Imports

In [ ]:
```
# General
import re, html, string
import numpy as np
import pandas as pd

# Plotting
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import seaborn as sns

# NLP
import nltk, spacy

nltk.download("stopwords")
nltk.download("punkt")
lem = spacy.load("de_core_news_sm")  # if error: run python -m spacy download de_core_news_md in terminal


# Tools
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity

# Models
from sklearn.naive_bayes import MultinomialNB
from sklearn.neural_network import MLPClassifier
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\schef\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\schef\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

1 Introduction

In this project the main focus revolves around the predicting the party affiliation of German politicians based on Natural Language.

For this I'll gather the Twitter data of almost all possible German politicians myself, and use the party affiliation of these politicians as labels.

The data will be preprocessed and then used to train various models. These then will be evaluated, and the results discussed.

This kind of service could be employed by media, journalism, political consultancies, or market research firms, thereby placing it within the tertiary sector.

2 Data

2.1 GESIS Dataset

GESIS is a German research institute specializing in empirical social sciences. It serves as a leading institution in research infrastructure by providing datasets and archives to other researchers. One of its two headquaters is located in Mannheim, within the university campus. They have provided us with the following dataset:

The German Federal Election 2021 Twitter Dataset includes 1,556 Twitter accounts belonging to candidates from the seven parties represented in the 19th session (2017-2021) of the Bundestag.

In [ ]:
```
politicians = pd.read_csv(
    filepath_or_buffer = "./data/twitter_handles/ZA7721_v2-0-0.csv",
    encoding = "Windows-1252",
    sep = ";",
    dtype = {
        "study": "object",
        "version": "object",
        "doi": "object",
        "field_start": "object",
        "field_end": "object",
        "id": "Int64",
        "lastname": "object",
        "firstname": "object",
        "gender": "object",
        "state": "object",
        "party": "object",
        "district_name": "object",
        "district_number": "Int64",
        "incumbent": "boolean",
        "list_place": "Int64",
        "isListed": "boolean",
        "isDC": "boolean",
        "screen_name1": "object",
        "screen_name2": "object",
        "user_id1": "Int64",
        "user_id2": "Int64",
        "change_v2": "boolean"
    }
)
```

Below is the variable description of the dataset:

Variable	Description
id	internal candidate id
lastname	Last name
firstname	First name
gender	Gender
state	German federal state in which the candidate is running
party	Party
district_name	Official name of the constituency if the candidate is running for a direct mandate
district_number	Official number of the constituency if the candidate is running for a direct mandate
incumbent	Is the candidate a sitting member of the federal parliament (Bundestag)?
list_place	The candidate"s place on the party list (if s/ he is listed)
isListed	Is the candidate running on a party list?
isDC	Is the candidate a direct candidate in one of the constituencies?
screen_name1	Screen name of the first Twitter account of the candidate (if it exists)
screen_name2	Screen name of the second Twitter account of the candidate (if it exists)
user_id1	Unique Twitter user ID of the first Twitter account of the candidate (if it exists)
user_id2	Unique Twitter user ID of the second Twitter account of the candidate (if it exists)
change_v2	Variable marking candidates With changes in Twitter accounts in version 2.0

This list of politicians will be used to scrape their tweets. As their screen names may have changed by now, I will use their `user_id` 's to identify them.

2.2 Twitter Data

"If you are interested in the Twitter API, subscribe to Free, Basic, or Enterprise tier. For Academia, we are looking at new ways to continue serving this community. Stay tuned [...]" [1].

Due to the current unavailability of a free option to utilize Twitter's API, *including academic research purposes* [2, 3], I felt obliged to manually scrape the tweets myself. For this, I utilized the snscrape python module and store the scraped tweets in the `tweets.csv` file.

FYI: I've disabled the cell below using the jupyter magic: `%%script false --no-raise-error` as it took ~25h to complete.

Skip to the next cell, marked by '⏩' to continue.

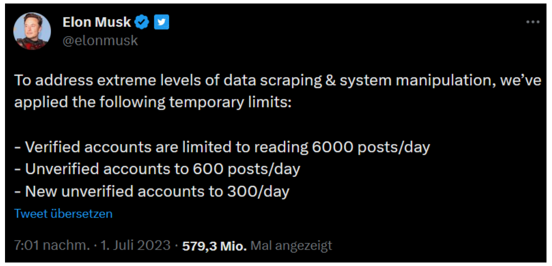
In [ ]:
```
%%script false --no-raise-error

import snscrape.modules.twitter as sntwitter

tweets_list = []
for row_index, row_data in politicians.iterrows():

    # check for second account
    accounts = []
    if pd.notna(row_data["user_id1"]):
        accounts.append(row_data["user_id1"])
    if pd.notna(row_data["user_id2"]):
        accounts.append(row_data["user_id2"])

    # start scraping tweets
    for acc in accounts:
        try:
            user = sntwitter.TwitterUserScraper(acc).entity # get username by ID
            for tweet in sntwitter.TwitterSearchScraper(f"from:{user.username}").get_items():
                tweets_list.append([row_data["id"], tweet.date, tweet.id, tweet.rawContent, tweet.user.username])

        except ScraperException as e:
            tweets_list.append([row_data["id"], pd.NA, pd.NA, f"{e}", user.username])

# export to csv
tweets_df = pd.DataFrame(tweets_list, columns=["politician_id", "date", "tweet_id", "content", "username"])
tweets_df.to_csv("./data/tweets/tweets.csv", index=False)
```

```
Couldn't find program: 'false'
```

⏩ Sadly, a mistake was made while storing the tweets to `.csv`, that resulted in failing to properly escape the `content` -column for some tweets.

2.3 Fixing Corrupt CSV File

By the time I've noticed this mistake, Twitter had already implemented counter-measures against scraping.


Elon Musk
@elonmusk

To address extreme levels of data scraping & system manipulation, we've applied the following temporary limits:

- Verified accounts are limited to reading 6000 posts/day
- Unverified accounts to 600 posts/day
- New unverified accounts to 300/day

Tweet übersetzen

7:01 nachm. · 1. Juli 2023 · 579,3 Mio. Mal angezeigt

As I did not want to

- look for an alternative way to properly scrape the tweets again
- reimplement above code to circumvent said counter-measures
- wait 25h for code completion

I've decided to fix the malformed `.csv` -file myself.

I've disabled the cell below as well. It only takes about 20 minutes to complete, but loading the resulting `.csv` -file is simply faster. Skip to the next cell, marked by '⏩', to continue.

In [ ]:
```
%%script false --no-raise-error

with open("./data/tweets/tweets_fixed.csv", "w", encoding="UTF-8") as f:
    f.write("politician_id,date,tweet_id,content,username\n")

i = 0
end_of_doc = False
with open("./data/tweets/tweets.csv", encoding="UTF-8") as f:
    content = ""
    first_tweet = True
    start_new = False
    for line in f:
        i += 1

        # skip header
        if i == 1:
            continue

        # if no new next line (end of document)
        if (not line) and (i >= 3_876_868):
            end_of_doc = True

        # find if a new tweet starts in this line
        start_of_new_tweet = re.search(r"^\d+,\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\+\d{2}:\d{2},\d+,", line)

        if end_of_doc or start_of_new_tweet:
            if start_of_new_tweet:
                # this is for the "new" tweet
                i_start = start_of_new_tweet.end()
                new_start = line[:i_start]
                new_content = line[i_start:]

            # the start of a "new" tweet marks the end of the "previous" tweet
            if i>2: # but not for the very first tweet
                if match := re.search(r",(?:.(?!,))+$", content, re.DOTALL):
                    last_comma_index = match.start() # Retrieve the index of the last comma

                    end = content[last_comma_index:]
                    content = content[:last_comma_index]

                    # handle "
                    matches = re.findall(r'^(\"+)', content)
                    if not matches:
                        content = '"' + content + '"'

                    # write "previous" tweet to file
                    with open("./data/tweets/tweets_fixed.csv", "a", encoding="UTF-8") as f:
                        f.write(start + content + end)

            if start_of_new_tweet:
                start = new_start
                content = new_content

        else:
            content += line # the currrent line is part of the content of the current tweet
```

```
Couldn't find program: 'false'
```

2.4 Merging Dataframes

⏩ Finally, I can load the properly formatted `tweets_fixed.csv` -file into a `pandas` -dataframe.

In [ ]:
```
# read "./data/tweets/tweets_fixed.csv" into dataframe
tweets = pd.read_csv(
    filepath_or_buffer = "./data/tweets/tweets_fixed.csv",
    encoding = "UTF-8",
    sep = ",",
    on_bad_lines='warn',
    dtype = {
        "politician_id": "Int64",
        "date": "object",
        "tweet_id": "object",
        "content": "object",
        "username": "object"
    }
)
```

Let's add the respective party affiliation of each candidate to the dataframe containing the tweets.

In [ ]:
```
# Merge df1 and df2 based on the common column "politician_id" and "id"
tweets = tweets.merge(politicians[["id", "party"]], left_on="politician_id", right_on="id", how="left")

# Drop the redundant "id" column
tweets.drop(columns=["id"], axis=1, inplace=True)
```

3 Exploration

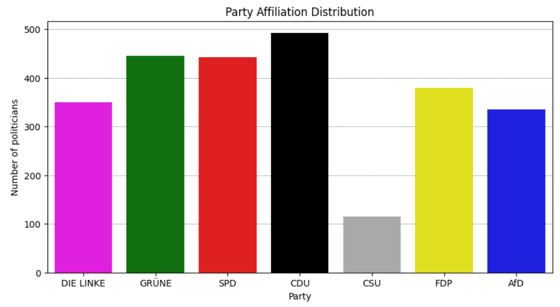
First, let's explore the data a bit.

```
In [ ]: party_order = ["DIE LINKE", "GRÜNE", "SPD", "CDU", "CSU", "FDP", "AfD"]
        colors = ["magenta", "green", "red", "black", "darkgrey", "yellow", "blue"]
```

3.1 Party Affiliation Distribution

Let's see how many politicians we have for each party.

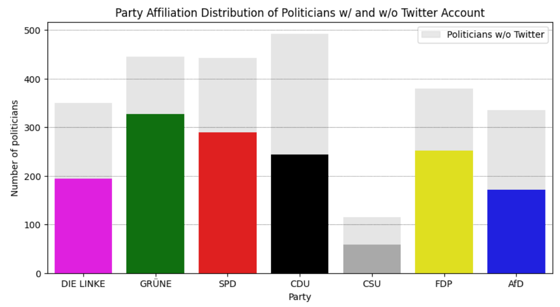
```
In [ ]: # plot the distribution of party affiliations in politicians
        fig, ax = plt.subplots(figsize=(10, 5))
        ax.grid(True, color="black", linestyle=":", linewidth=0.5)
        sns.countplot(x="party", data=politicians, order=party_order, palette=colors, zorder=3)
        ax.set_title("Party Affiliation Distribution")
        ax.set_xlabel("Party")
        ax.set_ylabel("Number of politicians")
        plt.show()
```



Now this contains simply all politicians, regardless of whether they have a Twitter account or not. Let's filter out those who don't have one.

```
In [ ]: politicians_with_twitter = politicians[~politicians["user_id1"].isna() | ~politicians["user_id2"].isna()]
```

In [ ]:
```python
fig, ax = plt.subplots(figsize=(10, 5))

ax.grid(True, color="black", linestyle=":", linewidth=0.5)

# Plot transparent bars from politicians dataset
sns.countplot(x="party", data=politicians, order=party_order, color="none", ax=ax, alpha=0.1, zorder=3)

# Plot bars from politicians_with_twitter dataset
sns.countplot(x="party", data=politicians_with_twitter, order=party_order, palette=colors, ax=ax, zorder=3)

ax.set_title("Party Affiliation Distribution of Politicians w/ and w/o Twitter Account")
ax.set_xlabel("Party")
ax.set_ylabel("Number of politicians")

# Create a custom legend for the transparent bars
transparent_patch = mpatches.Patch(color='lightgrey', alpha=0.5, label='Politicians w/o Twitter')
ax.legend(handles=[transparent_patch])

plt.show()
```



Apart from the `CSU`, we have a pretty even distribution of politicians with Twitter per party. Despite being second in total politicians, the `GRÜNE` has the most politicians with Twitter. This is probably due to the fact that they are the youngest party in the Bundestag, and thus more likely to have a Twitter account. Interestingly, the `FDP`, which is the fourth in total candidates, has more politicians with Twitter than the `CDU`, which is the largest party in the Bundestag even though the mean age is roughly the same for these two.

On average, 59% of all politicians have a Twitter account.

In [ ]:
```python
# calculate mean % of politcians with twitter account per party
percentage_of_politicians_with_twitter = politicians_with_twitter.groupby("party").size() / politicians.groupby("party").size() * 100

print(f"{round(percentage_of_politicians_with_twitter.mean())}% of all politicians have a twitter account.")
```

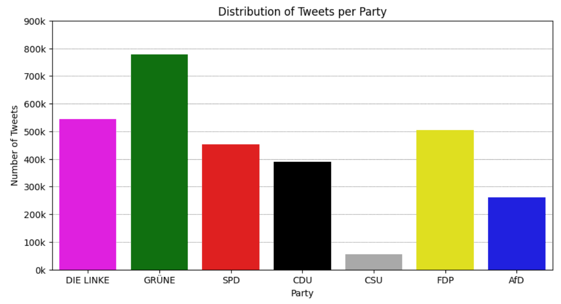
```
59% of all politicians have a twitter account.
```

This will yield enough data to work with.

3.2 Tweet Distribution per Party

Let's see how much each party tweets.

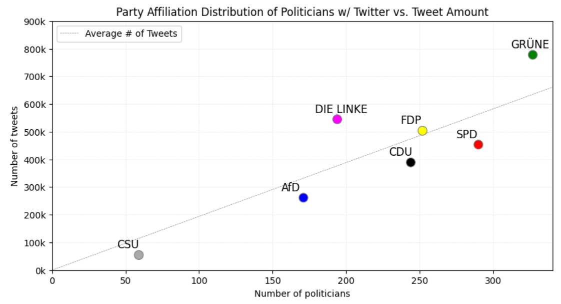
```
In [ ]: # plot the distribution of party affiliations in merged tweets
fig, ax = plt.subplots(figsize=(10, 5))
ax.grid(True, color="black", linestyle=":", linewidth=0.5)
sns.countplot(x="party", data=tweets, order=party_order, ax=ax, palette=colors, zorder=3)
ax.set_title("Distribution of Tweets per Party")
ax.set_xlabel("Party")
ax.set_ylabel("Number of Tweets")

# Modify the y-axis tick labels
y_ticks = ax.get_yticks()
ax.set_yticks(y_ticks)
ax.set_yticklabels([f"{int(tick/1000)}k" for tick in y_ticks])

plt.show()
```



Leading in total tweets, is the `GRÜNE` . We can see differences in the relation of tweets per party and politicians per party. `DIE LINKE` , for example, comes fifth in politicians, but second in total tweets. To visualize this, let's plot those two variables against each other.

```
In [ ]: y = dict(tweets["party"].value_counts())
y = [y[k] for k in party_order] # reorder y to match party_order

x = dict(politicians_with_twitter["party"].value_counts())
x = [x[k] for k in party_order] # reorder x to match party_order
```

```
In [ ]: # scatter plot the distribution of party affiliations in politicians vs. tweets
        fig, ax = plt.subplots(figsize=(10, 5))
        sns.scatterplot(x=x, y=y, hue=party_order, palette=colors, ax=ax, s=100, edgecolor="grey", zorder=3)

        # add annotations
        ax.legend_.remove()
        for i, txt in enumerate(party_order):
            ax.annotate(txt, (x[i], y[i]), fontsize=12,
                        xytext=(x[i]-15, y[i]+25_000))

        ax.grid(True, color="lightgray", linestyle=":", linewidth=0.5)
        ax.set_title("Party Affiliation Distribution of Politicians w/ Twitter vs. Tweet Amount")
        ax.set_xlabel("Number of politicians")
        ax.set_ylabel("Number of tweets")

        # Modify the y-axis tick labels
        y_ticks = ax.get_yticks()
        ax.set_yticks(y_ticks)
        ax.set_yticklabels([f"{int(tick/1000)}k" for tick in y_ticks])

        # Set the limits for x-axis and y-axis
        ax.set_xlim(0)
        ax.set_ylim(0)

        # add avgerage line
        slope = tweets.shape[0] / politicians_with_twitter.shape[0] # average number of tweets per politician
        x_line = np.array(ax.get_xlim())
        y_line = slope * x_line
        ax.plot(x_line, y_line, linestyle='--', color='grey', linewidth=0.5, zorder=0)
        avg_line_legend = mlines.Line2D([], [], linestyle='--', color='grey', linewidth=0.5, label='Average # of Tweets')
        ax.legend(handles=[avg_line_legend], loc='upper left')

        plt.show()
```



On average, `DIE LINKE` and `GRÜNE` tweet more than the other parties. With FDP marking the middle ground, and everyone else tweeting less. I'd guess that this phenomenon is probably also due to the fact that the `GRÜNE` and `DIE LINKE` are the youngest parties in the Bundestag on average.

3.3 Most Frequent Words

To get a first impression of the Twitter data, let's look at the most frequent words in the tweets.

```
In [ ]: sw = nltk.corpus.stopwords.words("german")
        sw += ["http", "https", "amp", "ja", "schon", "mal", "mehr", "rt"] # add some more stopwords
        sw = set(sw) # convert to set for faster lookup
```

To get meaningful results, we will have to preprocess the tweets first to remove stop words and punctuation. This is only for exploration purposes, so we will not do any more sophisticated preprocessing here. The proper preprocessing will be done in the respective section later on.

```
In [ ]: punctuation = list(string.punctuation) + ['-', '--', '..', '...', '“', '„', "''", '``', '+++'] # add some more punctuation
```

I've disabled the cell below as it takes quite some time to complete, but stored the results. Skip to the next cell, marked by '⏩', to see them.

```
In [ ]: %%script false --no-raise-error
        from collections import Counter

        # calculate the top 10 most frequent words (excluding stop words) for each party
        top_words = {}
        for party in party_order:
            # filter the tweets by party
            party_tweets = tweets[tweets["party"] == party]

            # tokenize the tweets
            tokens = [word_tokenize(tweet) for tweet in party_tweets["content"]]

            # flatten the list of tokens
            tokens = [item for sublist in tokens for item in sublist]

            # filter out stop words
            tokens = [token for token in tokens if (token.lower() not in sw) and (token not in punctuation)]

            # count the tokens
            token_counts = Counter(tokens)

            # get the top 100 most frequent words and add to the dictionary
            top_words[party] = token_counts.most_common(100)
```

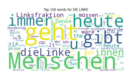
```
Couldn't find program: 'false'
```

```
In [ ]: %%script false --no-raise-error
        # show word clouds for the most frequent words for each party
        from wordcloud import WordCloud
        for party in party_order:
            wordcloud = WordCloud(background_color="white", width=800, height=400, max_words=100).generate_from_frequencies(dict(top_words[party]))
            plt.figure(figsize=(8, 4))
            plt.imshow(wordcloud, interpolation="bilinear")
            plt.axis("off")
            plt.title(f"Top 100 words for {party}")
            plt.show()
```

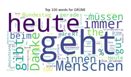
```
Couldn't find program: 'false'
```

⏩ Let's take a look at the resulting word clouds from the `pictures` -folder:

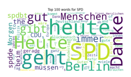
Linke



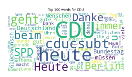
Grüne



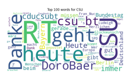
SPD



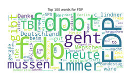
CDU



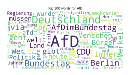
CSU



FDP



AfD



One commonality between all parties is the frequent use of the word 'heute' (today). This probably stems from the fact that the tweets were talking about current events. Unsurprisingly, many parties also talk about their own party and their respective candidates a lot. For example, the mention of prominent party members, such as "Christian Lindner (`c_lindner`)" for `FDP` , or "Dorothee Bär (`DoroBaer`)" for `CDU` , is quite common. This will help in identifying the tweets later on.

🤔 On another note, it would be interesting to train a classifier solely on the relationships between politicians, i.e. the mentions, follows, etc. and see how well it performs.

4 Baseline

4.1 Approach

For the baseline, I've decided to use a simple bag-of-words approach with a naive bayes classifier. A bag-of-words model represents text as a collection or "bag" of individual words, disregarding grammar, word order, and context.

In [ ]:
```
# train test split
X_train, X_test, y_train, y_test = train_test_split(tweets["content"], tweets["party"], test_size=0.2, random_state=42)
```

In [ ]:
```
# train a basic classifier as baseline
pipe = Pipeline([
    ("vect", CountVectorizer()),
    ("nb", MultinomialNB()),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

In [ ]:
```
# evaluate the model
cr_baseline = classification_report(y_test, y_pred, target_names=party_order)
print(cr_baseline)
```

```
              precision    recall  f1-score   support

   DIE LINKE       0.75      0.52      0.62     52192
       GRÜNE       0.63      0.46      0.53     77852
         SPD       0.81      0.04      0.08     10823
         CDU       0.67      0.63      0.65    109180
         CSU       0.59      0.61      0.60    100887
         FDP       0.51      0.78      0.62    155226
         AfD       0.68      0.46      0.55     90660

    accuracy                           0.60    596820
   macro avg       0.66      0.50      0.52    596820
weighted avg       0.62      0.60      0.59    596820
```

With these results, there's a lot to unpack. Overall, it looks as we have fairly decent results for all classes. The model is able to predict the correct party affiliation in 60% of the cases (accuracy). Recall that:

$$accuracy = \frac{TP + FP}{TP + TN + FP + FN}$$

With a 81% precision, it seems that the model works best for `SPD` tweets. Recall that precision is defined as:

$$precision = \frac{TP}{TP + FP}$$

However, with a recall value of only 4% the issue becomes apparent: The model is simply very picky about which tweets get classified as `SPD` (only few False Positives). After all, recall is defined as:

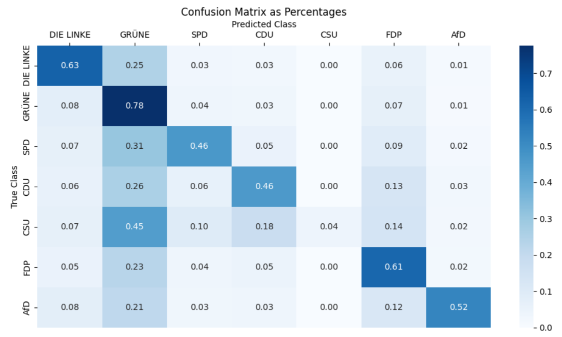
$$recall = \frac{TP}{TP + FN}$$

And it shows the big amount of False Negatives (tweets that should have been classified as `SPD` but weren't).

We can further visualize this by looking at the confusion matrix for our baseline's predictions.

In [ ]:
```
cm_baseline = confusion_matrix(y_test, y_pred, labels=party_order)
cm_baseline = cm_baseline / cm_baseline.sum(axis=1).reshape(-1, 1)
```

In [ ]:
```
# plot the confusion matrix as percentages
fig, ax = plt.subplots(figsize=(12, 6))
sns.heatmap(cm_baseline, annot=True, fmt=".2f", cmap="Blues", ax=ax)
ax.set_xlabel("Predicted Class")
ax.set_ylabel("True Class")
ax.set_title("Confusion Matrix as Percentages")
ax.set_xticklabels(party_order)
ax.set_yticklabels(party_order)
ax.xaxis.tick_top()
ax.xaxis.set_label_position('top')
plt.show()
```



We can see that the model can hardly differentiate a `SPD` tweet from a `GRÜNE` tweet. One might think that this is because both parties are left-wing parties and tend to share many of the same political views. But it just seems that the model is biased towards tweets from the `GRÜNE` .

This shows even clearer in the high number of False Positives. Every party is predicted to be `GRÜNE` more often than any other party (except for itself). We can see that our model predicts `GRÜNE` 39% of the time, while the actual number of `GRÜNE` tweets is only 25%.

In [ ]:
```
grüne_pred = np.sum(y_pred == 'GRÜNE')/ len(y_pred)
print(f"Predicted GRÜNE: {grüne_pred:.2%}")
```

```
Predicted GRÜNE: 39.33%
```

In [ ]:
```
# true number of grüne tweets:
grüne_true = np.sum(y_test == 'GRÜNE')/ len(y_test)
print(f"True GRÜNE: {grüne_true:.2%}")
```

```
True GRÜNE: 26.01%
```

For tweets from the `CSU` , this is extreme. The confusion matrix also shows things neither precision nor recall can: The model is simply not able to predict `CSU` tweets.

I strongly suspect, this is because the `CSU` is a regional party that only operates in Bavaria and as we saw in the previous section (cf. 3 Exploration), it is very underrepresented in our dataset, which probably lead to the model not being able to learn anything about it.

5 Preprocessing

To keep it simple, we'll preprocess on the `tweets` -dataframe and later redo the train-test-split on the preprocessed dataframe with the same `random_state` as before (cf. 4 Baseline). Thus ensuring that the splits are equal and that we can compare the results of the classifier.

Similarly, as before, I've disabled the cells below as they take quite some time to complete, but stored the results. Skip to the next cell, marked by "⏩", to the results them.

5.1 URL Removal

As URLs without access to their actual content are not very informative, let's remove them.

```
In [ ]: %%script false --no-raise-error
        def remove_urls(text):
            res = re.sub(r"https?:\/\/\S+|www\.\S+", "", text)
            return res

        tweets["pp_content"] = tweets.apply(lambda row: remove_urls(row["content"]), axis=1)
```

5.2 HTML Character Substitution

Due to scraping the tweets from the Twitter website, some HTML characters are still present in the tweets. For example `&amp;` instead of `&`. Let's replace those.

```
In [ ]: %%script false --no-raise-error
        tweets["pp_content"] = tweets.apply(lambda row: html.unescape(row["pp_content"]), axis=1)
```

5.3 Emoji Removal

For simplicity, I'll just remove all emojis from the tweets. For tasks such as Sentiment Analysis this might not be such a good idea, as emojis are often used to express emotions. In our classification task, however, I don't think that emojis are that important.

In [ ]:
```
%%script false --no-raise-error
# Reference : https://gist.github.com/slowkow/7a7f61f495e3dbb7e3d767f97bd7304b
def remove_emoji(string):
    emoji_pattern = re.compile("["
                               u"\U0001F600-\U0001F64F"  # emoticons
                               u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                               u"\U0001F680-\U0001F6FF"  # transport & map symbols
                               u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                               u"\U00002500-\U00002BEF"  # chinese char
                               u"\U00002702-\U000027B0"
                               u"\U00002702-\U000027B0"
                               u"\U000024C2-\U0001F251"
                               u"\U0001f926-\U0001f937"
                               u"\U00010000-\U0010ffff"
                               u"\u2640-\u2642"
                               u"\u2600-\u2B55"
                               u"\u200d"
                               u"\u23cf"
                               u"\u23e9"
                               u"\u231a"
                               u"\ufe0f"  # dingbats
                               u"\u3030"
                               "]+", flags=re.UNICODE)
    return emoji_pattern.sub(r'', string)

tweets["pp_content"] = tweets.apply(lambda row: remove_emoji(row["pp_content"]), axis=1)
```

5.4 Tokenization & Lemmatization

Lemmatization is a linguistic technique that reduces words to their base or root form, known as a lemma. It aims to transform different forms of a word, such as plurals or verb conjugations, into a common base form. For example, lemmatization would convert "running," "runs," and "ran" to the lemma "run."

This technique is useful for text classification because it helps reduce the dimensionality of the feature space by grouping together different forms of the same word. By treating related words as a single entity, it can improve the accuracy and generalization of text classification models. Lemmatization can also help to eliminate noisy variations and normalize the text, which can improve the performance of algorithms that rely on word frequencies or patterns.

Additionally, lemmatization helps to overcome the limitations of stemming, another word normalization technique. Unlike stemming, which simply chops off word suffixes, lemmatization uses knowledge of the word's meaning and context to determine the proper base form. This makes lemmatization more accurate and suitable for tasks like text classification that require a deeper understanding of the text.

This is where we'll use the open-source library spaCy, as it has a good German model and is very easy to use. Due to the high computational cost of lemmatization, we'll only use the small version of the model.

In [ ]:
```
%%script false --no-raise-error
tweets["pp_content"] = [[token.lemma_ for token in row] for row in lem.pipe(tweets["pp_content"])]
```

5.5 Removing Stopwords & Punctuation

Stopwords are commonly occurring words in a language that typically do not carry much meaning or contribute to the overall message or context of a sentence. Examples of stopwords in English include "the," "and," "is," "in," and so on.

For this we'll reuse the `punctuation` - and `sw` -variables from earlier (cf. 4.2 Most Frequent Words).

In [ ]:
```
%%script false --no-raise-error
tweets["pp_content"] = tweets.apply(lambda row: [token for token in row["pp_content"] if (token.lower() not in sw) and (token not in punctuation)], ax
```

5.6 Removing Empty Tweets

For simplicity's sake, we'll remove all tweets that are empty after preprocessing. For example, if a tweet only consists of a URL or an emoji, it will be removed.

In [ ]:
```
%%script false --no-raise-error
# remove empty elements
tweets["pp_content"] = tweets.apply(lambda row: [token for token in row["pp_content"] if len(token.strip()) > 0], axis=1)

# remove rows with empty content
tweets = tweets[tweets.apply(lambda row: len(row["pp_content"]) > 0, axis=1)]
```

5.7 Lowercasing

Lowercasing is a frequently used technique in text preprocessing. Its purpose is to convert all text to the same casing format, treating "this", "This", and "TiHs" as equivalent.

This technique may not be beneficial for tasks such as Part-of-Speech tagging or Sentiment Analysis, where proper casing can provide information about nouns and other linguistic elements. For example, "TeXt iN tHis FoRm" is called Alternating Caps and is used to convey sarcasm or mockery. In such cases, lowercasing may be detrimental to the model's performance.

For the use case of this project, lowercasing is beneficial as it reduces the number of unique words in the corpus, which in turn reduces the number of features in the model and thus reduces the training time.

In [ ]:
```
%%script false --no-raise-error
tweets["pp_content"] = tweets.apply(lambda row: [token.lower() for token in row["pp_content"]], axis=1)
```

5.8 CSV Export

To avoid repeating the preprocessing steps every time, we will save the preprocessed dataframe to a `.csv` file. This way, we can easily access the preprocessed data later on without having to reprocess it from scratch.

```
In [ ]: %%script false --no-raise-error

        # convert the list of tokens back to a string as else this would lead to issues with loading the csv later on
        pp_tweets["pp_content"] = pp_tweets["pp_content"].apply(lambda x: " ".join(x))
        # store to csv
        tweets.to_csv("./data/tweets/pp_tweets.csv")
```

6 Content Similiarity

```
In [ ]: pp_tweets = pd.read_csv("./data/tweets/pp_tweets.csv",
            encoding = "UTF-8",
            sep = ",",
            on_bad_lines='warn',
            dtype = {
                "politician_id": "Int64",
                "date": "object",
                "tweet_id": "object",
                "content": "object",
                "username": "object",
                "pp_content": "object",
            }
        ).set_index("index")
```

We'll drop the columns that do not contain values, as their fraction is so little that their absence won't affect the model's performance.

```
In [ ]: pp_tweets.dropna(inplace=True)
```

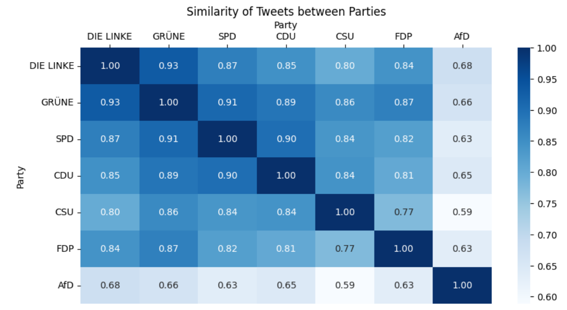
⏩ Now that we have a preprocessed dataframe, we can start with the actual task of this project: Finding similar tweets. For this we'll start of with calculating the Cosine Similarity for each tweet. In order to do that we first transform the tweets into a TF-IDF matrix.

In [ ]:
```
docs = []
for p1 in party_order:
    t = pp_tweets[pp_tweets["party"] == p1].copy()
    # Convert the values in the pp_content column to strings using .loc
    t.loc[:, "pp_content"] = t["pp_content"].astype(str)
    # Concatenate the tweets of party p1 into one string
    t = " ".join(t["pp_content"])
    docs.append(t)
```

In [ ]:
```
tfidf = TfidfVectorizer().fit_transform(docs)
pairwise_similarity = tfidf * tfidf.T
```

In [ ]:
```
# pairwise_similarity to df
tfidf_similarity_df = pd.DataFrame(pairwise_similarity.toarray(), index=party_order, columns=party_order)

# to csv
tfidf_similarity_df.to_csv("./data/tweets/tfidf_similarity.csv")
```

In [ ]:
```
# plot as heat map
fig, ax = plt.subplots(figsize=(10, 5))
sns.heatmap(tfidf_similarity_df, annot=True, fmt=".2f", cmap="Blues", ax=ax)
ax.set_xlabel("Party")
ax.set_ylabel("Party")
ax.set_title("Similarity of Tweets between Parties")
ax.set_xticklabels(party_order)
ax.set_yticklabels(party_order)
ax.xaxis.tick_top()
ax.xaxis.set_label_position('top')
plt.show()
```



We can nicely see the dis-/similarity of our tweets in the heatmap of the Cosine Similarity matrix. Recall that for Cosine Similarity, the closer the value is to 1, the more similar the tweets are.

Overall the tweets are somewhat alike, ranging from a 0.59 (`CSU - AfD`) to 0.93 score (`DIE LINKE - GRÜNE`). This is overall similarity is not that surprising, as the tweets likely cover similar events in the political and social sphere.

What can nicely be seen, is the gradual decrease in similarity from the left to the right. This is due to the fact that the tweets are ordered by their political orientation, with the most left-wing party (`DIE LINKE`) on the left and the most right-wing party (`AfD`) on the right.

Here, we can perhaps see one of the reasons, why the baseline classifier struggled with bias towards the `GRÜNE` . Content-wise, `GRÜNE` is the most similar one for almost every other party. Exceptions only being `CDU` $\rightarrow$ `SPD` $(0.01$ margin) and `AfD` $\rightarrow$ `LINKE` $(0.02$ margin).

I would not presume to interpret as to why they are so similar to everyone. I'd much rather leave that to the political scientists.

7 Classification

In [ ]:
```
# train-test split
X_train, X_test, y_train, y_test = train_test_split(pp_tweets["pp_content"], pp_tweets["party"], test_size=0.2, random_state=42)
```

7.1 Naive Bayes

7.1.1 Count Vectorizer

First, we'll try a Naive Bayes classifier as before (cf. 4 Baseline) to directly see the results of the preprocessing steps.

In [ ]:
```
# train a basic classifier as baseline
pipe = Pipeline([
    ("vect", CountVectorizer()),
    ("nb", MultinomialNB()),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

In [ ]:
```
# evaluate the model
cr_nb = classification_report(y_test, y_pred, target_names=party_order)
print(cr_nb)
```

```
              precision    recall  f1-score   support

   DIE LINKE       0.66      0.61      0.64     51224
       GRÜNE       0.59      0.53      0.56     76915
         SPD       0.77      0.10      0.18     10544
         CDU       0.67      0.65      0.66    107946
         CSU       0.61      0.62      0.61     99553
         FDP       0.57      0.72      0.64    154178
         AfD       0.64      0.52      0.57     89375

    accuracy                           0.61    589735
   macro avg       0.64      0.54      0.55    589735
weighted avg       0.62      0.61      0.61    589735
```

If you do not want to run the code yourself, I've included the results in the table below.

Class	Precision	Recall	F1-Score	Support
DIE LINKE	0.66	0.61	0.64	51224
GRÜNE	0.59	0.53	0.56	76915
SPD	0.77	0.10	0.18	10544
CDU	0.67	0.65	0.66	107946
CSU	0.61	0.62	0.61	99553
FDP	0.57	0.72	0.64	154178
AfD	0.64	0.52	0.57	89375

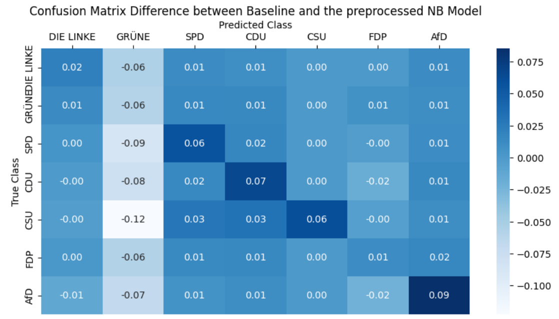
Accuracy			**0.61**	
Macro Avg	0.64	0.54	0.55	589735
Weighted Avg	0.62	0.61	0.61	589735

By simple preprocessing, we've already improved the accuracy by 1 %. This is a good start, but we surely can do better.

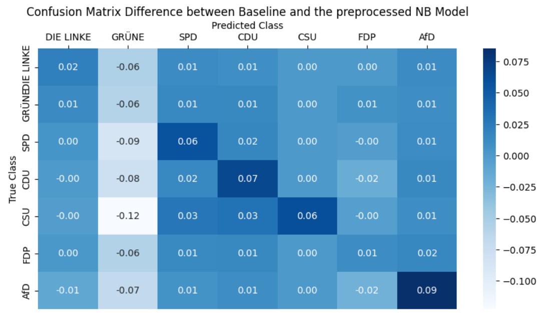
```
In [ ]: # calculate the confusion matrix as percentages
        cm_nb = confusion_matrix(y_test, y_pred, labels=party_order)
        cm_nb = cm_nb / cm_nb.sum(axis=1).reshape(-1, 1)
```

Let's view the differences in the two Confusion Matrices to see the effect of the preprocessing steps. For this we define a function we'll use later on as well.

In [ ]:
```python
def plot_confusion_matrix_diff(cm1, cm2, labels, title):
    cm_diff = cm1 - cm2
    fig, ax = plt.subplots(figsize=(10, 5))
    sns.heatmap(cm_diff, annot=True, fmt=".2f", cmap="Blues", ax=ax)
    ax.set_xlabel("Predicted Class")
    ax.set_ylabel("True Class")
    ax.set_title(title)
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    ax.xaxis.tick_top()
    ax.xaxis.set_label_position('top')
    plt.show()

# plot the difference between the confusion matrix of the baseline model and the tfidf model
plot_confusion_matrix_diff(cm_nb, cm_baseline, party_order, "Confusion Matrix Difference between Baseline and the preprocessed NB Model")
```



If you do not want to run the code yourself, I've included the plot in the cell below.



What instantly becomes visible is the decrease bias towards the `GRÜNE`. This is good news! Let's see if we can improve the model even further. We also improved the accuracy for almost every party (cf. main diagonal).

7.1.2 TF-IDF Vectorizer

Let's compare the difference in the result between different vectorizers. For this we'll use the TF-IDF vectorizer. The difference between the two vectorizers is that the TF-IDF vectorizer takes into account the frequency of the words in the corpus.

Term frequency (`TF`) measures how frequently a word appears in a document. The inverse document frequency (`IDF`) decreases proportionally to the number of documents in a corpus that contain said word. A high `IDF` value therefore means that a word is very unique/specific, as it does not appear in many documents.

In short, TF-IDF measures the importance of a term for a document, relative to the entire corpus.

In [ ]:
```
# train a basic classifier with different vectorizer
pipe = Pipeline([
    ("vect", TfidfVectorizer()),
    ("nb", MultinomialNB()),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

Let's look at the results.

In [ ]:
```
cm_nb_tfidf = confusion_matrix(y_test, y_pred, labels=party_order)
cm_nb_tfidf = cm_nb_tfidf / cm_nb_tfidf.sum(axis=1).reshape(-1, 1)
```

In [ ]:
```
cr_nb_tfidf = classification_report(y_test, y_pred, target_names=party_order)
print(cr_nb_tfidf)
```

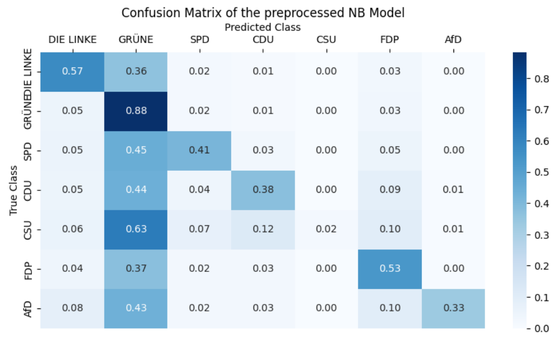
```
              precision    recall  f1-score   support

   DIE LINKE       0.89      0.33      0.49     51224
       GRÜNE       0.73      0.38      0.50     76915
         SPD       0.93      0.02      0.04     10544
         CDU       0.71      0.57      0.64    107946
         CSU       0.67      0.53      0.60     99553
         FDP       0.43      0.88      0.58    154178
         AfD       0.75      0.41      0.53     89375

    accuracy                           0.57    589735
   macro avg       0.73      0.45      0.48    589735
weighted avg       0.66      0.57      0.56    589735
```

If you do not want to run the code yourself, I've included the results in the table below.

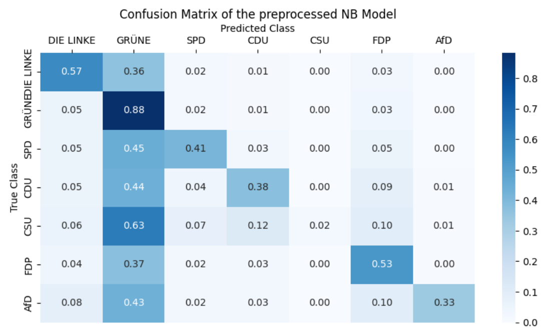
Class	Precision	Recall	F1-Score	Support
DIE LINKE	0.89	0.33	0.49	51224
GRÜNE	0.73	0.38	0.50	76915
SPD	0.93	0.02	0.04	10544
CDU	0.71	0.57	0.64	107946
CSU	0.67	0.53	0.60	99553
FDP	0.43	0.88	0.58	154178
AfD	0.75	0.41	0.53	89375

Accuracy			0.57	
Macro Avg	0.73	0.45	0.48	589735
Weighted Avg	0.66	0.57	0.56	589735

```
In [ ]: fig, ax = plt.subplots(figsize=(10, 5))
        sns.heatmap(cm_nb_tfidf, annot=True, fmt=".2f", cmap="Blues", ax=ax)
        ax.set_xlabel("Predicted Class")
        ax.set_ylabel("True Class")
        ax.set_title("Confusion Matrix of the preprocessed NB Model")
        ax.set_xticklabels(party_order)
        ax.set_yticklabels(party_order)
        ax.xaxis.tick_top()
        ax.xaxis.set_label_position('top')
        plt.show()
```



If you do not want to run the code yourself, I've included the plot in the cell below.



This didn't work out as well as I had hoped. The accuracy *decreased* by 1 % and the bias towards the `GRÜNE` is still very present. Let's see if we can improve the model by using a different classifier.

7.2 MLP Classifier

Let's try more sophisticated models. For this we'll use a Multi-Layer Perceptron (MLP) classifier. This is a feedforward neural network that maps sets of input data onto a set of appropriate outputs. It consists of multiple layers of nodes, hence the name. Each layer is fully connected to the next one.

MLPs can learn meaningful representations directly from the raw text data. Through the training process, the model can discover relevant features and representations that are most informative for the classification task. In contrast, Naive Bayes models might not capture the full complexity of the data.

In [ ]:
```
# train a MLP classifier
pipe = Pipeline([
    ("vect", TfidfVectorizer()),
    ("mlp", MLPClassifier()),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

```
c:\Users\schef\Entwicklung\Projekte\Python\Environments\ie694_iaoai\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: UserWarni
ng: Training interrupted by user.
  warnings.warn("Training interrupted by user.")
```

Due to time constraints I had to manually stop the training after 6h. The model was still improving, but I had to stop it at some point. Let's see how it performs.

In [ ]:
```
cm_mlp = confusion_matrix(y_test, y_pred, labels=party_order)
cm_mlp = cm_mlp / cm_mlp.sum(axis=1).reshape(-1, 1)
```

```
In [ ]: cr_mlp = classification_report(y_test, y_pred, target_names=party_order)
        print(cr_mlp)
```

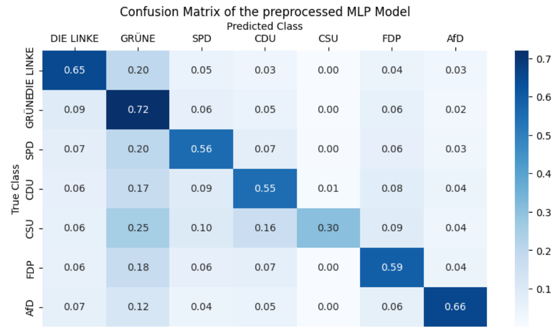
```
              precision    recall  f1-score   support

   DIE LINKE       0.66      0.66      0.66     51224
       GRÜNE       0.60      0.55      0.57     76915
         SPD       0.60      0.30      0.40     10544
         CDU       0.67      0.65      0.66    107946
         CSU       0.67      0.59      0.63     99553
         FDP       0.58      0.72      0.64    154178
         AfD       0.62      0.56      0.59     89375

    accuracy                           0.63    589735
   macro avg       0.63      0.58      0.59    589735
weighted avg       0.63      0.63      0.62    589735
```

If you do not want to run the code yourself, I've included the results in the table below.

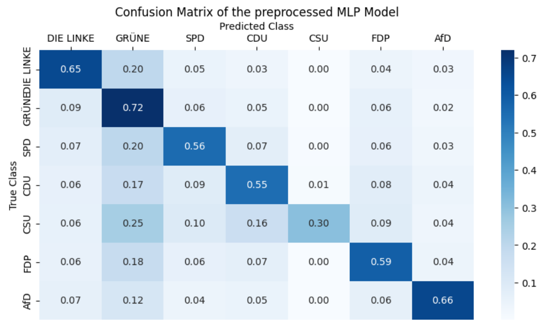
Class	Precision	Recall	F1-Score	Support
DIE LINKE	0.66	0.66	0.66	51224
GRÜNE	0.60	0.55	0.57	76915
SPD	0.60	0.30	0.40	10544
CDU	0.67	0.65	0.66	107946
CSU	0.67	0.59	0.63	99553
FDP	0.58	0.72	0.64	154178
AfD	0.62	0.56	0.59	89375

Accuracy			**0.63**	
Macro Avg	0.63	0.58	0.59	589735
Weighted Avg	0.63	0.63	0.62	589735

```
In [ ]: # plot the confusion matrix
        fig, ax = plt.subplots(figsize=(10, 5))
        sns.heatmap(cm_mlp, annot=True, fmt=".2f", cmap="Blues", ax=ax)
        ax.set_xlabel("Predicted Class")
        ax.set_ylabel("True Class")
        ax.set_title("Confusion Matrix of the preprocessed MLP Model")
        ax.set_xticklabels(party_order)
        ax.set_yticklabels(party_order)
        ax.xaxis.tick_top()
        ax.xaxis.set_label_position('top')
        plt.show()
```



If you do not want to run the code yourself, I've included the plot in the cell below.



What the MLP classifier has done nicely, is the results for the minority class `CSU` . It is now finally able to classify the majority of the `CSU` tweets correctly. However, the overall bias towards the `GRÜNE` is still present.

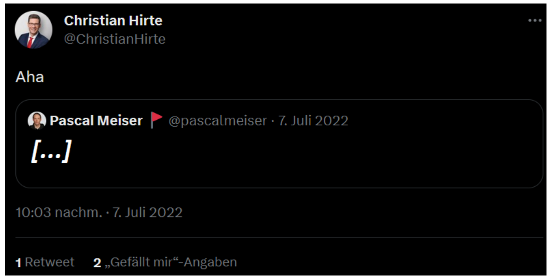
8 Conclusion

Return to Table of Contents

This project shows that it is possible to classify tweets by their political orientation with a high accuracy. We showed that proper preprocessing is beneficial for the performance of the model. The best model was a MLP that achieved an accuracy of 63%.

Sadly the classification section of this project was not as extensive as I had planned it to be. The CSV-Fixing as well as the preprocessing steps took way more time than expected, leaving not much time to train more complex models.

Furthermore, it would be interesting to see the classifier's performance on more than a single tweet. Some tweets seemingly do not contain any useful information, such as this one for example:



Christian Hirte
@ChristianHirte

Aha

Pascal Meiser @pascalmeiser · 7. Juli 2022
[...]

10:03 nachm. · 7. Juli 2022

1 Retweet **2** „Gefällt mir"-Angaben

As it is a reply to another tweet, that I have no access to, it is hard to say if it contains any useful information. In theory, the results should be better I think, as the classifier would have more information to work with. Sadly, I did not have the time to implement this.

Tweets like this (that were a reply to another Tweet) maybe should also have been removed from the data set, as right now I have no access to the original tweet. This could be a possible improvement for the future. It also would be interesting to see if the classifier would be able to classify this tweet correctly, if it had access to the original tweet.

As of right now the classifier does not get information about the date, amount of retweets or likes of the tweets. It would be interesting to see if the classifier would be able to classify the tweets better, if it had that information. Potentially, someone tweeting about "refugees" in 2023 could have a different political orientation than someone tweeting about it in 2015, when everyone was talking about the ongoing refugee crisis.

Nonetheless, the results by our models are promising as a Proof-of-Concept, and I'm happy with the outcome of this project.