleaning

def preprocess(df,
 text_field, # field that has text
```

```
new_text_field_name # name of field that has normalized text
):
    """
    normalizes dataframes by converting it to lowercase and
    removing characters that do not contribute to natural text meaning
    """
    df[new_text_field_name] = (df[text_field]
        .apply(lambda elem: re.sub(r"(@[A-Za-z0-9]+)|([^0-9A-Za-z \t])|(\w+:\/\/\S+)|^rt|http.+?",
            "", str(elem)))
        .apply(lambda elem: re.sub("<.*?>","", str(elem))) # strips out tags
        .apply(lambda elem: re.sub(r"\d+", "", str(elem)))
        )

    return df
```

Tokenization

We want to convert our bag of words for each report into the tidy text format, which Julia Silge and David Robinson define in Chapter 1 of *Text Mining in R* as:

> 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.

You may ask: why would you want to use a `tidy` format if we're using Python for mining? This is a good question, but here were some of my reasons:

1. `NLTK` is set up to use ML techniques for sentiment analysis
   - The basic sentiment analysis technique we are using involves matching words in our data with a predefined dictionary.
   - The `tidytext` package preloads the Loughran-McDonald dictionary, which is useful for analysis of finance (and economics-related) literature. More on this later, and the usage of different lexicon.
2. CSV files struggle with lengthy lists of lists.
   - Sometimes there are display errors.
   - Sometimes there are errors where the list is too many characters.
   - The processing speed can be quite slow.
3. I'm personally more familiar with visualization with `ggplot2`.
4. I do attempt to use `python` to repeat some of the data processing that was first completed with `R`. This is clearly indicated (and the reader can also ascertain which language is used by the programming syntax).

Here, we define a function for word tokenization, which borrows heavily from Michelle Fullwood's project to convert *Text Mining in R* to Python. I have modified the function to merge the tokens with the original list of links based on the date of the report.

```
# word tokenization
def unnest(df, # line-based dataframe
                  column_to_tokenize, # name of the column with the text
                  new_token_column_name, # what you want the column of words to be called
                  tokenizer_function, # what tokenizer to use = (nltk.word_tokenize)
                  original_list): # original list of data
```

```
    """
    unnests words from html and returns dataframe in long format merged with original list.
    word tokenization in tidy text format.
    """

    return (df[column_to_tokenize]
                .apply(str)
                .apply(tokenizer_function)
                .apply(pd.Series)
                .stack()
                .reset_index(level=0)
                .set_index('level_0')
                .rename(columns={0: new_token_column_name})
                .join(df.drop(column_to_tokenize, 1), how='left')
                .reset_index(drop=True)
                .merge(original_list, how = 'outer', on = 'url')
               )
```

To complete the preprocessing and tokenization process, I apply the functions to our raw data, convert it to lowercase (which is easier for future processes) and save it as a `csv` file to call it for future use.

```
fed_text_raw = preprocess(fed_text_raw, 'text', 'text')
fed_text_all = unnest(fed_text_raw, 'text', 'word', nltk.word_tokenize, links)
fed_text_all['word'] = fed_text_all['word'].str.lower() # convert to lowercase
fed_text_all.to_csv('fed_text_all.csv', index = False) # save as csv
```

Dask

If you find that `pandas` is too slow for these applications or for your data, you may want to look into `dask`, which uses multi-core processing in addition to other goodies to speed things up for large datasets. Shikhar Chauhan has a blog post about using dask for text pre-processing at MindOrks.

Stemming

Here's a stemming function that would work on a `tidy` formatted dataframe of words. I didn't end up using stemming (and instead used lemmatization) because I found stemming removed too much — what should have been words simply became nonsensical stems. DataCamp has a tutorial for both stemming and lemmatization and compares the two methods here.

```
snowball = SnowballStemmer(language = 'english')
def stemming(df,
             column_to_stem,
             stem_function, # what stemming function to use = snowball
             ):
     """
     returns stem of words in a dataframe
     """

     return (df[column_to_stem]
                 .apply(stem_function)
                 )

 stemming(fed_text_all, 'word', snowball.stem)
```

Lemmatization

Instead of stemming, I lemmatized the text by comparing the verbs in the data to the WordNet corpus in `NLTK`. This reduces a word like "swimming" to "swim," which dictionaries pick up better for sentiment analysis.

```
wordnet_lemmatizer = WordNetLemmatizer()
def lemmatize(df,
            column_to_lemmatize,
            lemmatize_func, # lemmatizing function to use = wordnet
            ):
    """
    returns root words by matching with the lemmatizing function
    """

    return (df[column_to_lemmatize]
                .apply(lambda word: lemmatize_func.lemmatize(word, pos = "v")) # part of speech is set
                )

fed_text_all['word'] = lemmatize(fed_text_all, 'word', wordnet_lemmatizer) # lemmatize
```

Stopwords

To finish the cleaning process, I remove stop words from the data, which keeps roughly 60 percent of the raw data (the csv file becomes 600 MB). Stop words are words that do not contribute to the meaning of a selection. The list of `NLTK` stopwords, which is used in this code is available in this resource for the package.

I add to the stop words list:

1. Left over html tags, such as "pp."
2. A custom list of stop words for the economic context.
    - Words such as "maturity" or "work" may be coded as positive in natural language but means something different in economics.
    - Similarly, words such as "gross" (domestic product) or "crude" (oil) may be coded as negative, but do not have such a negative conntation.
    - Len Kiefer's blog post about this was helpful, and the list is taken from his post.

```
# make custom stop words list
stop_words = stopwords.words("english")
stop_words_html = ['ppwe', 'uspp', 'pwe', 'usp', 'pp', 'usbrp'] # leftover stopwords from converting ht
# with 'p' tags
stop_words_econ = ['debt', 'gross', 'crude', 'well', 'maturity', 'work', 'marginally', 'leverage'] # st

stop_words.extend(stop_words_html)
stop_words.extend(stop_words_econ)

# delete stopwords
def remove_stops(df,
                column_to_process,
                stop_words_list, # list of stop words = stop_words
                index_reset = True # whether you want to reset index or not
                ):
    """
    function remove stopwords if data is a pandas dataframe and in tidy format e.g. long.
    different from typical pandas format which is lists of words in wide format.
    """
```

```
    if index_reset == True:
        return (df[~df[column_to_process].isin(stop_words_list)]
                .reset_index(drop = True)) # this line resets index
    else:
        return df[~df[column_to_process].isin(stop_words_list)]

# remove_stops(fed_text_test, 'word', stop_words) # test code
fed_text_all = remove_stops(fed_text_all, 'word', stop_words)
```

I saved the cleaned data as a csv to read it later for sentiment analysis.

```
fed_text_all.to_csv('fed_text_all.csv', index = False) # save as csv
```

But before that, some word frequency summary statistics:

```
# summary statistics
fed_text_all = pd.read_csv('~/_econ/fedspeak/text mining/fed_text_all.csv') # read csv
fed_text_all['word'].value_counts().head(20)
```

```
## report      131710
## district     89069
## increase     82694
## sales        79021
## price        76307
## continue     60095
## demand       56050
## contact      52839
## remain       50238
## activity     49760
## year         46716
## percent      46371
## new          45514
## p            43508
## loan         43425
## expect       37645
## level        36247
## market       36023
## strong       34933
## bank         34497
## Name: word, dtype: int64
```

It's unsurprising to me that reports and district are mentioned often. After all, the Beige Book is a report that is grouped by district. It appears that real economic activity is more important than financial activity, based on words like demand and sales. This isn't surprising, either, as the Fed's concern about financial activity is certainly more recent as markets have grown more important in the wake of the Great Recession.

Sentiment Analysis

in R

The `tidytext` package has a convenient `get_sentiments()` function, which essentially uses a merge method to match a pre-defined dictionary to the data in tidy format. We use python to emulate this method in a following section.

Let's first load some packages and set up our working environment. I use the `vroom` package that is part of the `tidyverse` to load in the file with all of the words. It is significantly faster and easier to use than any `read_csv` function that I've used before.

```r
library(tidyverse)
library(tidytext)
library(vroom)
library(ggthemes)
library(zoo)
library(tidyquant)
library(lubridate)
library(ggfortify)

setwd("C:\\Users\\darre\\Documents\\_econ\\fedspeak\\sentiment analysis")
fed_text_all <- vroom('../text mining/fed_text_all.csv') # read csv

recessions.df <- read.table(textConnection(
    "Peak, Trough
    1948-11-01, 1949-10-01
    1953-07-01, 1954-05-01
    1957-08-01, 1958-04-01
    1960-04-01, 1961-02-01
    1969-12-01, 1970-11-01
    1973-11-01, 1975-03-01
    1980-01-01, 1980-07-01
    1981-07-01, 1982-11-01
    1990-07-01, 1991-03-01
    2001-03-01, 2001-11-01
    2007-12-01, 2009-06-01"),
    sep=',',
    colClasses = c('Date', 'Date'),
    header = TRUE)
```

Polarity

We find the polarity from our data by report with the `inner_join` method in *Text Mining in R*:

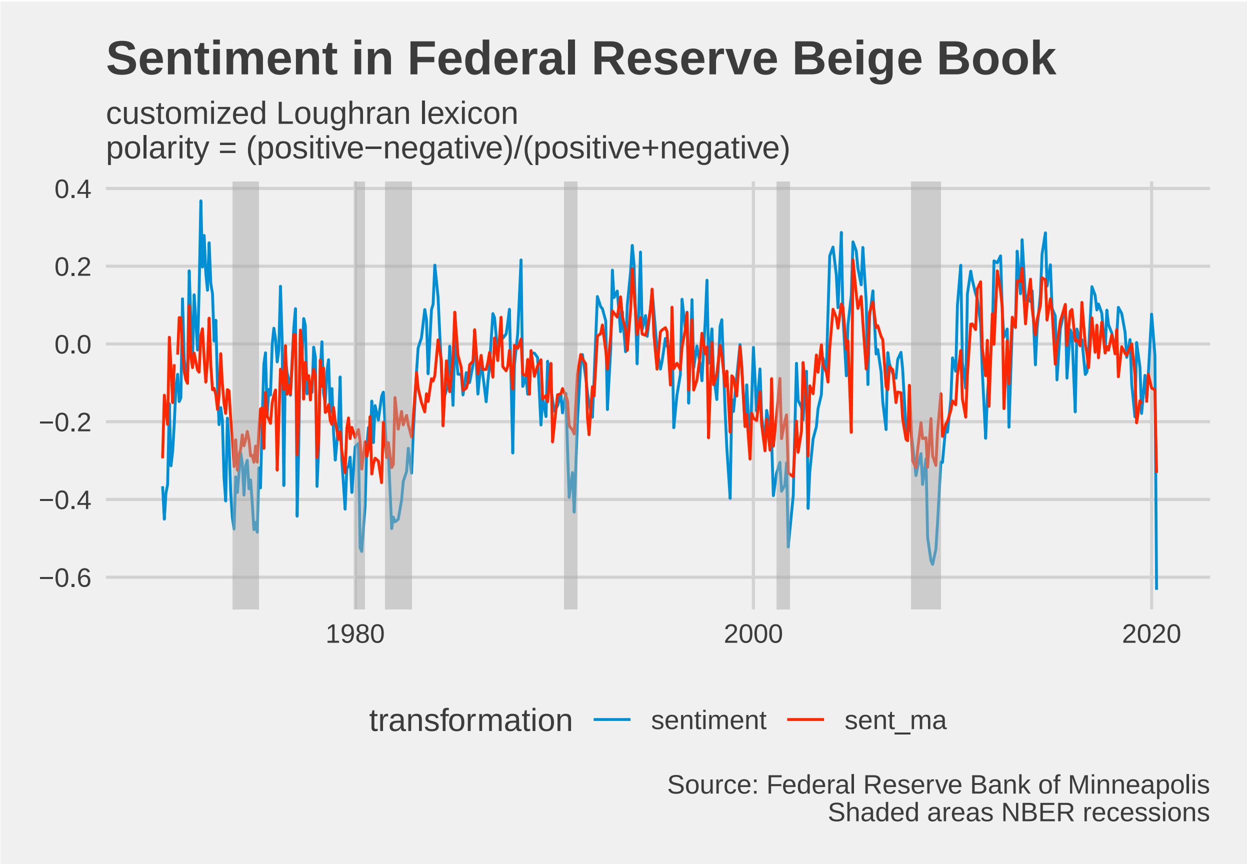
```r
fed_sentiment <-
    fed_text_all %>%
    inner_join(get_sentiments("loughran")) %>% # or bing
    # inner_join(get_sentiments("bing")) %>%
    count(date, sentiment) %>%
    pivot_wider(names_from = sentiment,
                values_from = n,
                values_fill = 0) %>%
    mutate(sentiment = (positive - negative)/(positive + negative)) %>%
    mutate(date = as.Date(date, format = "%m/%d/%Y")) %>%
    filter(sentiment != 1) %>%
    filter(date != "2015-07-01") %>%
    mutate(sent_ma = rollmean(sentiment, k = 3, fill = NA)) %>%
    select(date, sentiment, sent_ma) %>%
    pivot_longer(-date,
            names_to = 'transformation',
            values_to = 'value') %>%
    mutate(transformation = as_factor(transformation))
```

Much of this is based on Len Kiefer's code, which is linked to in an earlier part of this post. A few key differences:

- I use the Loughran-McDonald dictionary instead of the Bing dictionary. I compare the results later — they are similar, although the finance-specific lexicon seems to fit the context better.
- I substituted `pivot_wider` for `spread` as `spread` is beginning to become deprecated.
- The Minneapolis Fed doesn't have data for June 2015 (not sure the reason), so I filter that out.
- I use the `zoo` package to generate rolling means based on quarters, since there is quite a bit of noise.

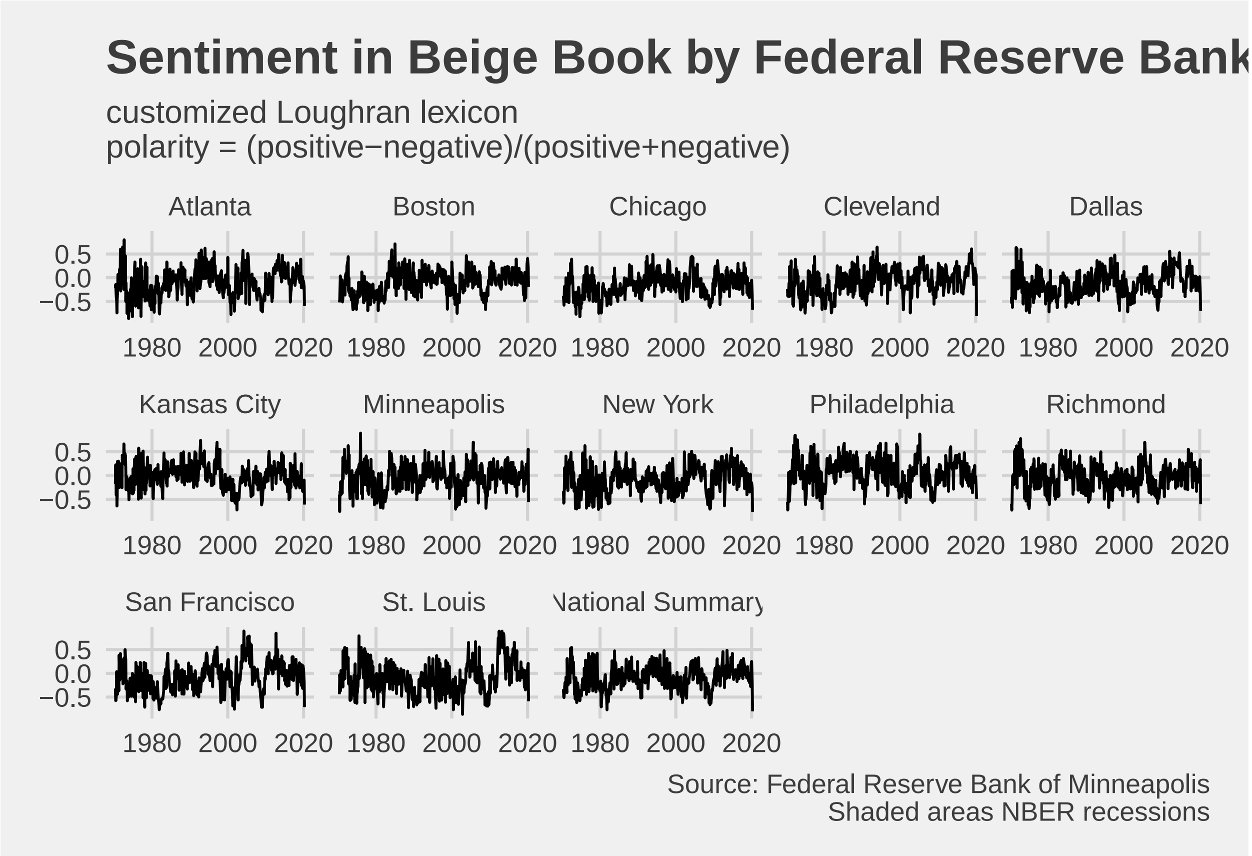
Here is the plot:

```
ggplot(fed_sentiment,
    aes(x = date,
        y = value,
        color = transformation)) +
    geom_line(aes()) +
    scale_color_fivethirtyeight() +
    theme_fivethirtyeight() +
    scale_x_date(
        #breaks = "5 years",
    limits = as.Date(c("1970-01-01","2020-06-01")),
    date_labels = "%Y") +
    labs(x = "Beige Book Report (~8/year)",
        y = "polarity",
        title = "Sentiment in Federal Reserve Beige Book",
        subtitle = "customized Loughran lexicon\npolarity = (positive-negative)/(positive+negative)",
        caption = "Source: Federal Reserve Bank of Minneapolis\nShaded areas NBER recessions") +
    geom_rect(data=recessions.df,
                    inherit.aes = F,
                aes(xmin = Peak,
                    xmax = Trough,
                    ymin = -Inf,
                    ymax = +Inf),
                    fill='darkgray',
                    alpha=0.5)
```



Even this raw data, without any adjustments, appears to track recessions well. It's not clear if this is a leading indicator, lagging indicator, or neither. Note that the moving averages may lose some of the power in terms of matching recessions.

We can also plot the sentiment by bank using the `facet_wrap` object in `ggplot2`. This yields interesting results: the authors at the different Federal Reserve banks have different scales, thus changing the composition of the data when each bank (and the national summary) is given equal weight.



Scaling

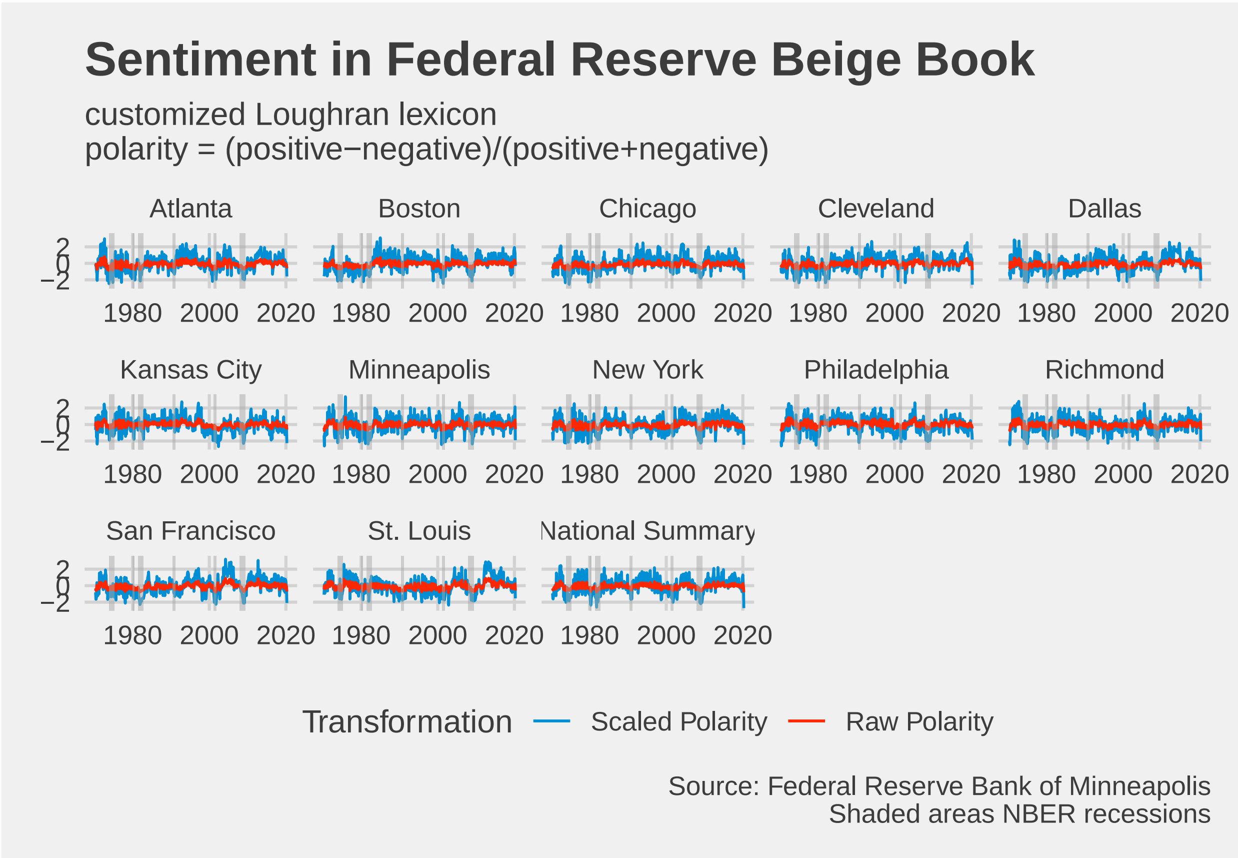
To adjust for the differences in each bank's Beige Book reports, I applied a simple scaling function. No, seriously, very simple. The range of values is larger but adjusted to the mean and standard deviation of each sample, which is bank-specific.

Here is the code and plot:

```
fed_sentiment_bank <-
    fed_text_all %>%
    inner_join(get_sentiments("loughran")) %>% # or bing
    # inner_join(get_sentiments("bing")) %>%
    count(report, year, date, bank, sentiment) %>%
    pivot_wider(names_from = sentiment,
                values_from = n,
                values_fill = 0) %>%
    mutate(sentiment = (positive - negative)/(positive+negative)) %>%
    group_by(bank) %>%
    mutate(sent_norm = scale(sentiment)) %>%
    ungroup() %>%
    mutate(date = as.Date(date, format = "%m/%d/%Y")) %>%
    filter(sentiment != 1) %>%
    filter(date != "2015-07-01") %>%
    select(date, bank, sent_norm, sentiment) %>%
    pivot_longer(-c(date, bank),
                names_to = "transformation",
                values_to = "value") %>%
```

```r
    mutate(transformation = as_factor(transformation))

ggplot(fed_sentiment_bank,
    aes(x = date, y = value, color = transformation)) +
    geom_line() +
    theme_fivethirtyeight() +
    scale_x_date(
        limits = as.Date(c("1970-01-01","2020-06-01")),
        date_labels = "%Y") +
    scale_color_fivethirtyeight(
        name = "Transformation",
        labels = c('Scaled Polarity', 'Raw Polarity')) +
    labs(x = "Beige Book Report (~8/year)",
        y = "polarity",
        title = "Sentiment in Federal Reserve Beige Book",
        subtitle = "customized Loughran lexicon\npolarity = (positive-negative)/(positive+negative)",
        caption = "Source: Federal Reserve Bank of Minneapolis\nShaded areas NBER recessions") +
    facet_wrap(~bank, scales = 'free_x', ncol = 5,
    labeller = as_labeller(c('at' = 'Atlanta', 'bo' = 'Boston',
                    'ch' = 'Chicago', 'cl' = 'Cleveland',
                    'da' = 'Dallas', 'kc' = 'Kansas City',
                    'mi' = 'Minneapolis', 'ny' = 'New York',
                    'ph' = 'Philadelphia', 'ri' = 'Richmond',
                    'sf' = 'San Francisco', 'sl' = 'St. Louis',
                    'su' = 'National Summary'))) +
    geom_rect(data = recessions.df,
                    inherit.aes = F,
                aes(xmin = Peak,
                    xmax = Trough,
                    ymin = -Inf,
                    ymax = +Inf),
                    fill='darkgray',
                    alpha=0.5)
```

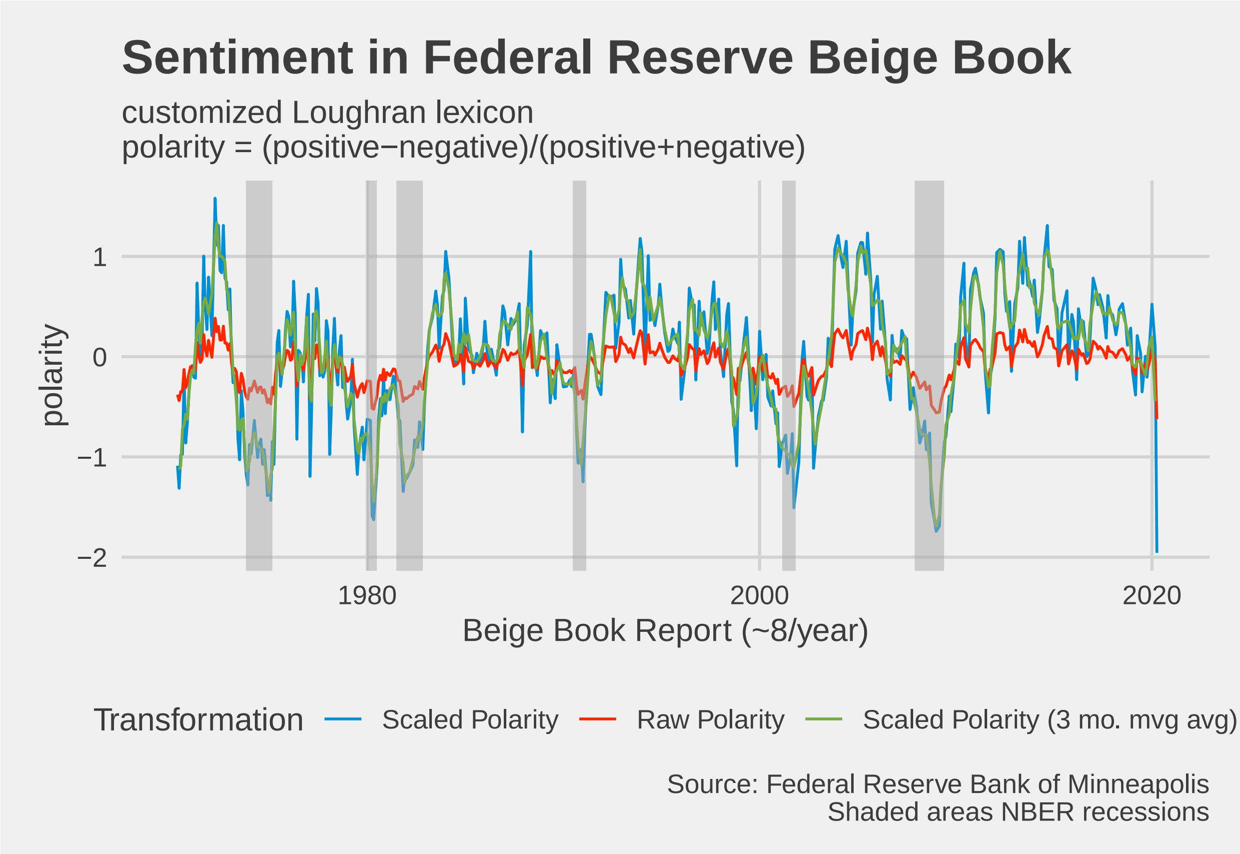


We can now put the scaling and moving averages together:

```
fed_sentiment_scale <-
    fed_text_all %>%
    inner_join(get_sentiments("loughran")) %>% # or bing
    # inner_join(get_sentiments("bing")) %>%
    count(report, year, date, bank, sentiment) %>%
    pivot_wider(names_from = sentiment,
                values_from = n,
                values_fill = 0) %>%
    mutate(sentiment = (positive - negative)/(positive+negative)) %>%
    group_by(bank) %>%
    mutate(sent_norm = scale(sentiment)) %>%
    ungroup() %>%
    mutate(date = as.Date(date, format = "%m/%d/%Y")) %>%
    filter(sentiment != 1) %>%
    filter(date != "2015-07-01") %>%
    select(date, sent_norm, bank, sentiment) %>%
    group_by(date) %>%
    summarize(norm_mean = mean(sent_norm),
            sent_mean = mean(sentiment)) %>%
    mutate(sent_norm_mean_ma = rollmean(norm_mean,
            k = 3,
            fill = NA)) %>%
    mutate(sent_mean_ma = rollmean(sent_mean,
            k = 3,
            fill = NA)) %>%
```

```r
    pivot_longer(-date,
                 names_to = "transformation",
                 values_to = "value") %>%
    mutate(transformation = as_factor(transformation))

ggplot(filter(fed_sentiment_scale,
transformation == "sent_norm_mean_ma" | transformation == "norm_mean" | transformation == 'sent_mean'),
    aes(x = date, y = value, color = transformation)) +
    geom_line() +
    theme_fivethirtyeight() +
    scale_x_date(limits = as.Date(c("1970-01-01","2020-06-01")),
                 date_labels = "%Y") +
    scale_color_fivethirtyeight(
        name = "Transformation",
        labels = c('Scaled Polarity',
                   'Raw Polarity',
                   'Scaled Polarity (3 mo. mvg avg)')) +
    labs(x = "Beige Book Report (~8/year)",
        y = "polarity",
        title = "Sentiment in Federal Reserve Beige Book",
        subtitle = "customized Loughran lexicon\npolarity = (positive-negative)/(positive+negative)",
        caption = "Source: Federal Reserve Bank of Minneapolis\nShaded areas NBER recessions") +
    geom_rect(data = recessions.df,
                   inherit.aes = F,
                 aes(xmin = Peak,
                     xmax = Trough,
                     ymin = -Inf,
                     ymax = +Inf),
                     fill='darkgray',
                     alpha=0.5) +
    theme(axis.title = element_text())
```



Are the transformations we applied actually any better at tracking recessions? Comparing to GDP growth rates (which we will do in the next section) will give more clues as to how we can use this quantification of the Beige Book. The scaled polarity tracks the 2007-09 Great Recession as a more protracted or more serious recession than those in the past, which backs up conventional wisdom at data. The moving averages (unsurprisingly) get rid of some of the noise without removing the general effects. COVID-19 also has clear negative impacts on Beige Book sentiment. From eyeballing, it seems that the transformations smooth out some issues with the data.

in Python

I downloaded the Loughran-McDonald Sentiment World list as a `csv`, which is available here. Tim Loughran and Bill McDonald include other resources, such as linked papers and a parser for assigning sentiments to each file in a folder; however, our data is not cleanly separated into files (although it could be). I resort to constructing a dictionary as a tidy dataframe.

Dictionary Construction

This chunk separates different columns of the word list and then appends them into our preferred dictionary format that we can use for a join.

```
lm_dict = (pd.read_csv('C:\\Users\\darre\\Documents\\_econ\\fedspeak\\sentiment analysis\\loughran_mcdon
                .fillna('')
                .apply(lambda x: x.astype(str))
                .apply(lambda x: x.str.lower())
                # .to_dict()
                )
```

```python
positive = lm_dict['positive'].tolist()
positive = (pd.DataFrame(list(filter(None, positive)), columns = ['word'])
                .assign(sentiment = 'positive'))

negative = lm_dict['negative'].tolist()
negative = (pd.DataFrame(list(filter(None, negative)), columns = ['word'])
                .assign(sentiment = 'negative'))

uncertainty = lm_dict['uncertainty'].tolist()
uncertainty = (pd.DataFrame(list(filter(None, uncertainty)), columns = ['word'])
                .assign(sentiment = 'uncertainty'))

weak_modal = lm_dict['weak_modal'].tolist()
weak_modal = (pd.DataFrame(list(filter(None, weak_modal)), columns = ['word'])
                .assign(sentiment = 'weak_modal'))

strong_modal = lm_dict['strong_modal'].tolist()
strong_modal = (pd.DataFrame(list(filter(None, strong_modal)), columns = ['word'])
                .assign(sentiment = 'strong_modal'))

constraining = lm_dict['constraining'].tolist()
constraining = (pd.DataFrame(list(filter(None, constraining)), columns = ['word'])
                .assign(sentiment = 'constraining'))

lm_dict = (positive.append(negative, ignore_index = True)
        .append(uncertainty, ignore_index = True)
        .append(weak_modal, ignore_index = True)
        .append(strong_modal, ignore_index = True)
        .append(constraining, ignore_index = True)
)
lm_dict.head(20)
```

```
##                word sentiment
## 0              able  positive
## 1         abundance  positive
## 2          abundant  positive
## 3         acclaimed  positive
## 4        accomplish  positive
## 5      accomplished  positive
## 6      accomplishes  positive
## 7     accomplishing  positive
## 8    accomplishment  positive
## 9   accomplishments  positive
## 10          achieve  positive
## 11         achieved  positive
## 12      achievement  positive
## 13     achievements  positive
## 14         achieves  positive
## 15        achieving  positive
## 16       adequately  positive
## 17      advancement  positive
## 18     advancements  positive
## 19         advances  positive
```

Get Sentiments

I use `merge` to join our data and our dictionary to create a column with our assigned sentiments.

```
def get_sentiment(
    df, # dataframe of words
    dict # dataframe of dictionary that you want to use
):
    return (df
            .merge(dict)
            .sort_values(by = ['report'], ignore_index = False))
```

Here, I calculate the polarity score for each report.

```
fed_sentiment = get_sentiment(fed_text_all, lm_dict)

fed_sentiment = (fed_sentiment
    .groupby(['report','date','url','year','month','bank','sentiment'])
    .count()
    .unstack(-1, fill_value = 0)
    )

# get rid of some wacky indexing
fed_sentiment.reset_index(inplace = True)

# rename columns
fed_sentiment.columns = ['report', 'date', 'url', 'year', 'month',
'bank', 'constraining', 'negative', 'positive', 'strong_modal',
'uncertainty', 'weak_modal']

fed_sentiment = (fed_sentiment
    .drop(['url'], axis = 1)
    .assign(polarity = lambda x: (fed_sentiment['positive']-fed_sentiment['negative'])/
    (fed_sentiment['positive']+fed_sentiment['negative']))
)

fed_sentiment.head(20)
```

```
##          report      date  year  ...  uncertainty weak_modal  polarity
## 0    1970-05-at  5/1/1970  1970  ...           10          2 -0.212121
## 1    1970-05-bo  5/1/1970  1970  ...            9          7 -0.520000
## 2    1970-05-ch  5/1/1970  1970  ...           20          8 -0.384615
## 3    1970-05-cl  5/1/1970  1970  ...            9          5 -0.241379
## 4    1970-05-da  5/1/1970  1970  ...           15         11 -0.411765
## 5    1970-05-kc  5/1/1970  1970  ...           10          1 -0.176471
## 6    1970-05-mi  5/1/1970  1970  ...           11          5 -0.400000
## 7    1970-05-ny  5/1/1970  1970  ...            4          3 -0.333333
## 8    1970-05-ph  5/1/1970  1970  ...            8          3 -0.555556
## 9    1970-05-ri  5/1/1970  1970  ...           11          8 -0.600000
## 10   1970-05-sf  5/1/1970  1970  ...            9          1 -0.319149
## 11   1970-05-sl  5/1/1970  1970  ...            7          3 -0.440000
## 12   1970-05-su  5/1/1970  1970  ...            2          0 -0.333333
## 13   1970-06-at  6/1/1970  1970  ...            4          3 -0.142857
## 14   1970-06-bo  6/1/1970  1970  ...           13          6 -0.454545
## 15   1970-06-ch  6/1/1970  1970  ...            9          5 -0.500000
## 16   1970-06-cl  6/1/1970  1970  ...           10          2 -0.388889
```

```
## 17  1970-06-da  6/1/1970  1970  ...          16          13 -0.534884
## 18  1970-06-kc  6/1/1970  1970  ...           6           5  0.225806
## 19  1970-06-mi  6/1/1970  1970  ...          11           8 -0.757576
##
## [20 rows x 12 columns]
```

The BB Index vs Economic Indicators

Next, I compare the BB index that I have constructed with the steps above against economic indicators. My hypothesis is that the BB index can be used as a helpful quantitative indicator with some qualitative dimensions, e.g. understanding inflation or liquidity. Furthermore, the BB index may be useful for understanding changes across Federal Reserve districts, for which not a lot of data exists. Looking up data segmented by FRB district on FRED returns limited results (versus data segmented by state, for example).

GDP Growth

First, I collect the data from FRED using the `tidyquant` package, which is incredibly versatile and integrated with the `tidyverse` ecosystem. I download two different measures for GDP that are similar. A191RL1Q225SBEA is Real GDP with units of Percent Change from Preceding Period, Seasonally Adjusted Annual Rate. It is released quarterly. A191RO1Q156NBEA is Real GDP with units of Percent Change from Quarter One Year Ago, Seasonally Adjusted, also released quarterly. I name the two series `pch` and `pca`, respectively.

```r
## ---- INDICATOR COMPARISON

# US real GDP
gdp_tickers <- c("A191RL1Q225SBEA", "A191RO1Q156NBEA")
gdp <- tq_get(gdp_tickers, get = "economic.data", from = "1970-01-01") %>%
    mutate(series = case_when(symbol == "A191RL1Q225SBEA" ~ "gdp_pch",
                              symbol == "A191RO1Q156NBEA" ~ "gdp_pca")) %>%
    select(-symbol) %>%
    rename(value = price)
```

Second, I scale the GDP estimates, which is important because the BB index is also scaled. This makes comparison on the same graph easier without using multiple y-axes, although it becomes less meaningful in terms of real percentage estimates. Of course, it is possible to "de-scale" these data later to obtain the GDP estimate in percent. The data that is used in the regression is not scaled, but the following chunk of code is included because it is used in visualizations. The data and date transformations are discussed in more detail in the next step.

```r
gdp_scale <- gdp %>%
    group_by(series) %>%
    mutate(value = scale(value)) %>%
    ungroup()

sent_gdp_scale <-
    fed_sentiment_scale %>%
    filter(transformation == "sent_norm_mean_ma" | transformation == "norm_mean") %>%
    mutate(series = case_when(transformation == "norm_mean" ~ "polarity",
                              transformation == "sent_norm_mean_ma" ~ "polarity_ma")) %>%
    select(-transformation) %>%
    mutate(quarter = quarter(date,
            with_year = T,
            fiscal_start = 1)) %>%
    mutate(q_date = as.Date(as.yearqtr(as.character(quarter),
                              format = "%Y.%q"))) %>%
    group_by(quarter, series) %>%
```

```r
    mutate(q_value = mean(value)) %>%
    distinct(q_value, .keep_all = T) %>%
    ungroup() %>%
    select(-value, -date, -quarter) %>%
    rename(date = q_date) %>%
    rename(value = q_value) %>%
    bind_rows(gdp_scale)
```

Third, I perform some transformations on the GDP data and merge it with the BB index data. Most importantly, the quarterly dates on the BB index have to matched with the quarterly dates on the GDP data. This is done with the `lubridate::quarter()` function. The conversion back to R date objects is done with `zoo::as.yearqtr()`. GDP data is released quarterly, so I match the BB index to GDP quarters by taking the mean of the BB index values within quarters.

```r
sent_gdp <-
    fed_sentiment_scale %>%
    filter(transformation == "sent_norm_mean_ma" | transformation == "norm_mean") %>%
    mutate(series = case_when(transformation == "norm_mean" ~ "polarity",
                        transformation == "sent_norm_mean_ma" ~ "polarity_ma")) %>%
    select(-transformation) %>%
    mutate(quarter = quarter(date,
            with_year = T,
            fiscal_start = 1)) %>%
    mutate(q_date = as.Date(as.yearqtr(as.character(quarter),
                        format = "%Y.%q"))) %>%
    group_by(quarter, series) %>%
    mutate(q_value = mean(value)) %>%
    distinct(q_value, .keep_all = T) %>%
    ungroup() %>%
    select(-value, -date, -quarter) %>%
    rename(date = q_date) %>%
    rename(value = q_value) %>%
    bind_rows(gdp)
```

Fourth, I convert the tibble with the GDP data and the BB index into the wide format so I can perform a regression.

```r
sent_gdp_wide <-
    sent_gdp %>%
    pivot_wider(
        names_from = series,
        values_from = value)
```

Finally, I run the regression. Obviously, this is an extreme simplification of understanding how GDP and the BB index co-move. However, I'm not looking for detailed information on how that happens, simply that there is some correlation between the two datasets. I choose the `polarity_ma` (moving average index) and the `gdp_pca` (percent change year over year) measures because these generate the highest R^2 values.

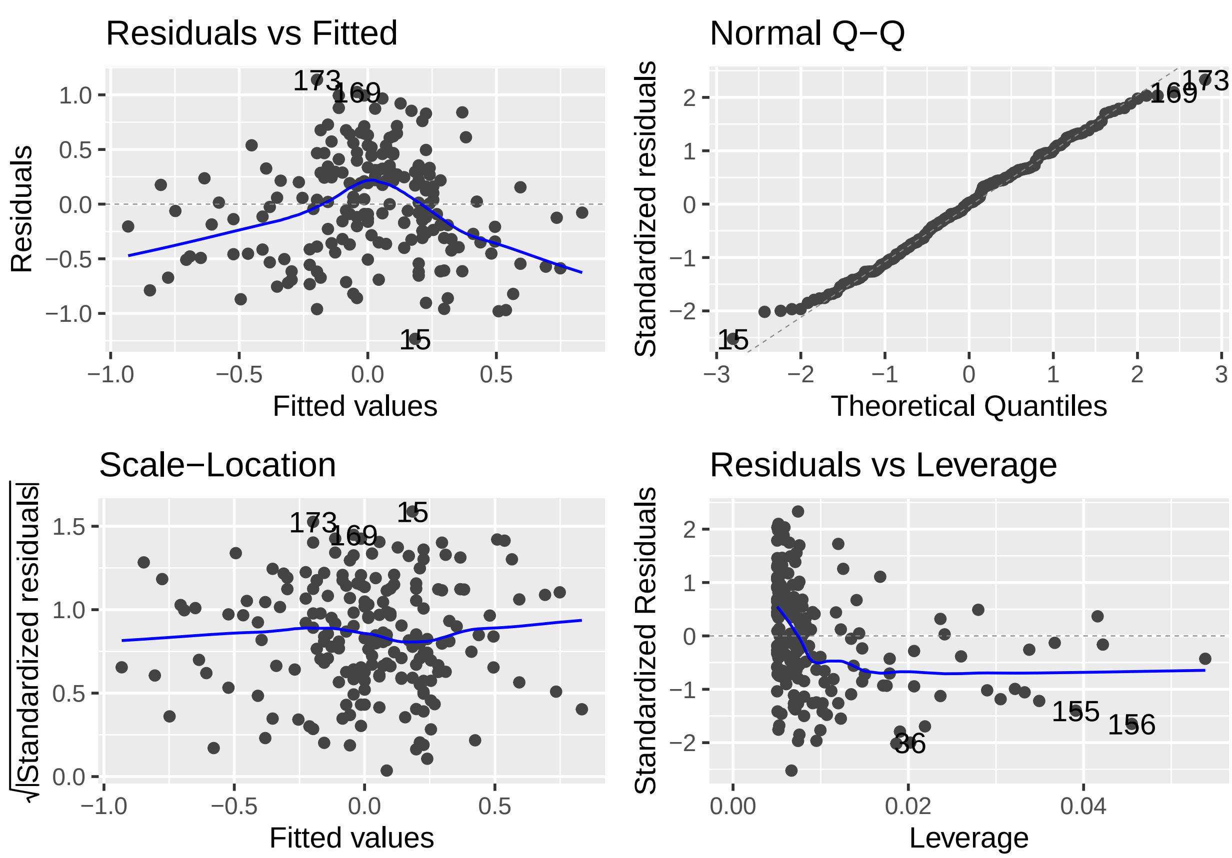
```r
lm_ma_gdp <- lm(polarity_ma ~ gdp_pca,
    data = sent_gdp_wide)

summary(lm_ma_gdp)
```

```
##
## Call:
## lm(formula = polarity_ma ~ gdp_pca, data = sent_gdp_wide)
```

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.2326 -0.3611  0.0130  0.3252  1.1379
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.38110    0.05690  -6.698 2.16e-10 ***
## gdp_pca      0.14126    0.01623   8.705 1.27e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4902 on 197 degrees of freedom
##   (3 observations deleted due to missingness)
## Multiple R-squared:  0.2778, Adjusted R-squared:  0.2741
## F-statistic: 75.78 on 1 and 197 DF,  p-value: 1.273e-15
```

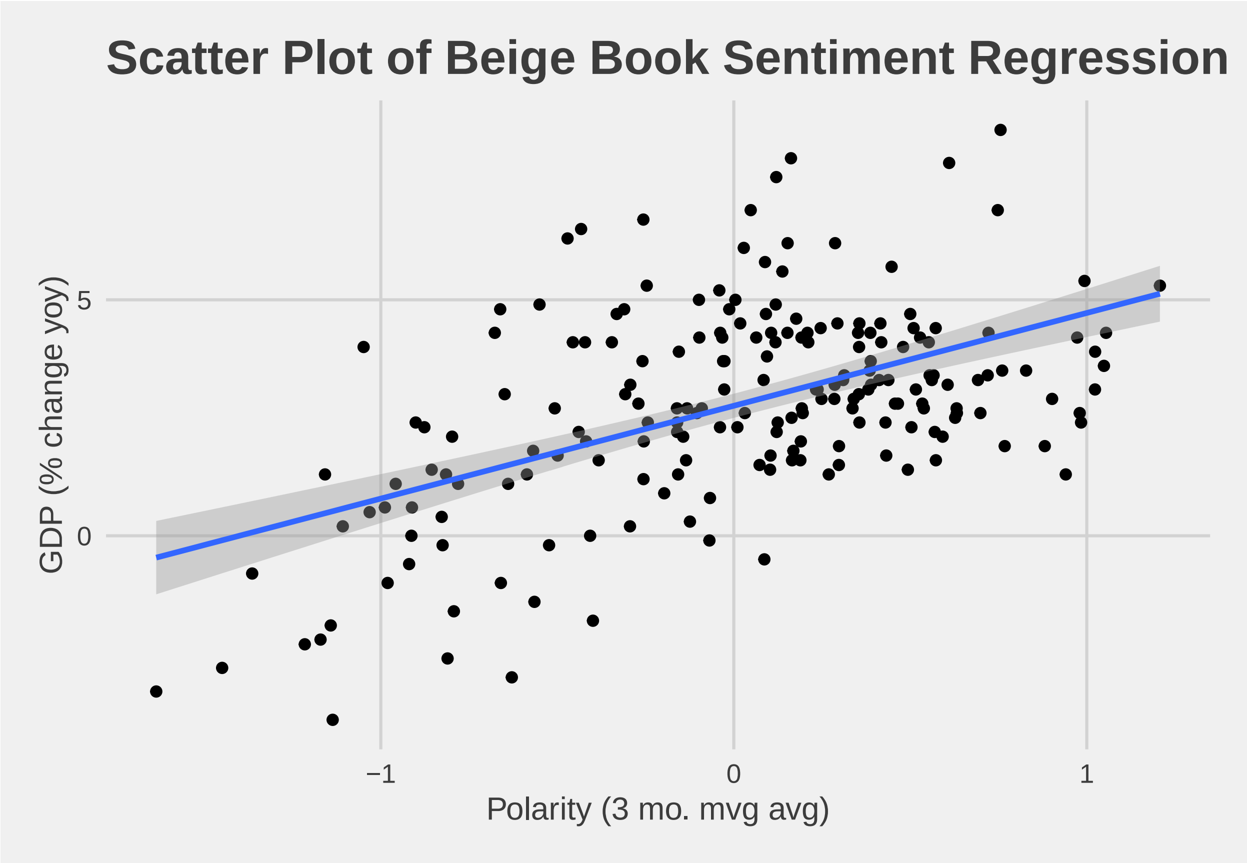
```r
autoplot(lm_ma_gdp)
```




```r
ggplot(select(sent_gdp_wide, date, polarity_ma, gdp_pca),
        aes(x = polarity_ma, y = gdp_pca)) +
    geom_point() +
    theme_fivethirtyeight() +
    geom_smooth(method = 'lm') +
    labs(x = 'Polarity (3 mo. mvg avg)',
        y = 'GDP (% change yoy)',
        title = 'Scatter Plot of Beige Book Sentiment Regression') +
```

```
    theme(axis.title = element_text())
```

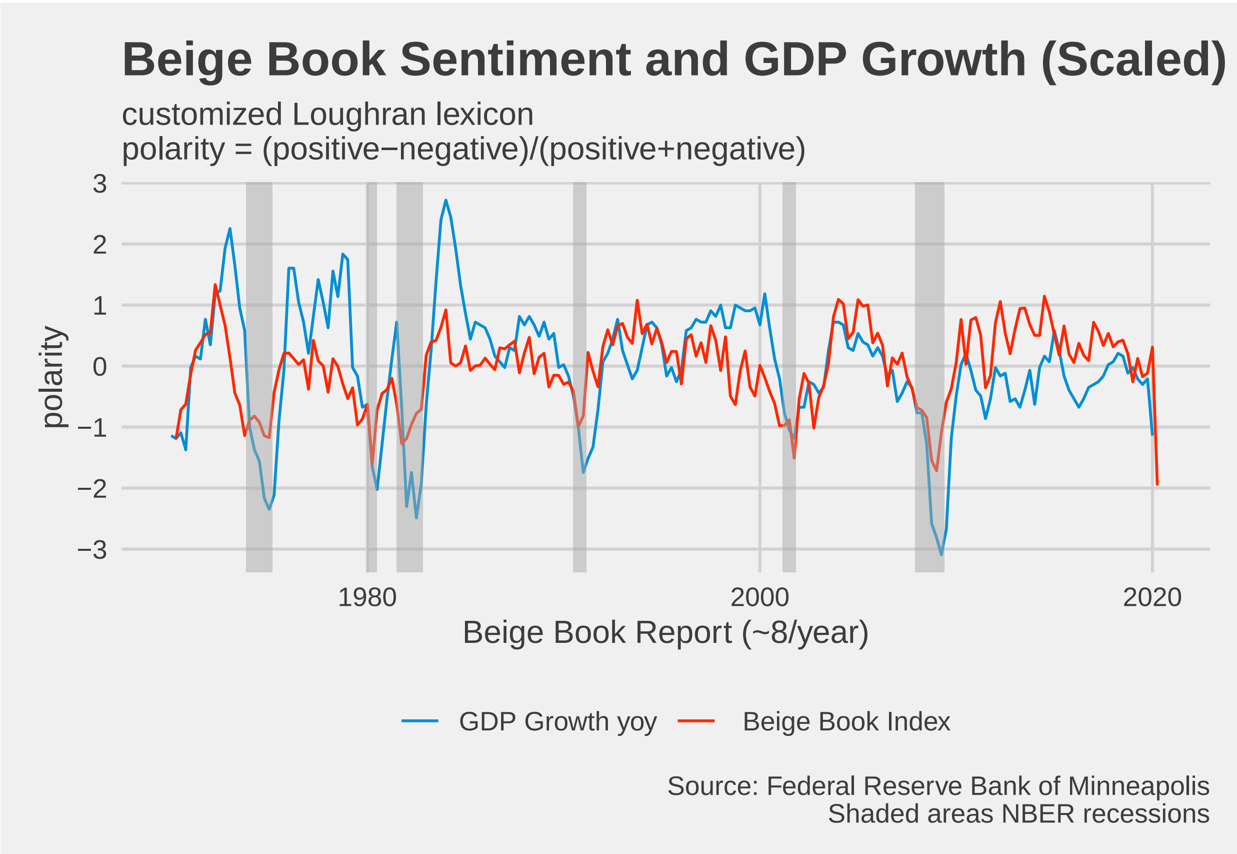
```
## `geom_smooth()` using formula 'y ~ x'
```



There is some evidence that the residuals may not be normally distributed, but with such a simple model, it's unclear if a different model would make much of an improvement. It appears that GDP is a significant predictor (with $p = 1.26 * 10^{-15}$) of the BB index measures; the BB index is also a significant predictor of GDP. With just the two variables, there is little to be said about the direction of effect. The models explain nearly 28% of the variation of the data, which is a decent chunk, considering that the BB index is a constructed measure. I've also included the scatter plot for convenience. Originally, I thought it would be helpful to include lagged regressions, but it appears that the GDP data and BB index match closely on date.

```
ggplot(filter(sent_gdp_scale, series == 'polarity' | series == 'gdp_pca'),
    aes(x = date, y = value, color = series)) +
    geom_line() +
    theme_fivethirtyeight() +
    scale_color_fivethirtyeight(
        name = '',
        labels = c('GDP Growth yoy', ' Beige Book Index')
    ) +
    scale_x_date(limits = as.Date(c("1970-01-01","2020-06-01")),
                 date_labels = "%Y") +
    labs(x = "Beige Book Report (~8/year)",
        y = "polarity",
        title = "Beige Book Sentiment and GDP Growth (Scaled)",
        subtitle = "customized Loughran lexicon\npolarity = (positive-negative)/(positive+negative)",
        caption = "Source: Federal Reserve Bank of Minneapolis\nShaded areas NBER recessions") +
```

```
    geom_rect(data = recessions.df,
                     inherit.aes = F,
                aes(xmin = Peak,
                     xmax = Trough,
                     ymin = -Inf,
                     ymax = +Inf),
                     fill='darkgray',
                     alpha=0.5)+
    theme(axis.title = element_text())
```



Forecasting

I first attempted to use the `fbprophet` package for some (very light and very automated) forecasting, but it didn't work well for this purpose, as business cycles are of variable length. Instead, I used the `nowcasting` package, which is available on CRAN and uses a dynamic factor method for estimation based on Giannone, Reichlin, and Small (2008).

Our exercise is similar to what the New York Fed does in their weekly nowcasting report, although limited access to data constrains our forecast's timeliness. In addition, the specific method chosen for nowcasting from the package does not use Kalman filtering, although other methods in the package do utilize Kalman filtering.

Dynamic Factor Model

The dynamic factor model specification is given by:

$$x_t = \mu + \Lambda f_t + \epsilon_t$$

(1)

$$f_t = \sum_{i=1}^{p} A_i f_{t-1} + B u_t$$

(2)

where $u_t \sim i.i.d.N(0, I_q)$. x_t is a vector of N monthly time series and f_t are common factors.

We use the Expectation-Maximization (EM) method, which the `nowcasting` package authors describe in this paper in *The R Journal*. Importantly, the EM method can adjust for arbitrary patterns in missing values, which is a key issue of using the Beige Book as an indexing measure. There is no clear pattern as to which eight months out of the year the Beige Book will be released. In addition, the factors do not need to be global. Our data has four blocks of factors: global, soft, real, and labor. Note that on the EM model, the error term ϵ_t is defined as an AR(1) process, but we have to set the number of shocks to the factors `q` equal to the number of factors `r`.

We can rewrite equation (1) above by first rewriting Λ and f_t as a matrix of our four factors:

$$\Lambda = \begin{pmatrix} \Lambda_{S,G} & \Lambda_{S,S} & 0 & 0 \\ \Lambda_{R,G} & 0 & \Lambda_{R,R} & 0 \\ \Lambda_{L,G} & 0 & 0 & \Lambda_{L,L} \end{pmatrix}$$

(3)

$$f_t = \begin{pmatrix} f_t^G \\ f_t^S \\ f_t^R \\ f_t^L \end{pmatrix}$$

(4)

Plugging this into (1), we have

$$x_t = \mu + \begin{pmatrix} \Lambda_{S,G} & \Lambda_{S,S} & 0 & 0 \\ \Lambda_{R,G} & 0 & \Lambda_{R,R} & 0 \\ \Lambda_{L,G} & 0 & 0 & \Lambda_{L,L} \end{pmatrix} \begin{pmatrix} f_t^G \\ f_t^S \\ f_t^R \\ f_t^L \end{pmatrix} + \epsilon_t$$

The package uses the Expectation_Maximization algorithm to recursively estimate the model by maximum likelihood estimation until the log-likelihood function is less than 10^{-4}.

To do

- Regional economic comparison