Can I Hear Me Now?
Echo Chambers in Online Discourse
Process Article

This article covers the methodology used in this project by walking through key elements of the code, and describing the steps taken all the way from initial data ingestion to the final outputs of the project. Results, implications, and general discussion of the project can be found in the accompanying summary article which focuses on the results of this project.

To briefly recap, the purpose of this project is to measure to what degree, if any, two Canada political forums on popular discussion site Reddit have become polarized over the years. My hypothesis is that if a machine learning algorithm can more easily distinguish between two subreddits as time goes on, the content of those subreddits is becoming more distinct.

To accomplish this task, I broke the project into four rough 'segments', and this article will discuss each in turn. They are as follows:

1. **Data collection**: the scraping of comment data from each of the subreddits to allow for analysis of differences between the communities.

2. **Initial data cleaning**: removing non-contributing comments and other such noise from the dataset.

3. **Exploration**: drawing out trends and other interesting characteristics of the data in its cleaned form.

4. **Classification**: selecting and implementing the appropriate machine learning algorithm to try to classify comments.

Data Collection

To adequately answer the question of whether two subreddits have become more or less polarized over time, one must first collect data from the subreddits in question. In a business context, downloading **all** of the information may be impractical as there are through-put limitations, making large downloads time prohibitive. However, on this project, the time implications meant only a slight delay, so I elected to go for a complete download rather than pursuing some form of sampling.

At minimum, one must acquire the text of the comments themselves, the date they are posted, as well as what subreddit they are made on. In addition, there are a variety of other features available that may be useful to the analysis, such as:

1. A comment's score (the total number of upvotes minus the total number of downvotes).

2. The author of the comment.

3. The title of the post the comment was made on.

4. The link provided as part of the post the comment was made on.

Fortunately, APIs exist to allow us to scrape this data off of the web. I conduct my work for this project in Python, and use two APIs: Reddit's own API, and the Pushshift API. Pushshift is an archiver of Reddit content, though comment level information from Pushshift is not always fully up to date.

Each of these two resources have different strengths but working together are able to acquire all of the essential information, as well as the "bonus" features listed in items one to four above. I describe each below:

1. **Reddit's API** allows for the downloading of both post and comment level data from a chosen subreddit, but in its current form **does not** allow for searching between two specified dates. Additionally, the API has throughput restrictions, making large data-pulls costly in terms of time.

2. **Pushshift's API** also allows for the downloading of both post and comment level data from a chosen subreddit, and **does** allow for searching between two specified dates, an important feature for this project as I am seeking to measure the level of polarization **over time**. Pushshift also has throughput restrictions, though they are not as severe. One major drawback of the Pushshift API is that it is only as good as whatever information is currently archived, meaning that in some cases comments are missed.

Combining these two API's struck me as the best course for this project, as Pushshift would allow me to seek out all comments made within a specified timeframe, while Reddit's API would ensure that I was getting complete and up to date comment information.

Thus, the data collection proceeded in two steps:

1. Using Pushshift, I collected a list of specific posts to query, alongside specific **post** level information.

2. Using Reddit's API, I took the list of specific posts from Pushshift, and downloaded all comments from these posts.

Step 1: Collect a list of posts to query

After importing the necessary packages, I defined a function to actually run the query.

```
def PushShift_Query(after: int, before: int, sub: str) -> list:
    """
    Queries the Pushshift API for a list of posts to extract data from.

    Args:
        after: The date after which to search for posts
        before: The date before which to search for posts.
        sub: The subreddit to search for posts on.

    Returns:
        A list of dictionaries containing post information.
```

```
    """
    link = 'https://api.pushshift.io/reddit/search/submission/?
after='+str(after)+'&before='+str(before)+'&subreddit='+str(sub)+'&size=100'
    r = requests.get(link)
    print(link)
    print('Start Date ' + str(datetime.fromtimestamp(after)))
    try:
        return json.loads(r.text)['data']
    except ValueError:
        print("JSON Decode error - trying again")
        return "try again"
```

Here, the variables 'after', 'before', and 'sub' define the range of dates to search between, and the subreddit to search on. The maximum size of a return is 100 posts. As I was scraping the data for this project, I noticed that the JSON data would fail to download sporadically, creating an error. I could not reliably reproduce this error, and often simply passing the same query again would result in a successful download. Thus, the code block above contains a 'try' clause to address this situation.

Afterwards, I defined a function that collects information from all the posts returned by the above query.

```
def collectPostData(post: dict) -> list:
    """
    Collects a specific list of values from a dictionary

    Args:
        post: The dictionary to extract values from.

    Returns:
        A list of values from the dictionary.
    """
    post_id = post['id']
    title = post['title']
    url = post['url']
    date = datetime.fromtimestamp(post['created_utc'])
    author = post['author']
    score = post['score']
    permalink = post['permalink']
    sub = post['subreddit']
    comments = post['num_comments']
    return [post_id, title, url, date, author, comments, score, permalink, sub]
```

The above code extracts information from the results of the previous function, creating a list that can be fed as values into a dictionary that will be called "poststats". Poststats keys are the individual post IDs, which can later be used to instruct

the Reddit API which posts to scrape comments from. Each key has a list of associated values that collectively represent:

- the title of the post;
- any url it links to;
- the date it was published;
- the post's author;
- the number of on the post comments;
- the post's net upvote/downvote score;
- a link to the post itself; and
- the subreddit it comes from.

Next, I brought these two functions together to collect all posts between January 1st 2015, and September 30th 2020, the timeframe for my analysis. Additionally, this function populates the poststats dictionary.

```
subs = ["CanadaPolitics", "canada"]
poststats = {}

#Scraping the post level data
for i in subs:
    after = 1420070400
    before = 1601424000
    pull = PushShift_Query(after, before, i)
    while len(pull)>0:
        print('End Date ' + str(datetime.fromtimestamp(pull[-1]['created_utc'])))
        for post in pull:
            poststats[collectPostData(post)[0]] = collectPostData(post)[1:]
        after = pull[-1]['created_utc']
        pull = PushShift_Query(after, before, i)
        while pull == "try again":
            pull = PushShift_Query(after, before, i)

    permalink = post['permalink']
    sub = post['subreddit']
    comments = post['num_comments']
    return [post_id, title, url, date, author, comments, score, permalink, sub]
```

This loop iterates through both of the selected subreddits, and first establishes the 'after' date as January 1st 2015 in Unix timestamp form; the 'before' assignment is the same, but for September 30th 2020. The loop then executes the Pushshift query I have already defined to collect the first 100 posts in the time frame and runs the data collection function on each post in turn, populating the 'poststats' dictionary.

However, there are well over 100 posts in the relevant timeframe, even though the Pushshift query is capped at 100 posts. Thus, after completing the first 100 posts, the loop updates the "after" date to be the date of the last post already downloaded, and then repeats. This ensures that the loop picks up right where it left off, and continues until it has covered the entire time period. Once there are no more posts between the 'after' date and the 'before' date, the loop finishes.

Afterwards I ran a quick sanity check, to make sure that the number of posts in the dictionary is the same as the number of **unique** posts in the dictionary. This ensures that no posts have been duplicated. Lastly, as this is a large and somewhat time-consuming download, I saved the information so that I could pick it up later if needed.

```
print(len(poststats.keys()) == len(set(poststats.keys())))

postcount = len(poststats)

save = pd.DataFrame.from_dict(poststats, orient = 'index')
save.to_csv('posts.csv')
```

Step 2: Download all comments from the list of posts

First, I initialized the Reddit scraper, and prepared to download the comments. The code I have listed below redacts my account specific information.

```
reddit = praw.Reddit(client_id='REDACTED',
                     client_secret='REDACTED',
                     user_agent='REDACTED')

remaining = [key for key in poststats]

comments = []
empties = []
unable = []
counter = 0
```

In addition to initializing the scraper (here called 'reddit'), I established a few variables that were useful during the comment download:

1. **remaining** begins as a list of all the keys downloaded in the previous step but will be updated to remove post IDs that have already been queried as the download proceeds.

2. **comments** holds the output of the download; the comment level information from each post.

3. **empties** holds a list of all posts which have no comments.

4. **unable** holds a list of any posts that the API was unable to query successfully.

5. **counter** tracks how many posts have been queried.

Afterwards, I began the download.

```
download = [key for key in remaining]
for post_id in download:
    counter +=1
    if counter % (round(postcount/20)) == 0:
        print("{:.2%}".format(counter/postcount))
    post = reddit.submission(id=post_id)
    try:
        post.comments.replace_more(limit=None)
        if len(post.comments.list()) == 0:
            empties.append(post_id)
            remaining.remove(post_id)
            pass
        else:
            for comment in post.comments.list():
                comments.append([comment.body, comment.score, post_id, poststats[post_id]
[0], comment.author, poststats[post_id][7], comment.created_utc, poststats[post_id][1]])
            remaining.remove(post_id)
    except:
        unable.append(post_id)
        remaining.remove(post_id)
```

This loop iterates through all the post IDs established in the Pushshift Query and sends the results to the appropriate list. After doing so, it purges that key from the 'remaining' list.

As there are millions upon millions of comments in the time frame, and Reddit limits the number of queries that can be made per minute, this download takes a great deal of time. Purging downloaded posts from the 'remaining' list results in the 'remaining' list containing only posts that **have not** been queried yet. This allows the user to pause the download for any reason. They can then simply rerun the above code block at a convenient time, and know that they will be picking up right where they left off.

Lastly, I finish by saving each of the relevant lists for later use, and run a quick sanity check to make sure that every post identified by the Pushshift API has been queried for comments.

```
remaining_df = pd.DataFrame(remaining, columns = ["post_id"])
empties_df = pd.DataFrame(empties, columns = ['post_id'])
df = pd.DataFrame(comments, columns = ['comment', 'score', 'post_id', 'post_title',
'author', 'sub', 'date', 'url'])

remaining_df.to_csv("remaining.csv")
empties_df.to_csv("empties.csv")
```

```
df.to_csv('reddit_dataset.csv')

#sanity check
print(len(set(df.post_id)) + len(set(remaining_df.post_id)) +
len(set(empties_df.post_id)) + len(unable) == len(poststats))
```

If the last line returns true, I know that no posts have been missed, and I can move onto the next stage in my analysis.

Initial Data Cleaning

First, as I discuss in the summary article, I needed to find a way to ensure that posts were about exactly the same topics across subreddits. This would help control for differentiation that had more to do with topic than with inherent differences in the userbase. If I could restrict the dataset to **just** articles that appeared in both subreddits, that topic based-differentiation wouldn't be a concern.

To do so I filtered based on the media link a specific post provided. In order for a post to be contained in my dataset, the **exact** same link had to be posted to both subreddits. I acknowledge that there may be stories about the same topic – but from different news providers posted to the two subreddits – and such posts have been lost using this method. That said, even with the exact match, I was left with approximately 8,500 posts per subreddit, and over 3.2 million user comments. As such, I considered that the dataset is of a sufficient size to still be viable for analysis.

The code I used to accomplish this filtering is contained below:

```
df= df[df['url'].isin(set(df[df['sub'] == 'r/Canada']
['url']).intersection(set(df[df['sub'] == 'r/CanadaPolitics']['url']))) == True]
```

Here I filtered on the 'posted link' column of the dataframe, using the intersection between the unique list of links posted to each subreddit. This removed any comments associated with non-matching posts.

Next, I needed to clean out comments that were spam or other forms of noise that did not contribute to the discussion.

```
#filter based on common link
df= df[df['url'].isin(set(df[df['sub'] == 'canada']['url']).intersection(set(df[df['sub']
== 'CanadaPolitics']['url']))) == True]

#Remove any comments from the AutoModerator
df = df[df['author'] != 'AutoModerator']

#Remove memes, repeated responses, and other such spam --> Comments must be unique
df['count'] = df.groupby('comment')['comment'].transform('count')
df['count'] = np.where(df['comment'] == '[removed]', 1, df['count'])
```

```
df = df[df['count'] <= 1]

#Convert the date column into datetime
df['date'] = df['date'].apply(lambda x: datetime.datetime.fromtimestamp(x)).dt.floor('d')

#Remove comments outside of time scope
df = df[(df['date'] > '2014-12-31') & (df['date'] < '2020-10-01')]

#Correct subreddit names
df['sub'] = df['sub'].str.replace('canada', 'r/Canada')
df['sub'] = df['sub'].str.replace('CanadaPolitics', 'r/CanadaPolitics')
```

Here I removed all comments made by the AutoModerator bot, as well as any comment that was not unique. This addressed comments which parrot popular memes or are simply one-word responses such as 'Yes' or 'Agreed'. This also cleaned out any moderator action comments such as 'Removed: Rule 3' as these are formulaic and often repeated.

The code above, however, does leave in comments that has been specifically removed by the moderator, as those comments are important for one of the metrics I am interested in: the removal rate over time. Removed comments were cleaned from the dataset prior to the machine learning analysis.

Afterwards, while the data download pulled all **posts** between January 1st 2015 and September 30th 2020, users can continue to **comment** after September 30th 2020, leading to some comments in my dataset being outside of my chosen timeframe. As such, I dropped all comments outside of the time scope to maintain consistency.

Finally, I corrected the subreddit names to be their proper names.

These operations reduced my dataset to just under 3 million unique comments.

Exploration

Next I needed to calculate some statistics about the data for initial exploration.

```
weekly_stats = df.groupby(['date', 'sub']).count()[['comment']]
x2 = df[df['comment'] == '[removed]'].groupby(['date', 'sub']).count()[['comment']]
x3 = df.groupby(['date', 'sub']).mean()[['score']]

for i in [x2, x3]:
    weekly_stats = weekly_stats.merge(i, left_index=True, right_index=True, how='outer')

weekly_stats = weekly_stats.rename(columns = {'comment_x':'num_comments',
                                              'comment_y':'num_removed',
                                              'score':'av_score'})
```

```
weekly_stats.reset_index(inplace = True)
weekly_stats = weekly_stats.sort_values('date', ascending = True)
```

The above code calculates the following three columns, and then organizes them into a new dataframe called 'weekly_stats':

1. **num_comments:** the number of comments posted on each sub on each day.
2. **num_removed:** the number of comments that were removed by moderators on each subreddit on each day.
3. **av_score:** the average net upvote/downvote (expressed as total upvotes minus total downvotes) on each subreddit on each day.

date	sub	num_comments	num_removed	av_score
2020-09-30 00:00:00	r/CanadaPolitics	503	66	5.74354
2020-09-30 00:00:00	r/Canada	3545	141	6.022
2020-09-29 00:00:00	r/CanadaPolitics	766	47	9.68799
2020-09-29 00:00:00	r/Canada	2021	79	12.3949
2020-09-28 00:00:00	r/CanadaPolitics	661	78	7.51891
2020-09-28 00:00:00	r/Canada	2453	88	7.42152
2020-09-27 00:00:00	r/CanadaPolitics	414	84	5.99275
2020-09-27 00:00:00	r/Canada	1600	16	7.11875

After that, on any day where no comments had been removed, num_removed would be a null value, instead of zero. So I filled all the null values with zero, and then calculated the percentage of removed comments on each day for each subreddit.

```
weekly_stats['num_removed'] = weekly_stats['num_removed'].fillna(0)
weekly_stats['% removed'] = (weekly_stats['num_removed']/weekly_stats['num_comments']) *
100
```

date	sub	num_comments	num_removed	av_score	% removed
2020-09-30 00:00:00	r/CanadaPolitics	503	66	5.74354	13.1213
2020-09-30 00:00:00	r/Canada	3545	141	6.022	3.97743
2020-09-29 00:00:00	r/CanadaPolitics	766	47	9.68799	6.13577
2020-09-29 00:00:00	r/Canada	2021	79	12.3949	3.90896

2020-09-28 00:00:00	r/CanadaPolitics	661	78	7.51891	11.8003
2020-09-28 00:00:00	r/Canada	2453	88	7.42152	3.58744
2020-09-27 00:00:00	r/CanadaPolitics	414	84	5.99275	20.2899
2020-09-27 00:00:00	r/Canada	1600	16	7.11875	1

Lastly, I converted all of these figures to rolling seven-day averages to help smooth out short term noise, and converted the date information into a form that Matplotlib will understand for charts.

```
weekly_stats.set_index('date', inplace = True)

for i in weekly_stats.columns[1:]:
    weekly_stats[i] = weekly_stats.groupby('sub')[i].transform(lambda row:
row.rolling('7d', min_periods = 1).mean())

weekly_stats['chart_dates'] = [mdates.date2num(i) for i in weekly_stats.index]
```

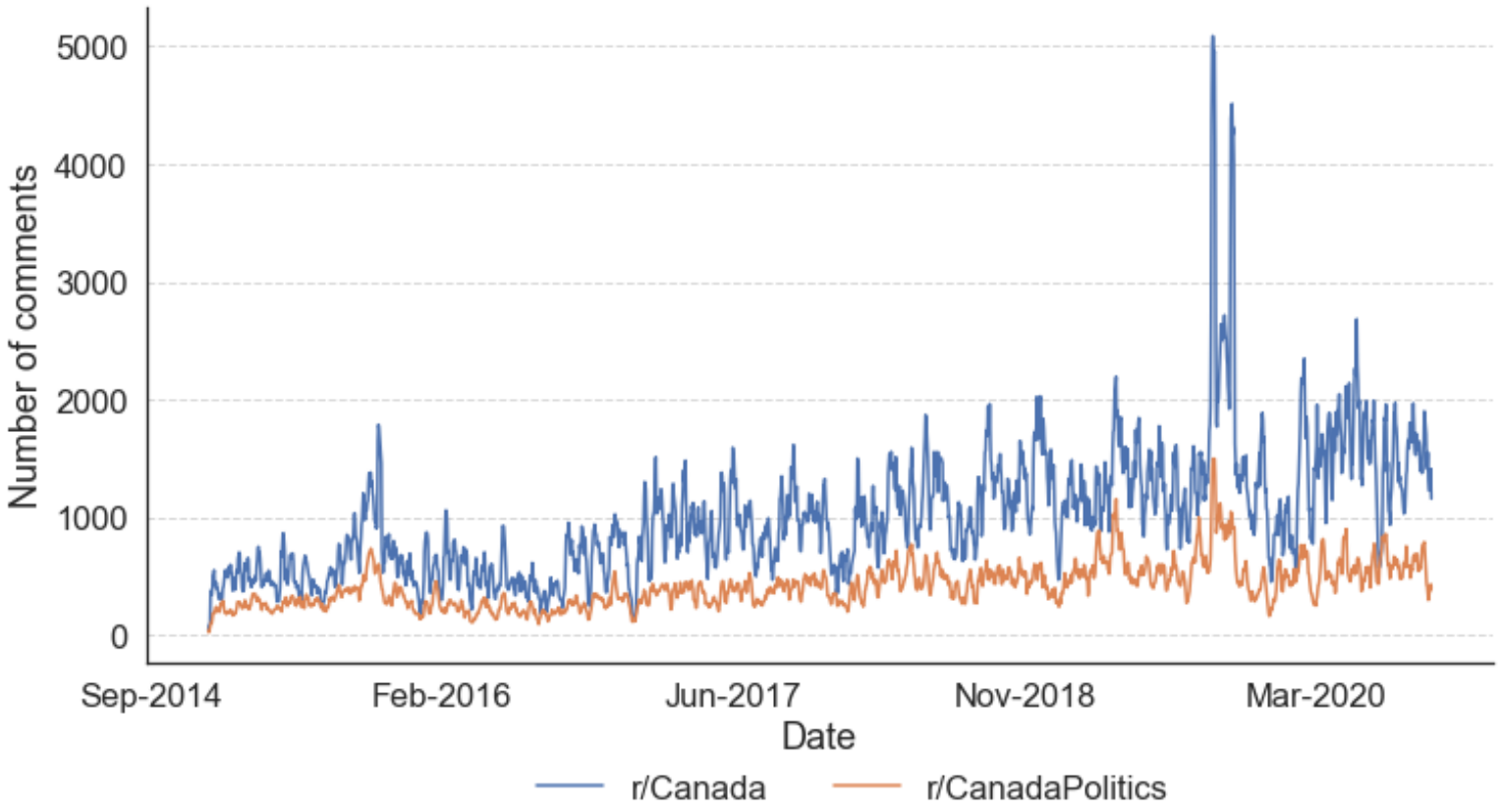
Basic Charts

Next I was ready to see some visualizations of these metrics over time! I began with the basics, a simple line chart showing the evolution of comments over time. The chart itself, and the code that produces it, is displayed below. For this, and all other visualizations displayed in this section, discussion of the actual content of the charts can be found in the summary article.

```
def line_plot(y_variable: str, y_title: str, categories: str, data: object, palette: str,
alpha: float):
    """
    Creates a line chart of the chosen variables over time

    Args:
        y_variable: The variable to plot on the vertical axis.
        y_title: The vertical axis title.
        categories: The variable to along which to split into different colour coded
lines.
        data: The input dataframe.
        palette: The Seaborn palette to use.
        alpha: Transparency of the lines.
    """
    fig, ax = plt.subplots(figsize = (12,6))
```

```
    sns.lineplot(x='chart_dates', y=y_variable, data = data, hue = categories, palette =
palette, alpha = alpha, ax = ax)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b-%Y'))
    ax.yaxis.grid(True, which='major', alpha = 0.75, linestyle = '--')
    ax.set(xlabel="Date", ylabel = y_title)
    handles, labels = ax.get_legend_handles_labels()
    ax.legend(handles=handles[1:], labels=labels[1:],bbox_to_anchor=(0.28, -0.15), loc=2,
borderaxespad=0., ncol = 3, frameon = False)
    sns.despine()

line_plot('num_comments', 'Number of comments', 'sub', weekly_stats, 'deep', 1)
```

Figure 1: Number of comments over time

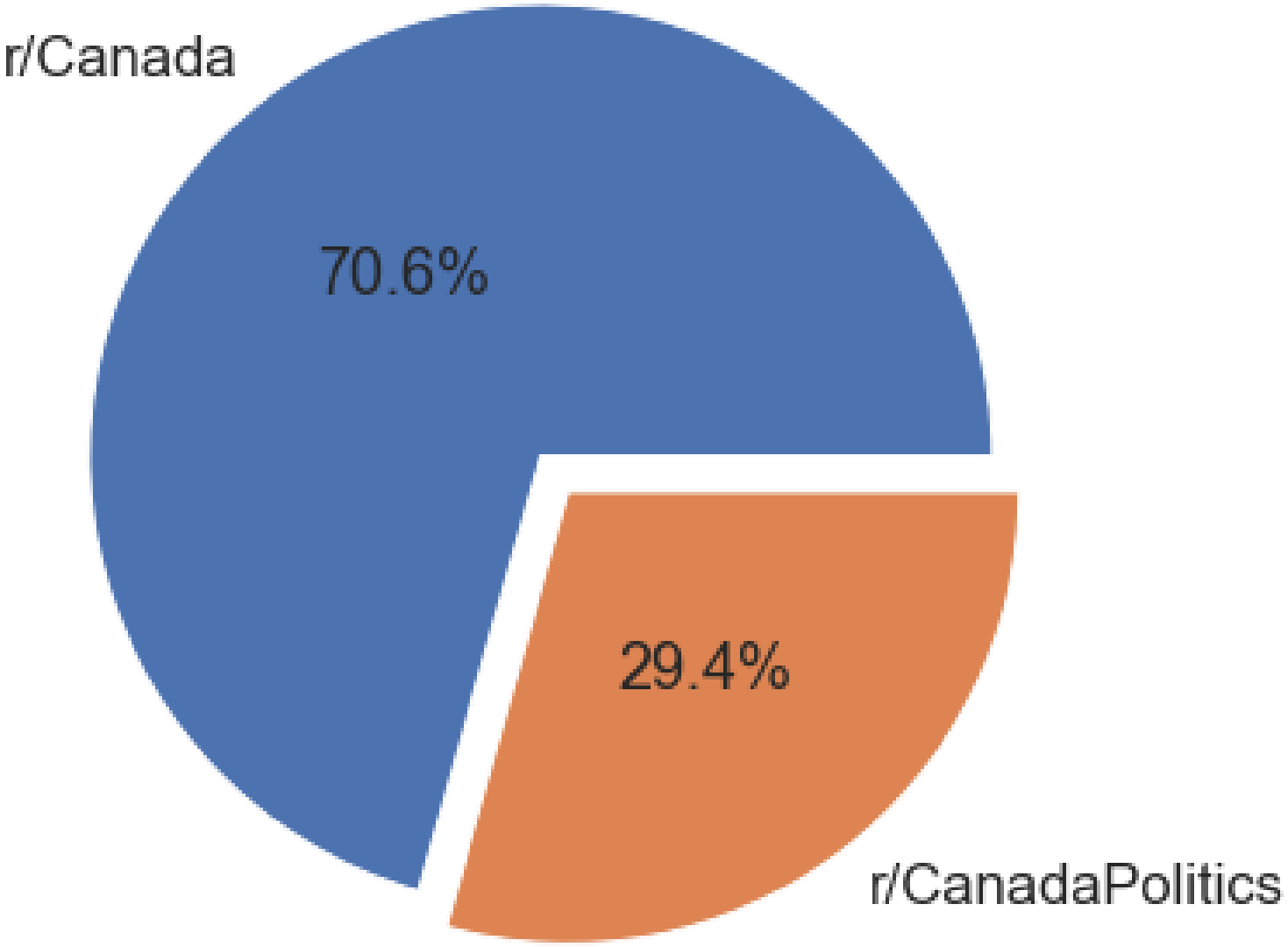


Next, I generated a pie chart to get a better understanding of the distribution of comments between the two subreddits.

```
pie_chart = df.groupby(['sub']).count()[['comment']]
pie_chart.reset_index(inplace = True)

pie, ax = plt.subplots(figsize = (12,6))
```

```
plt.pie(x=pie_chart['comment'], autopct="%.1f%%", explode=[0.05]*2,
labels=pie_chart['sub'], pctdistance=0.5)
```

Figure 2: Comment count by subreddit



With the basic visualizations complete, I was ready to move onto some more complex charts.

Trend plots and venn diagrams

First, I needed to generate scatterplots with an overall trend line for the average score, and removal rate metrics. The code to which accomplished this is displayed below.

```
def trend_plot(y_variable: str, y_units: str, y_title: str, categories: str,
category_values: list, data: object, palette: str, alpha: float, y_decimals: int):
    """
    Creates a scatter plot of the chosen variables with time as the X value.
    Layers a regression line for each variable on top of this scatter plot to help
display trends.

    Args:
        y_variable: The variable to plot on the vertical axis.
        y_units: The units to use for the y axis (percent or float)
        y_title: The vertical axis title.
        categories: The variable to along which to split into different colour coded
lines.
        category_values: All possible category values
        data: The input dataframe.
        palette: The Seaborn palette to use.
        alpha: Transparency of the lines.
        y_decimals: The number of decimal points to use on the y axis.
    """
    fig, ax = plt.subplots(figsize = (12,6))
    sns.scatterplot(x='chart_dates', y=y_variable, data = data, hue = categories, palette
= palette, alpha = alpha, ax = ax)
    sns.regplot(x='chart_dates', y=y_variable, data = data[data[categories] ==
category_values[0]], color = sns.color_palette(palette)[0], marker = '', ax = ax)
    sns.regplot(x='chart_dates', y=y_variable, data = data[data[categories] ==
category_values[1]], color = sns.color_palette(palette)[1],  marker = '', ax = ax)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b-%Y'))
    if y_units == 'percent':
        ax.yaxis.set_major_formatter(mtick.PercentFormatter(decimals = y_decimals))
    ax.yaxis.grid(True, which='major', alpha = 0.75, linestyle = '--')
    ax.set(xlabel="Date", ylabel = y_title)
    handles, labels = ax.get_legend_handles_labels()
    labels = category_values
    ax.legend(handles=handles[1:], labels=labels[:],bbox_to_anchor=(0.28, -0.15), loc=2,
borderaxespad=0., ncol = 3, frameon = False)
    sns.despine()

trend_plot("av_score", 'integer', 'Average net upvote score', 'sub', ['r/Canada',
'r/CanadaPolitics'], weekly_stats, 'deep', 0.2, 0)
trend_plot("% removed", 'percent', 'Percentage of comments moderator removed', 'sub',
['r/Canada', 'r/CanadaPolitics'], weekly_stats, 'deep', 0.2, 0)
```

Figure 3: changes in the average net upvote score

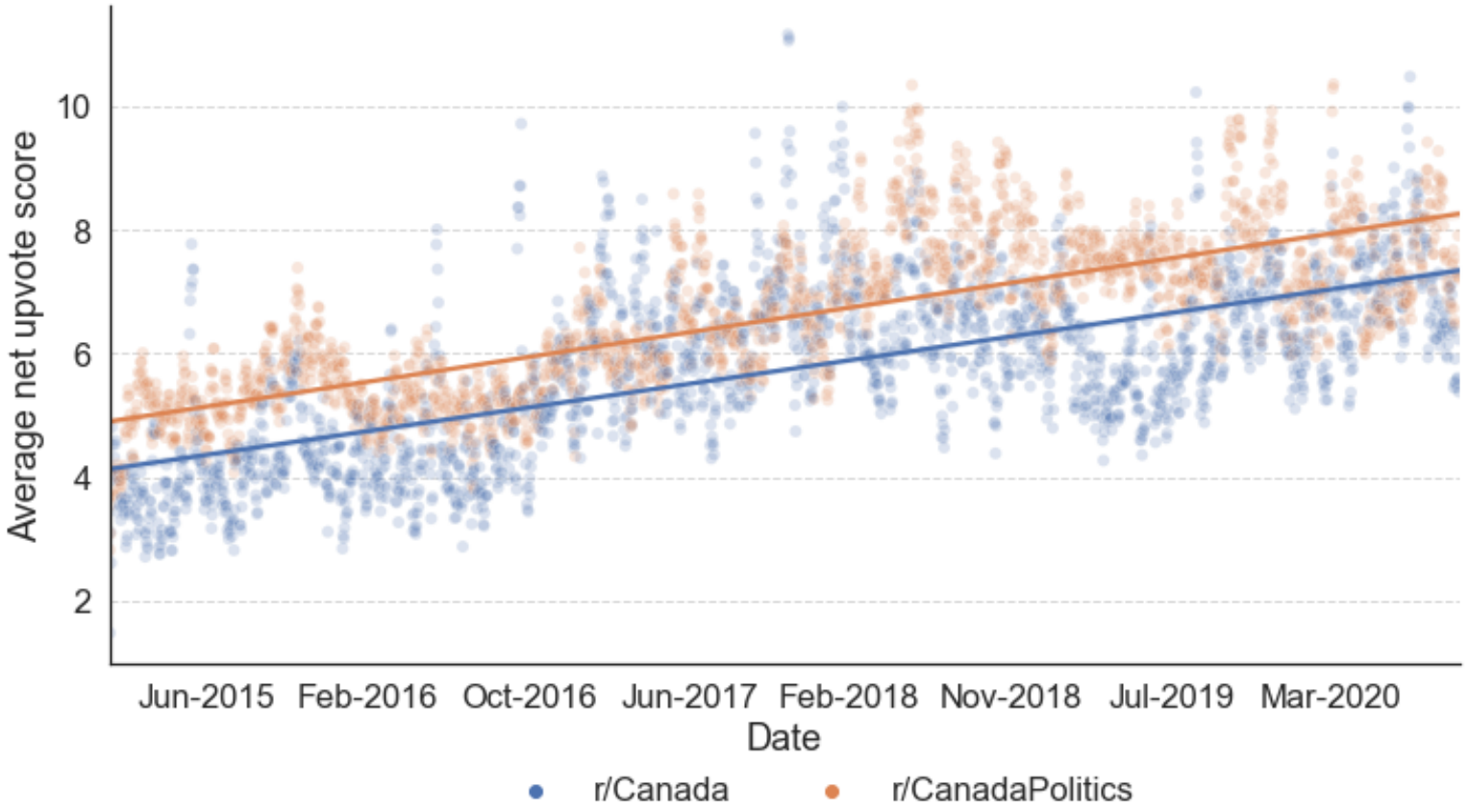
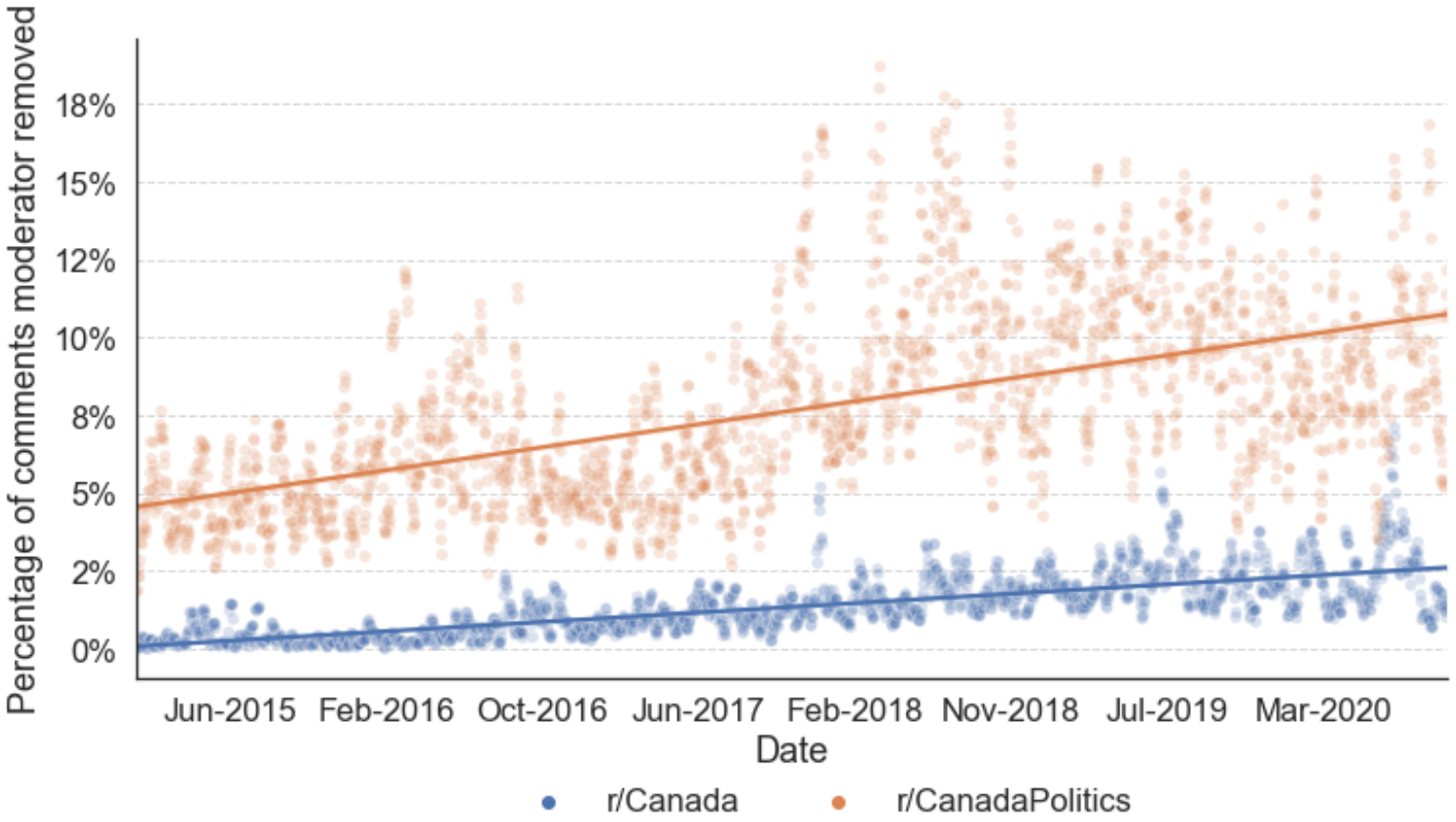


Figure 4: Changes in the rate of moderator removal



Subsequently, I created some venn diagrams to better understand the crossover between subreddits.

First, I generated a dataframe which grouped comments by the date they were made, the author who made them, and the subreddit they were posted to. Using this information, I generated a column for each subreddit, which took the value of 1 if the user had posted to that subreddit on that day, and zero otherwise. After this, I removed authors who have susequently deleted their accounts (and thus all appear as a 'null' author). Including this would show inaccurate crossover between subreddits, as two different deleted account will both appear as 'null', and thus look like a match.

```
vdata = df.groupby(['date','author','sub']).count()[['comment']].reset_index()
[['date','author','sub']]

vdata = pd.concat([vdata, pd.get_dummies(vdata['sub'])], axis = 1).drop(columns =
['sub'])

vdata = vdata[vdata['author'] != 'nan']
```

I was then left with a dataframe where each line consisted of a day, an author, whether or not they had commented on r/Canada that day, and whether or not they had commented on r/CanadaPolitics that day.

date	author	r/Canada	r/CanadaPolitics
2015-01-01 00:00:00	1403205418	1	0
2015-01-01 00:00:00	2IRRC	1	0
2015-01-01 00:00:00	AggregateTurtle	1	0
2015-01-01 00:00:00	AlfredTheGreatest	0	1
2015-01-01 00:00:00	AngryMulcair	1	0
2015-01-01 00:00:00	ArchieMoses	1	0
2015-01-01 00:00:00	ArchieMoses	0	1

However, as is evident from user 'ArchieMoses' above, if a user had posted to both subreddits on the same day, there would be two separate entries for them on that day: one where r/Canada was equal to one and r/CanadaPolitics equal to zero, and vice versa.

As such, I then grouped this dataframe by author and date, which would collapse cases like this into a single line where both subreddits carry a value of one.

```
vdata = vdata.groupby(['date', 'author']).sum()[['r/CanadaPolitics','r/Canada']]
```

date	author	r/CanadaPolitics	r/Canada
2015-01-01 00:00:00	1403205418	0	1
2015-01-01 00:00:00	2IRRC	0	1
2015-01-01 00:00:00	AggregateTurtle	0	1
2015-01-01 00:00:00	AlfredTheGreatest	1	0
2015-01-01 00:00:00	AngryMulcair	0	1
2015-01-01 00:00:00	ArchieMoses	1	1
2015-01-01 00:00:00	AutisticGoyim	0	1

Above we see user ArchieMoses' two lines have now become one.

Next I created a column called 'cp_ca' which was defined as the corresponding rows of the r/Canada column multiplied by the corresponding rows of the r/CanadaPolitics column. Because these columns take a value of either zero or one, this allowed for the identification of authors who posted in both subreddits on a given day.

```
vdata['cp_ca'] = vdata['r/CanadaPolitics'] * vdata['r/Canada']
```

date	author	r/CanadaPolitics	r/Canada	cp_ca
2015-01-01 00:00:00	1403205418	0	1	0
2015-01-01 00:00:00	2IRRC	0	1	0
2015-01-01 00:00:00	AggregateTurtle	0	1	0
2015-01-01 00:00:00	AlfredTheGreatest	1	0	0
2015-01-01 00:00:00	AngryMulcair	0	1	0
2015-01-01 00:00:00	ArchieMoses	1	1	1
2015-01-01 00:00:00	AutisticGoyim	0	1	0

Take, for example, 'ArchieMoses' again. He posted in both subreddits, and so has values of one in both the r/Canada column and the r/CanadaPolitics column. Thus, the multiplication that created the 'cp_ca' column would be 1 x 1 = 1. The value of one identifies 'ArchieMoses' as a commenter in both subreddits on that day.

If instead I looked at user 'AngryMulcair', the equation would be 1 x 0 = 0. The **lack** of a one in the 'cp_ca' column identified him as a user who did **not** post in both subs.

Next I grouped the dataframe only by date, summing up the ones and zeros. This had the effect of counting the **number** of users which posted to each subreddit, and both subreddits on any given day.

```
vdata = vdata.groupby(['date']).sum()
```

date	r/CanadaPolitics	r/Canada	cp_ca
2015-01-01 00:00:00	10	21	0
2015-01-02 00:00:00	6	64	0
2015-01-03 00:00:00	31	35	0
2015-01-04 00:00:00	52	45	3
2015-01-05 00:00:00	60	308	5
2015-01-06 00:00:00	48	106	5

For example, I saw that on January 4th 2015, 52 users posted on r/CanadaPolitics, 45 users posted on r/Canada, and 3 of those 45 were also part of the 52 who posted on r/CanadaPolitics.

I then generated rolling seven day averages to smooth out noise again, added a few time identifier variables to build specific venn diagrams, and was ready to to make some charts!

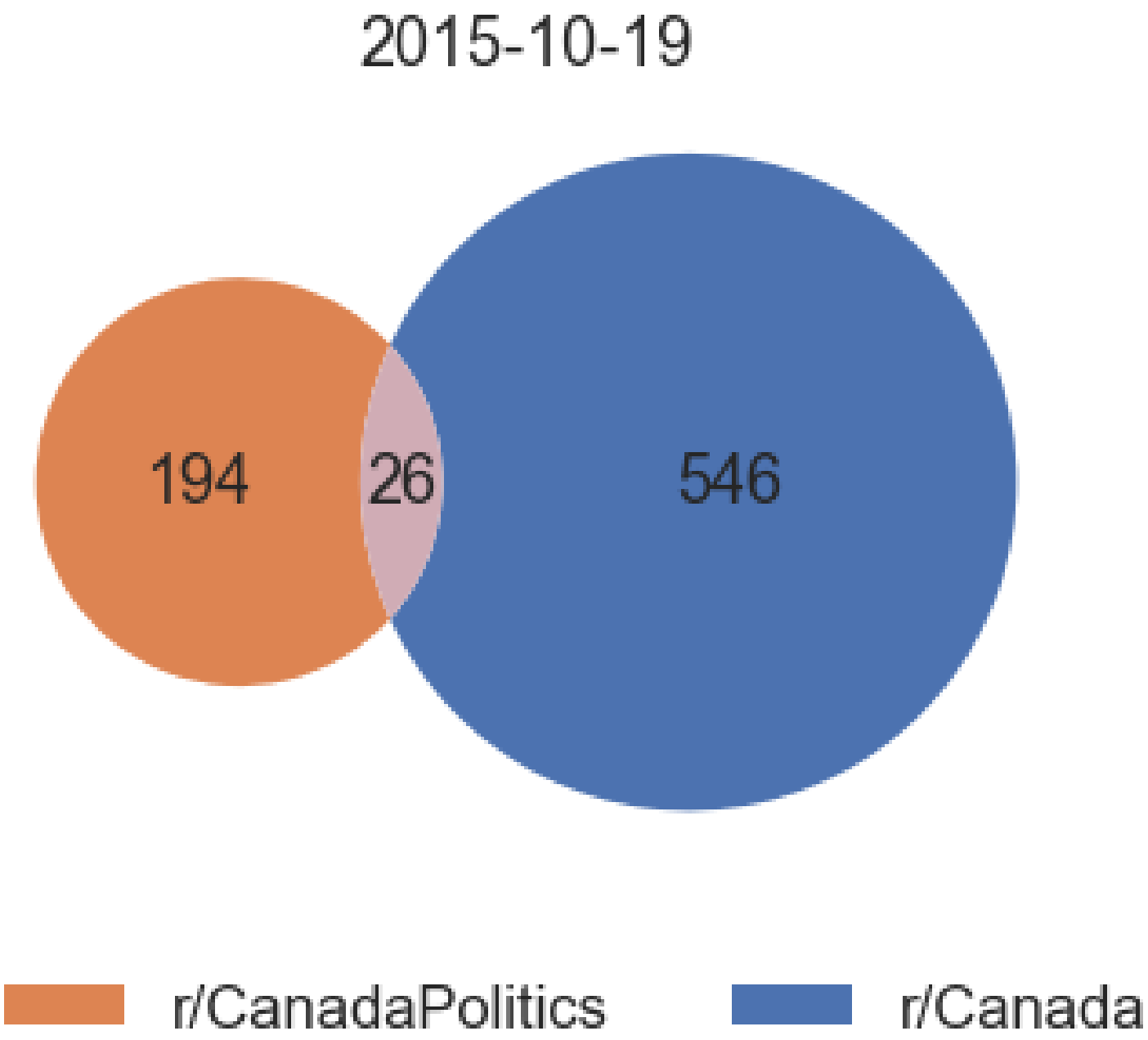
```
for i in vdata.columns[1:]:
    vdata[i] = vdata[i].transform(lambda row: round(row.rolling('7d', min_periods =
1).mean()))

year = pd.DatetimeIndex(vdata.index).year.tolist()
```

```
month = pd.DatetimeIndex(vdata.index).month.tolist()
day = pd.DatetimeIndex(vdata.index).day.tolist()
```

First, I defined the function that would create the venn diagrams and save them. Then, I constructed a loop that calls that function for each day in the dataframe, generating a venn diagram for each day. An example of one of those diagrams is included below.

```
def venn(year: int, month: int, day: int, data: object, filenumber: float):
    """
    Creates a venn diagram of users between subreddits.
    Subsequently saves that venn diagram as a file, for later use in producing an
animation.

    Args:
        year: The year to filter the input data on.
        month: The month to filter the input data on.
        day: The day to filter the input data on.
        data: The input data.
        filenumber: The number to name the file as when saving.
    """
    v1 = data[(pd.DatetimeIndex(vdata.index).year == year) &
(pd.DatetimeIndex(vdata.index).month == month) & (pd.DatetimeIndex(vdata.index).day ==
day)]
    values = tuple(v1[['r/CanadaPolitics','r/Canada','cp_ca']].astype(int).values[0])
    labels = ['r/CanadaPolitics', 'r/Canada']
    plt.figure()
    ax = plt.gca()
    v = venn2(subsets = values, set_labels = None, alpha = 1, ax = ax, set_colors =
(sns.color_palette('deep')[1], sns.color_palette('deep')[0]))
    h, l = [],[]
    for i, j in zip(['10', '01'], labels):
        h.append(v.get_patch_by_id(i))
        l.append(j)
    ax.legend(handles = h, labels = l ,bbox_to_anchor=(0.01, 0.05), loc=2,
borderaxespad=0., ncol = 3, frameon = False)
    plt.title(str(year) + '-' + str(month) + '-' + str(day))
    plt.savefig('Venn Diagrams\\' + str(filenumber) + '.png')
    plt.close()

for i, j, w, z in zip(year, month, day, range(len(year))):
    venn(i, j, w, vdata, z)
```

Figure 5 – Example venn diagram



To get a view for trends in user evolution and crossover across time, I also generated a pair of line charts using the following code and predefined plotting functions from earlier:

```
vdata['c_perc'] = (vdata['cp_ca']/(vdata['r/Canada']+vdata['cp_ca']))* 100
vdata['cp_perc'] = (vdata['cp_ca']/(vdata['r/CanadaPolitics']+vdata['cp_ca'])) * 100
vdata[['c_perc', 'cp_perc']] = vdata[['c_perc', 'cp_perc']].rolling(7).mean()
vdata['chart_dates'] = [mdates.date2num(i) for i in vdata.index]

melt = pd.melt(vdata, id_vars = ['chart_dates'], value_vars = ['r/Canada',
'r/CanadaPolitics'])
```

```
line_plot('value', 'Userbase', 'variable', melt, 'deep', 1)

melt = pd.melt(vdata, id_vars = ['chart_dates'], value_vars = ['c_perc', 'cp_perc'])
melt = melt.dropna()

trend_plot("value", 'percent', 'Percentage crossover', 'variable', ['c_perc', 'cp_perc'],
melt, 'deep', 0.2, 1, ['r/Canada', 'r/CanadaPolitics'])
```

Figure 6: Evolution of subreddit active userbases

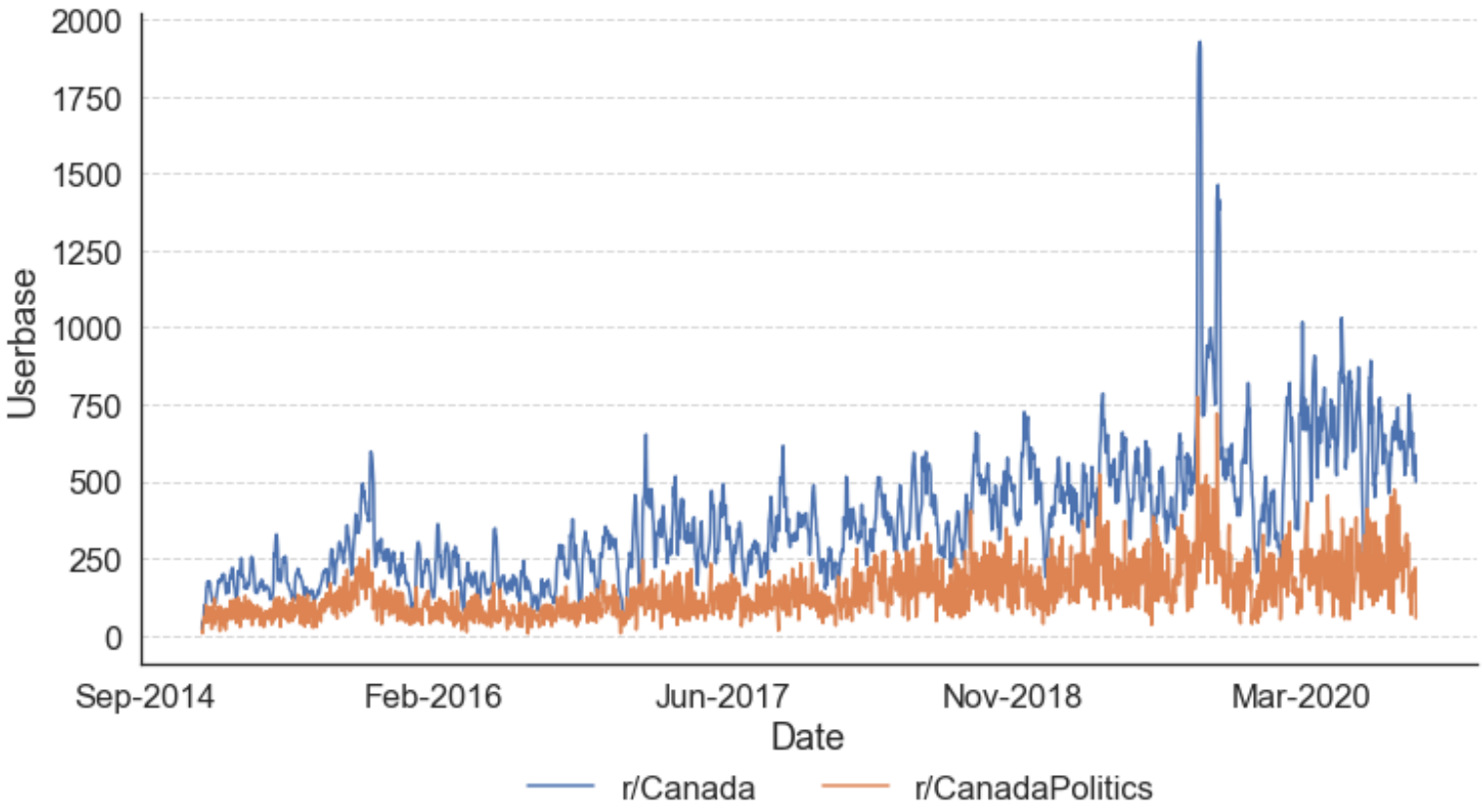
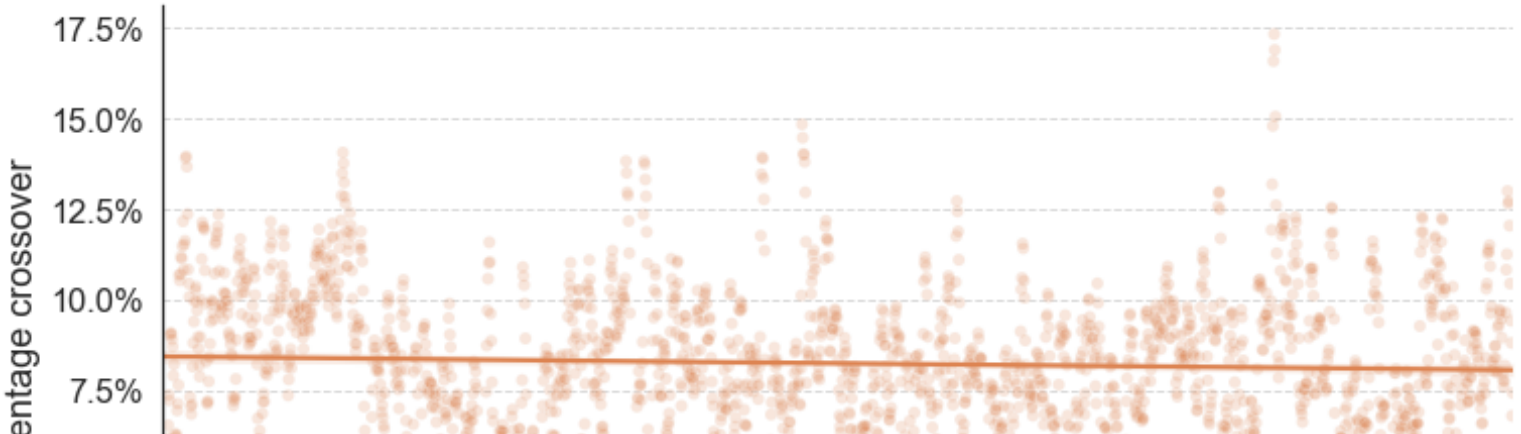
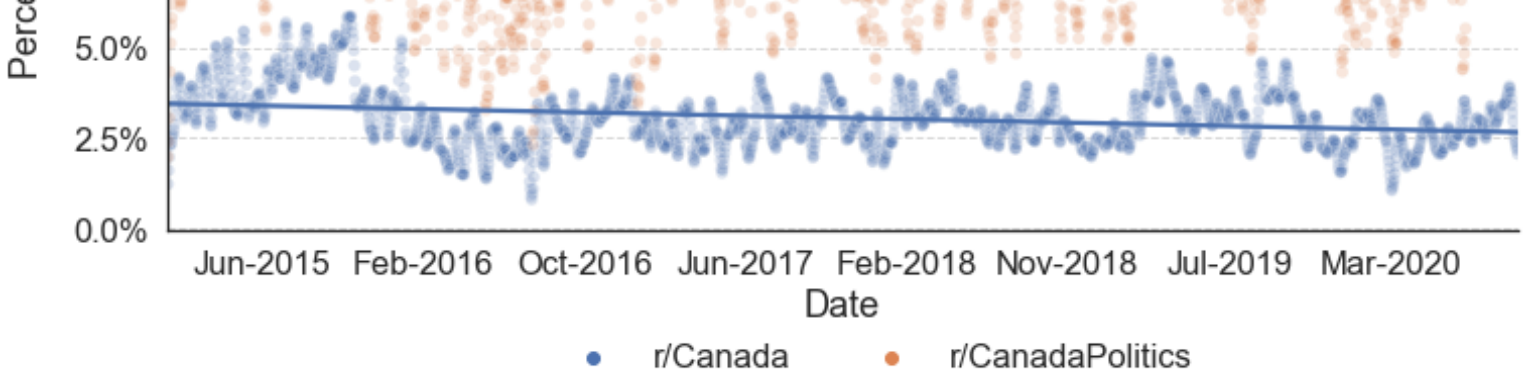


Figure 7: Evolution of user crossover





Lastly, I converted all the saved venn diagrams into an animation, to further help get a view for trends over time. The necessary code is below.

```
files = [f for f in listdir('Venn Diagrams\\')]

files = sorted([int(f.replace('.png','')) for f in files])

files = [str(i) +'.png' for i in files]

images = []

for i in files:
    im = Image.open('Venn Diagrams\\' + i)
    images.append(im)

images[0].save('GIFS\\venns.gif',
               save_all=True, append_images=images[1:], optimize=False, duration=100,
loop=1)


melt = pd.melt(venn_cons, id_vars = ['chart_dates'], value_vars = ['c_perc', 'cp_perc'])
melt = melt.dropna()

trend_plot("value", 'percent', 'Percentage crossover', 'variable', ['c_perc', 'cp_perc'],
melt, 'deep', 0.2, 1, ['r/Canada', 'r/CanadaPolitics'])
```

Word Clouds

For the next set of visualizations it became important to now clean out comments that were removed by moderators. While these were relevant for the statistics I had previously generated, such as the moderator removal rate on any given day, there is no text for these comments except '[removed]'. As such, they would dominate a word cloud.

I also needed to clean out any punctuation, digits, website links, and multi-spaces (spaces between words consisting of more than one space).

```
df = df[df['comment'] != '[removed]']

def strip(data: object, column: str) -> object:
    """
    Remove all non-letter characters, all hyperlinks, and all multispaces from a series of strings.
    Subsequently remove any comments that have become empty because of the previous operations.

    Args:
        data: a dataframe object.
        column: a column of text within the dataframe.

    Returns:
        A dataframe object with the desired column cleaned.
    """
    data[column] = data[column].str.replace('[^\w\s]','')
    data[column] = data[column].str.replace('[\d+]','')
    data[column] = data[column].str.replace("http\w+", '')
    data[column] = data[column].str.replace('\s+', ' ')
    data = data[(data[column] != ' ') & (data['comment'] != '')]
    return data

df = strip(df, 'comment')
```

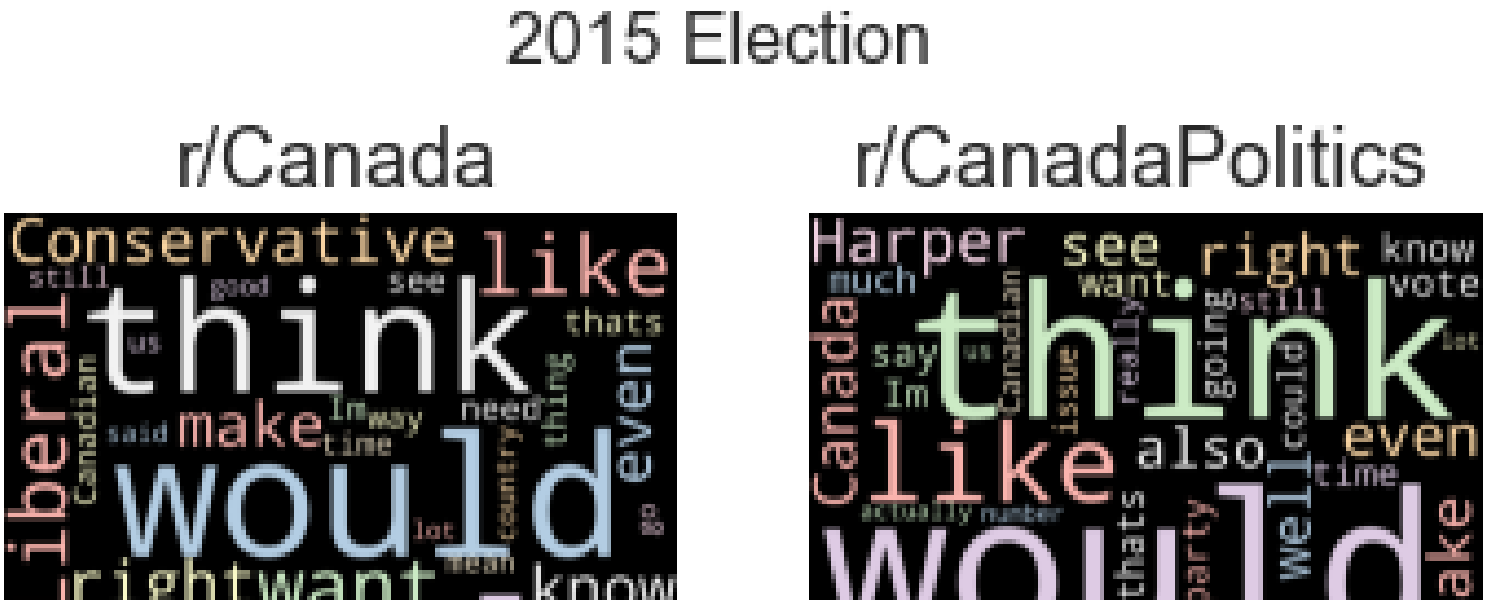
Next, I needed to remove 'stop words' – commonly used English words that carry no important meaning for the overall message. One example is 'myself'. I made use of the Natural Language Toolkit (NLTK) library for this task. However, I removed any punctuation from NLTK's stop words – as I needed to match against the text that I had just removed all punctuation from within my own dataset. Lastly, NLTK's stop words are all lowercase. I had thus far maintained the capitalization of my dataset, as capitalization of named entities is more visually enticing in word clouds. As such, I added to the stop words list, capitalized versions of all stop words, so that they were caught and removed as well.

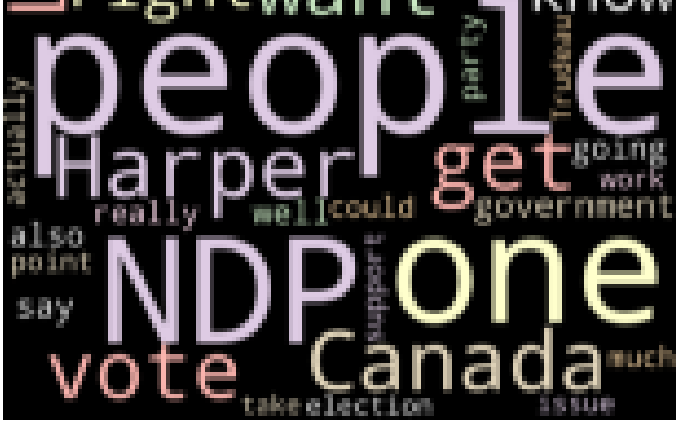
```
#Word clouds
stop_words = list(stopwords.words('english'))

stop_words = [i.replace("'", "") for i in stop_words]

l = len(stop_words)
for i in range(l):
    stop_words.append(stop_words[i].capitalize())
```

In the summary article, I generated word clouds for each of the two election periods held during my data collection range, one of which is displayed below. The code includes a random state set at 9330, for reproducibility.

```
election_2015 = df[(df['date'] > '2015-08-03') & (df['date'] < '2015-10-21')]
election_2019 = df[(df['date'] > '2019-09-10') & (df['date'] < '2019-10-22')]

for j, z in zip([election_2015, election_2019], ['2015 Election', '2019 Election']):
    fig = plt.figure()
    gs = GridSpec(nrows=1, ncols=2)
    for i, w in zip(['r/Canada','r/CanadaPolitics'], [0,1]):
        test = j[j['sub'] == str(i)]
        text = " ".join(comment for comment in test.comment)
        wordcloud = WordCloud(stopwords=stop_words,
                              width = 2500,
                              height = 3000,
                              background_color="black",
                              random_state = 9330,
                              collocations = True,
                              colormap = 'Pastel1',
                              max_words = 50).generate(text)
        axw = fig.add_subplot(gs[w])
        axw.imshow(wordcloud, interpolation='bilinear')
        plt.axis("off")
        plt.imshow(wordcloud, interpolation='bilinear')
        axw.set_title(str(i))
    fig.suptitle(z, fontsize=16)
```

Figure 8: Example word cloud





Classification

After completing the previous work, I was almost ready to move on to classification. A little bit more data manipulation was necessary to prepare the dataset for this process. First, I needed to turn all words into lowercase, and then tokenize the comments. Tokenization is the process of turning a string into a list of elements, be that words, sentences, or otherwise. In this case, I used word tokenization, so 'this is an example sentence' becomes [this, is, an, example, sentence].

```
#convert comments to lowercase
def lower(input_string: str) -> str:
    """ Convert a string to all lowercase. """
    return input_string.lower()

df['comment'] = df['comment'].apply(lower)

#tokenize comments
def word_split(input_string: str) -> list:
    """ Tokenize a string. """
    return word_tokenize(input_string, language = 'english')

df['comment'] = df['comment'].apply(word_split)
```

Subsequently, I removed all stop words from the data set (previously I had just suppressed them in the word clouds). If any comments had consisted **entirely** of stop words, they would then be empty comments. As such, I added additional code to remove them.

```
def remove_stops(list_of_strings: list) -> list:
    """ Remove all stopwords from a list of strings. """
    return [i for i in list_of_strings if i not in stop_words]

df['comment'] = df['comment'].apply(remove_stops)
```

```
#remove null comments
df = df[df['comment'].apply(len) > 0]
```

Lastly, to try and get the number of features for my model down to a more manageable level, I reduced the word tokens to their stems. For instance, 'fisher', 'fishes', and 'fishing', would all be changed to 'fish'. For this I used the PorterStemmer function from the NLTK library.

```
def stem(list_of_strings: list) -> list:
    """ Stem words in a list of strings. """
    return [stemmer.stem(i) for i in list_of_strings]

df['comment'] = df['comment'].apply(stem)
```

I was left with a dataset of approximately 2.85 million comments, which became very computationally expensive when running the classification algorithm, particularly when I got to k-fold cross validation. To deal with this issue, I took a sample of 1.5 million comments from the overall dataset.

```
sample = df[['comment','sub', 'date', 'year']].sample(1500000, random_state = 9330)
```

To sense check the sample, I compared the distribution of comments in the original and sample dataframes by subreddit, and by year.

Figure 9: Distributions in the full dataset

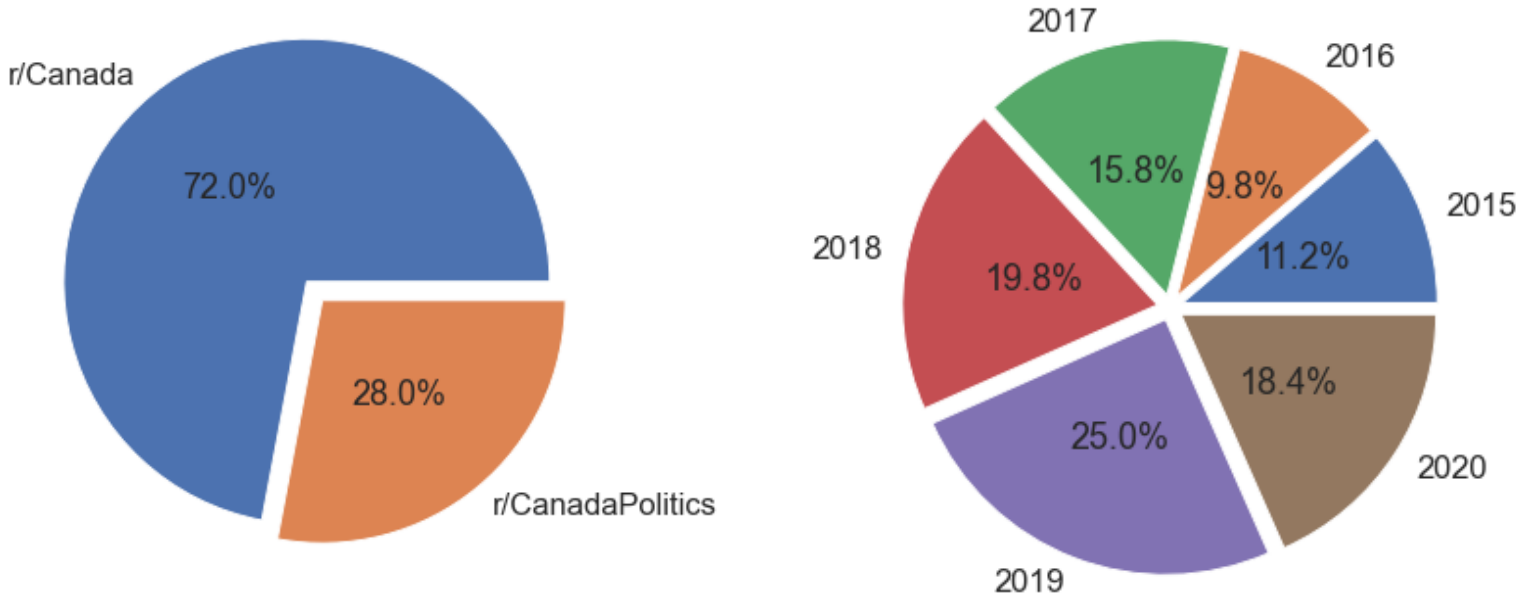
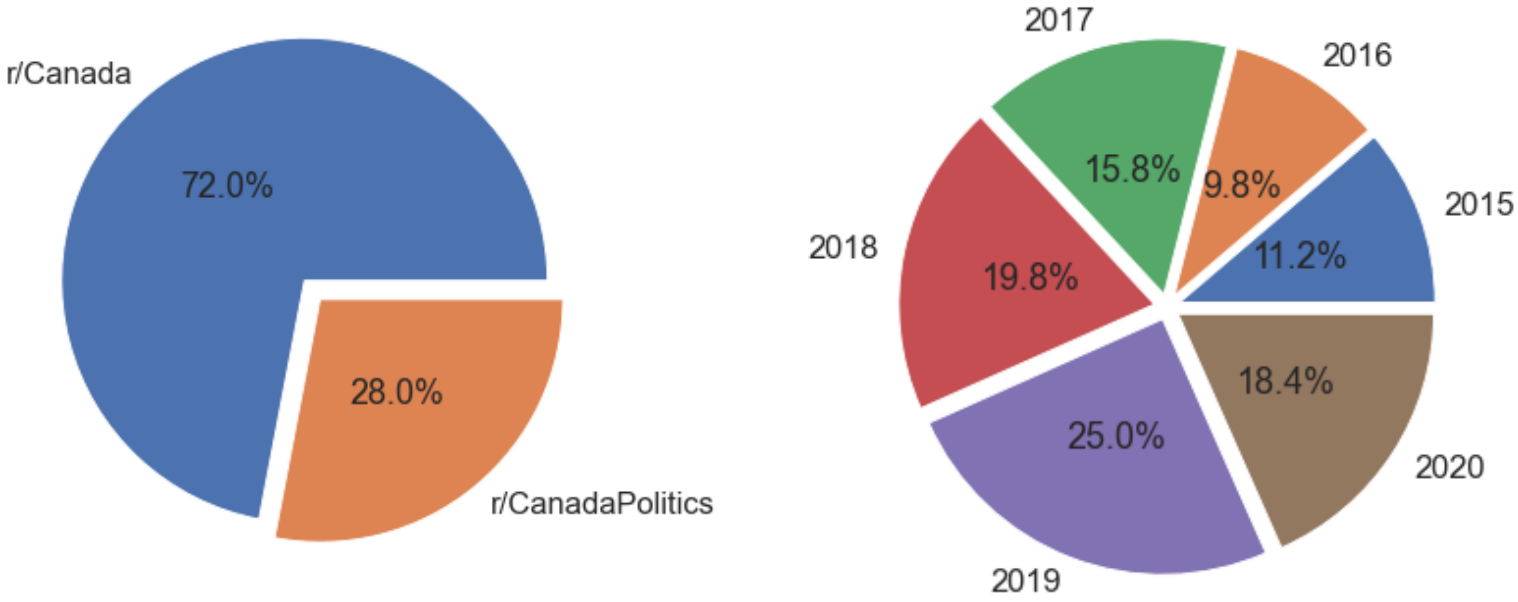


Figure 10: Distributions in the sample dataset



As these distributions matched, I was comfortable moving on to the next stage, which was conducting the classification.

First, I prepared the data for modeling by running the word tokens through Sci-Kit Learn's TfidfVectorizer. A TfidfVectorizer is distinct from a count vectorizer as it attempts to account for the importance of the frequency of words in the entire corpus. By contrast, a count vectorizer simply counts occurrences of a word. As was evident in the word clouds, the word 'people' appears often – but this may not be terribly useful in determining meaning. A TfidfVectorizer helps correct for this.

```
vectorizer = TfidfVectorizer(ngram_range = (1, 1), max_features = 1500)
```

Above, I instantiated the vectorizer, instructing it to seek only unigrams, and to limit the number of features to 1500. I experimented with bigrams and trigrams for this model, as well as larger feature numbers. In general, this tended to vastly increase computation time for no material improvement to the model's performance.

Next, I assigned the target-variable – or the thing I want the model to predict – to be the subreddit a comment was made on, vectorize the comments, and split the data into a training and test set.

```
X_train, X_test, y_train, y_test =
train_test_split(vectorizer.fit_transform(sample['comment'].values.astype(str)).toarray()
, sample['sub'], test_size = 0.25, random_state = 9330)
```

After this, I established which models I would be testing, as follows:

1. **Linear Support Vector Machine**, with class_weight set to balanced. This will penalize mispredictions on the minority class, preventing the model from simply taking the fact that the sample is skewed towards r/Canada, and just predicting almost everything as r/Canada as a result.

2. **Multinomial Naïve Bayes**, with fit_prior set to false. This prevents the model from learning the 70/30 distribution of comment data, and factoring this into its prediction. We want the model to predict based on the comment text alone.

3. **Logistic Regression**, again with class_weight set to balanced, and with solver set to 'saga', as this is a computationally efficient solver for large datasets such as ours. Random state is set to 9330 here for reproducibility.

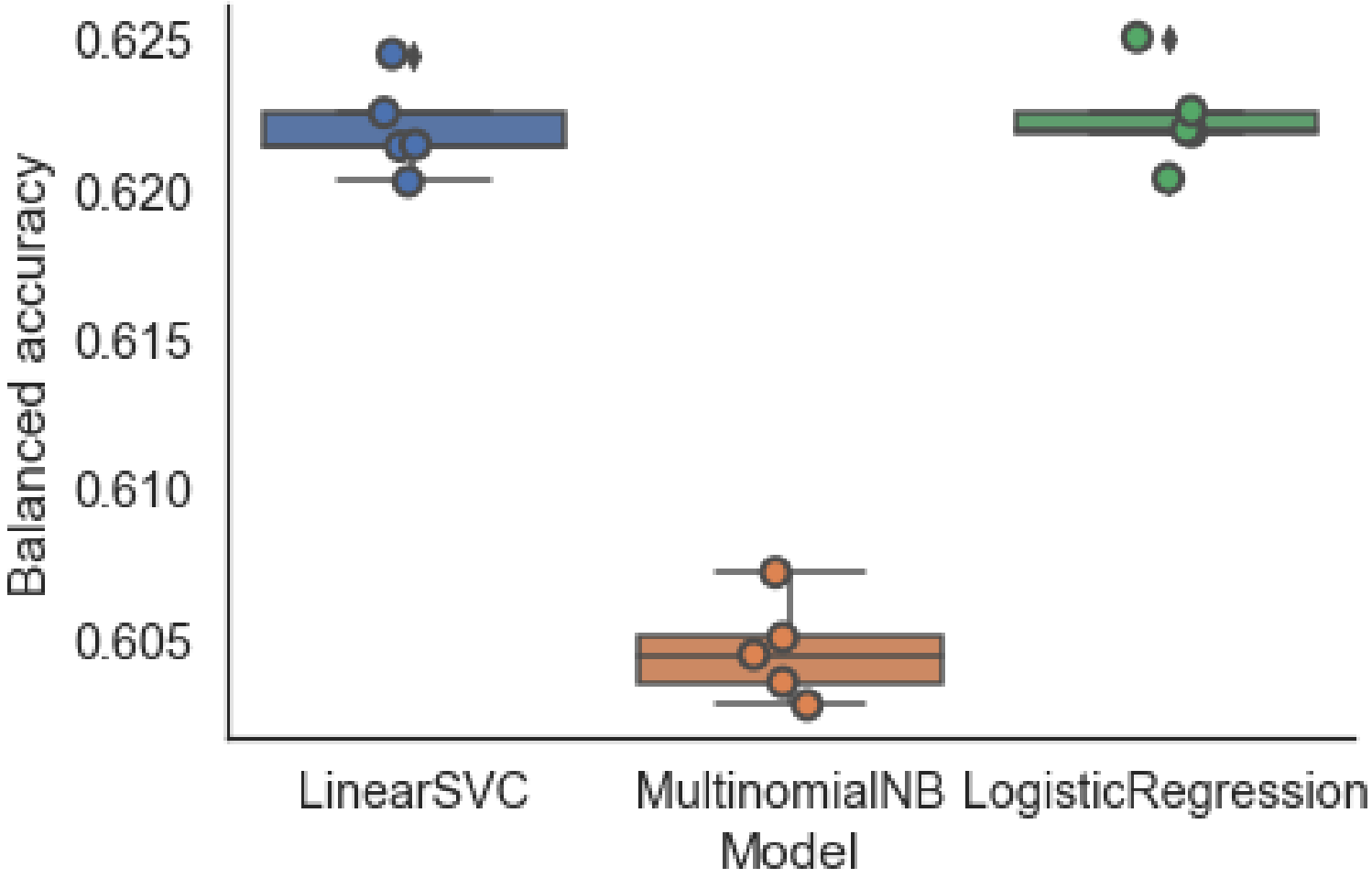
```
models = [
    LinearSVC(class_weight='balanced'),
    MultinomialNB(fit_prior=False),
    LogisticRegression(class_weight='balanced', random_state = 9330, solver = 'saga')
    ]
```

Before choosing which model to use, it is worth considering which measure of accuracy one might select to evaluate performance in this context. Given that the sample is imbalanced, a simple measure like accuracy would be deceiving, as a classifier which simply assigned everything to the majority class would be fairly accurate, but not very useful.

To address this, I elected to use 'balanced accuracy', which is defined as the **average** of the recall score between both classes. Recall measures the model's ability to find all positive instances of a class. By taking the average, this metric penalizes models which underperform on either class.

Thus, the selection was determined by which model performed best in terms of balanced accuracy across five-fold cross-validation. In practice the results – in terms of the best fitting model – tended to be consistent across many different model evaluation metrics. The code and output for this selection process is included below.

```
k = 5
cv_df = pd.DataFrame(index=range(k * len(models)))
entries = []
for model in models:
  model_name = model.__class__.__name__
  accuracies = cross_val_score(model, X_train, y_train, scoring='balanced_accuracy',
cv=k)
  for fold_idx, accuracy in enumerate(accuracies):
    entries.append((model_name, fold_idx, accuracy))
cv_df = pd.DataFrame(entries, columns=['Model', 'fold_idx', 'Balanced accuracy'])
sns.boxplot(x='Model', y='Balanced accuracy', data=cv_df)
sns.stripplot(x='Model', y='Balanced accuracy', data=cv_df,
              size=8, jitter=True, edgecolor="gray", linewidth=2, palette = 'deep')
sns.despine()
plt.show()
```



As is evident above, the Logistic Regression and Linear Support Vector Machine both strongly outperform the Multinomial Naïve Bayes. Additionally, the Logistic Regression has a slightly higher average and tighter interquartile range, so I select the Logistic Regression as my classifier.

Next, I conducted some hyper parameter tuning using grid search and find that the optimal value for C is 0.1.

```
model = LogisticRegression(class_weight = 'balanced', random_state = 9330, solver =
'saga')

param_grid = {'C': [0.01,0.1,1,10]}

grid = GridSearchCV(model, param_grid,refit=True, scoring = 'balanced_accuracy')
grid.fit(X_train,y_train)
print(grid.best_estimator_)
```

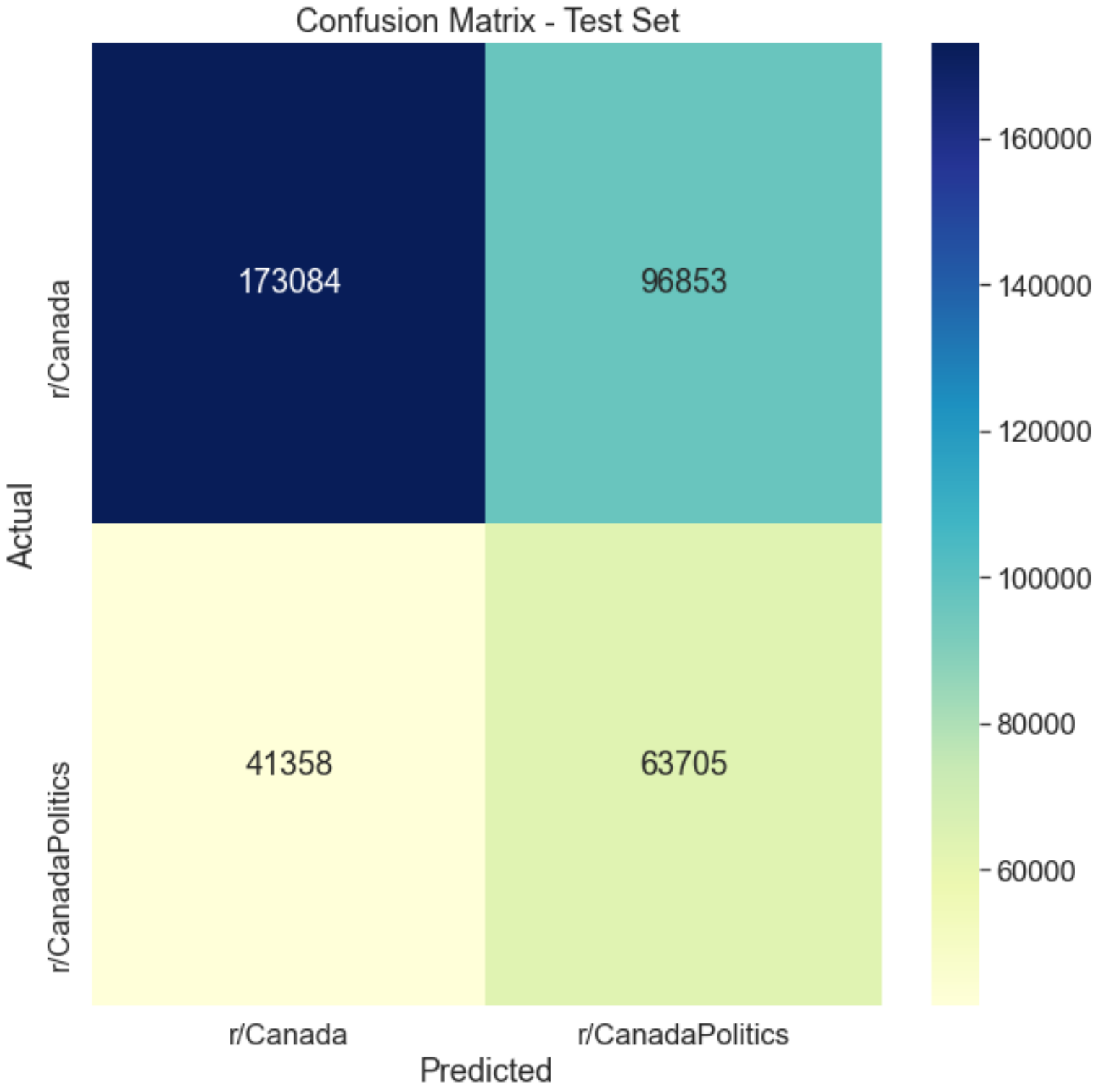
Using the tuned C value, I am able to fit and test the model, and then define a function that scores the training and test sets. This allowed me to check for overfitting in the model.

```
model = LogisticRegression(C=0.1, random_state = 9330, solver = 'saga',
class_weight='balanced')
```

```
model.fit(X_train, y_train)

def results_score(X: object, y: object, model:object) -> float:
    """ Calculates the balanced accuracy score for a series of predictions versus true
values

    Args:
        X: The input data that informs the prediction.
        y: The true values to compare predictions against.
        model: the fitted classifier

    Returns:
        The balanced accuracy score
    """
    y_pred = model.predict(X)
    return [balanced_accuracy_score(y, y_pred)]

print(results_score(X_train, y_train, model))
print(results_score(X_test, y_test, model))
```

The training set achieves a balanced accuracy score of 62.49%, while the testing set achieves 62.37%. The closeness in these values helps reassure me that the model is not overfit. I also generated a confusion matrix, which showed that the model was accurately predicting over half of each class, which aligns with the balanced accuracy scores.

```
def conf_m(X: object, y: object, title: str):
    """ Plots a confusion matrix for a series of predictions versus true values

    Args:
        X: The input data that informs the prediction.
        y: The true values to compare predictions against.
        title: the title to place on the confusion matrix
    """
    y_pred = model.predict(X)
    conf_mat = confusion_matrix(y, y_pred)
    fig, ax = plt.subplots(figsize=(10,10))
    sns.heatmap(conf_mat, annot=True, fmt='d',
                xticklabels=['r/Canada', 'r/CanadaPolitics'], yticklabels=['r/Canada',
'r/CanadaPolitics'], cmap = 'YlGnBu')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title(title)
    plt.show()

conf_m(X_test, y_test, "Confusion Matrix - Test Set")
```



As the key metric for this model is the evolution of balanced accuracy **over time**, I next needed to rearrange the model outputs into a form that allows for that temporal comparison.

```
y_test = y_test.to_frame()
y_test['predictions_test'] = model.predict(X_test)
y_test['category_test'] = "Test"

y_train = y_train.to_frame()
y_train['predictions_train'] = model.predict(X_train)
```

```
y_train['category_train'] = "Train"

results = sample.merge(y_test[['predictions_test', 'category_test']], how = 'left',
left_index = True, right_index = True)
results = results.merge(y_train[['predictions_train', 'category_train']], how = 'left',
left_index = True, right_index = True)

results['predictions'] = results['predictions_test'].fillna(results['predictions_train'])
results['category'] = results['category_test'].fillna(results['category_train'])

results = results[['sub','predictions', 'category','date']]

results['date'] = pd.to_datetime(results['date'])
```

The above code creates a dataframe of the following form:

sub	predictions	category	date
r/Canada	r/CanadaPolitics	Train	2017-02-13
r/CanadaPolitics	r/CanadaPolitics	Train	2020-05-14
r/Canada	r/CanadaPolitics	Test	2018-06-13
r/Canada	r/Canada	Train	2020-06-09
r/Canada	r/CanadaPolitics	Train	2018-10-30
r/Canada	r/CanadaPolitics	Train	2020-05-12
r/Canada	r/CanadaPolitics	Train	2019-06-04
r/Canada	r/CanadaPolitics	Test	2015-09-29
r/Canada	r/Canada	Test	2019-04-23
r/CanadaPolitics	r/CanadaPolitics	Test	2019-02-14

In the above dataframe, each line contains:

- the true class of a given comment;
- a comment's predicted class;
- whether the comment was part of the training or test dataset; and
- the date on which the comment was made.

This allowed me to calculate a recall metric for specific time periods. In the following code, I did just that on a monthly basis, by comparing the true class and the predicted class. This converted the dataframe into a form where the final column allowed me to track balanced accuracy on a monthly basis.

```
def recall_processing(subs: list, data: object) -> object:
    """
    Creates helper columns and then uses those columns to calculate a balanced accuracy
score for each time period in the dataframe.
    Additionally converts the time period in the dataframe into months.

    Args:
        subs: The different subreddits in the dataframe
        data: The input dataframe.

    Returns:
        A modified dataframe with a balanced accuracy score for each time period.
    """
    for i in subs:
        data[i + '_tp'] = np.where((data['sub'] == i) & (data['predictions'] == i), 1, 0)
        data[i + '_fp'] = np.where((data['sub'] != i) & (data['predictions'] == i), 1, 0)
        data[i + '_tn'] = np.where((data['sub'] != i) & (data['predictions'] != i), 1, 0)
        data[i + '_fn'] = np.where((data['sub'] == i) & (data['predictions'] != i), 1, 0)
    data['date'] = data['date'].dt.to_period('M').apply(lambda r: r.start_time)
    data = data.groupby(['date', 'category']).sum()[data.columns[4:]]
    for j in subs:
        data[j + '_recall'] = data[j +'_tp'] / (data[j +'_tp'] + data[j +'_fn'])
    data['balanced_accuracy'] = (data['r/Canada_recall'] +
data['r/CanadaPolitics_recall']) / 2
    return data

results = recall_processing(['r/Canada', 'r/CanadaPolitics'], results)

results.reset_index(inplace = True)
```

```
results['balanced_accuracy'] = results.groupby('category')
['balanced_accuracy'].transform(lambda row: row.rolling(6, min_periods = 1, center =
True).mean())
```

date	category	r/Canada_recall	r/CanadaPolitics_recall	balanced_accuracy
2015-01-01 00:00:00	Test	0.603908	0.663564	0.633736
2015-01-01 00:00:00	Train	0.590702	0.677778	0.63424
2015-02-01 00:00:00	Test	0.603082	0.651194	0.627138
2015-02-01 00:00:00	Train	0.586564	0.675096	0.63083
2015-03-01 00:00:00	Test	0.589805	0.646308	0.618056
2015-03-01 00:00:00	Train	0.597968	0.658179	0.628073
2015-04-01 00:00:00	Test	0.539813	0.668552	0.604182
2015-04-01 00:00:00	Train	0.543708	0.68175	0.612729
2015-05-01 00:00:00	Test	0.579017	0.682399	0.630708
2015-05-01 00:00:00	Train	0.568438	0.686136	0.627287
2015-06-01 00:00:00	Test	0.508781	0.710357	0.609569

Note that had I grouped only by the 'train' and 'test' categories (i.e. no time differentiation), the balanced accuracy score entries would have the output of the 'results_score' function discussed previously. This confirms that the underlying scores had not been altered by these transformations.

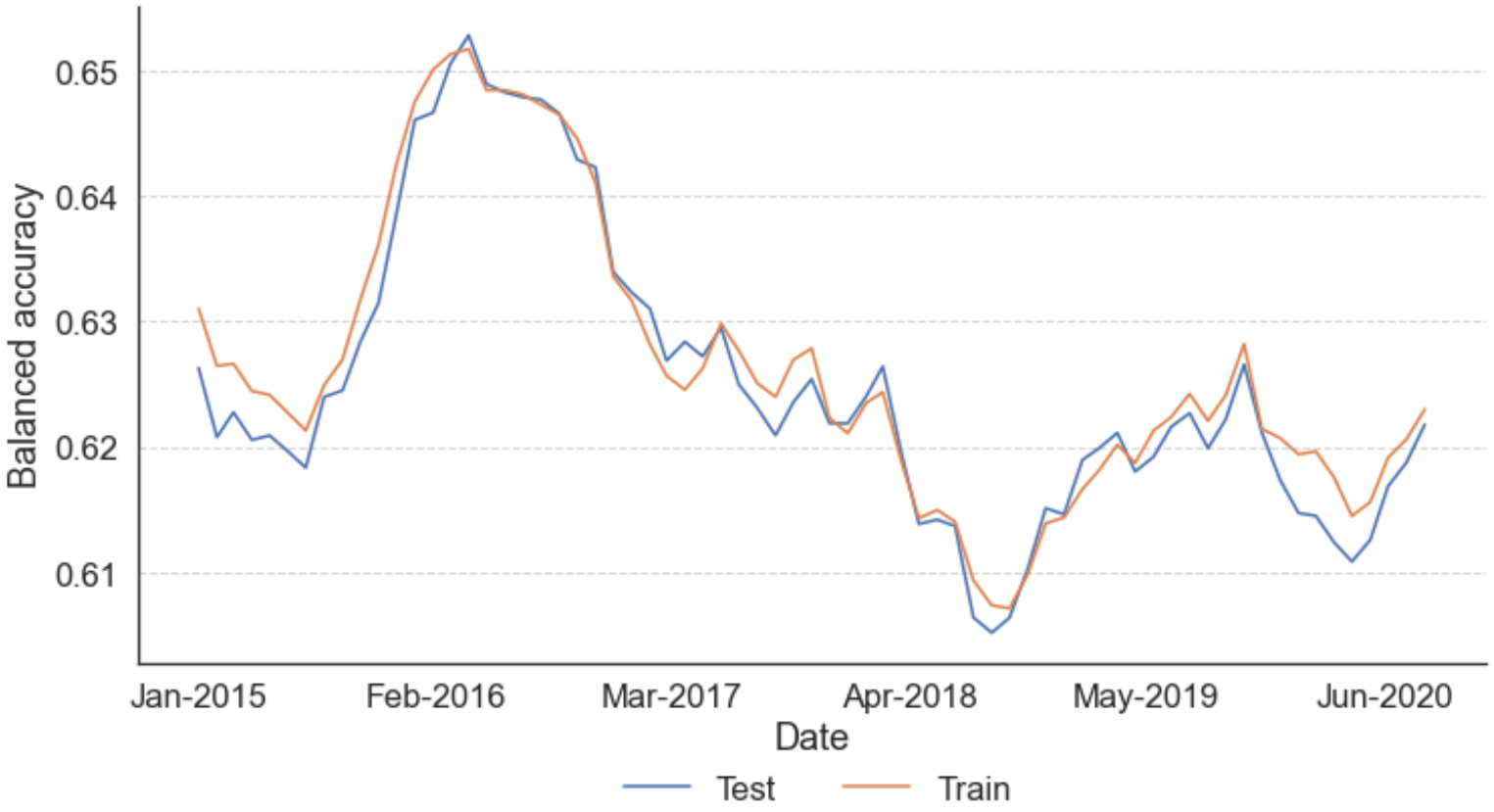
I then took a monthly rolling average on a six-month basis to help smooth out short term noise, and graphed the changes in balanced accuracy over time.

```
results['chart_dates'] = [mdates.date2num(i) for i in results['date']]

def line_plot(y_variable: str, y_title: str, categories: str, data: object, palette: str,
alpha: float):
    """
    Creates a line chart of the chosen variables over time

    Args:
        y_variable: The variable to plot on the vertical axis.
        y_title: The vertical axis title.
        categories: The variable to along which to split into different colour coded
```

```
lines.
        data: The input dataframe.
        palette: The Seaborn palette to use.
        alpha: Transparency of the lines.
    """
    fig, ax = plt.subplots(figsize = (12,6))
    sns.lineplot(x='chart_dates', y=y_variable, data = data, hue = categories, palette =
palette, alpha = alpha, ax = ax)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b-%Y'))
    ax.yaxis.grid(True, which='major', alpha = 0.75, linestyle = '--')
    ax.xaxis.set_major_locator(plt.MaxNLocator(6))
    ax.set(xlabel="Date", ylabel = y_title)
    handles, labels = ax.get_legend_handles_labels()
    ax.legend(handles=handles[1:], labels=labels[1:],bbox_to_anchor=(0.35, -0.15), loc=2,
borderaxespad=0., ncol = 3, frameon = False)
    sns.despine()

line_plot("balanced_accuracy", 'Balanced accuracy', 'category',
results[results['category'] == 'Test'], 'deep', 1)
line_plot("balanced_accuracy", 'Balanced accuracy', 'category', results, 'deep', 1)
```

The above chart makes it clear that the model is most accurate in 2016, progressively less accurate leading up to late 2018, and then increasingly accurate again in 2019, though not to 2016 levels.

A comparison of the testing and training sets helped confirm that the model is not substantially overfit across any particular time periods



Discussion of these outputs and their implications is available in the summary article. All of the code underlying this project is available on my github page, and the underlying dataset here.