QIAN_1002675544_assignment3

June 10, 2021

[1]:
```
#pip install tweet-preprocessor
#pip install HTMLParser
#pip install unicodedata2
#pip install nltk
#pip install more-itertools
```

[2]:
```
import pandas as pd
import numpy as np
```

[3]:
```
import re
import html
import string
import unicodedata
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
#nltk.download('stopwords')
#nltk.download('punkt')
#nltk.download("wordnet")
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
```

[4]:
```
import os
import itertools
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import seaborn as sns
```

[5]:
```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import scipy as sp
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from imblearn.under_sampling import RandomUnderSampler
from sklearn.linear_model import LogisticRegression
from sklearn import neighbors
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
```

```
from sklearn import svm
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, log_loss
from sklearn.multiclass import OneVsRestClassifier
from sklearn import model_selection
from sklearn.utils import class_weight
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
```

[6]:
```
import plotly.express as px
import plotly.graph_objects as go
import plotly.offline as pyo
from IPython.display import HTML
import plotly.io as pio
```

[7]:
```
import warnings
warnings.filterwarnings("ignore")
```

0.1 1. Data Cleaning

- 1.1 Load datasets
- 1.2 Data Cleaning

0.1.1 1.1 Load datasets

[8]:
```
sentiment=pd.read_csv("sentiment_analysis.csv")
election=pd.read_csv("US_Elections_2020.csv")
```

[9]:
```
sentiment.head(3)
```

[9]:

	ID	text	label
0	7.680980e+17	Josh Jenkins is looking forward to TAB Breeder…	1
1	7.680980e+17	RT @MianUsmanJaved: Congratulations Pakistan o…	1
2	7.680980e+17	RT @PEPalerts: This September, @YESmag is taki…	1

[10]:
```
election.head(3)
```

[10]:

	text	sentiment	\
0	b'@robreiner so afraid of Nov, Dec, and Jan! E…	0	
1	b"RT @SueCOOK: Lord Sumption launches Recovery…	0	
2	b'RT @WalidPhares: Uber Timing: after #Biden a…	0	

```
  negative_reason
0         covid19
1          others
2         covid19
```

Notice that each tweet in the US 2020 election dataset begins with b, which stands for the data type of bytes and is meaningless; so I decide to remove it.

[11]:
```
election["text"]=election["text"].apply(lambda x: str(x)[2:])
```

[12]:
```
election.head(3)
```

[12]:
```
                                                text  sentiment  \
0  @robreiner so afraid of Nov, Dec, and Jan! Eve…          0
1  RT @SueCOOK: Lord Sumption launches Recovery -…          0
2  RT @WalidPhares: Uber Timing: after #Biden adv…          0

  negative_reason
0         covid19
1          others
2         covid19
```

0.1.2 1.2 Data Cleaning

To further process and analyze both datasets, I follow the following procedure and define each step's function to clean up the datasets' tweets. **Procedure:** * 1.2.1 Remove all html tags and attributes (i.e., /<[^>]+>/) * 1.2.2 Replace Html character codes (i.e., &…;) with an ASCII equivalent * 1.2.3 Remove all URLs * 1.2.4 All characters in the text are in lowercase. * 1.2.5 Remove the word after @ * 1.2.6 Remove non-alphabetic words, whitespaces and puctuation * 1.2.7 Remove all stop words are removed * 1.2.8 Stemming and Lemmatization

1.2.1 Remove all html tags and attributes The idea is to use a regular expression to find all the Html tags and attributes characters "/<[^>]+>/" appearing first in the text, and use the sub-function to replace all the text between these symbols with an empty string.

[13]:
```
def parse(text):
    text = re.sub(re.compile("/<[^>]+>/"),"",text)
    return text
```

[14]:
```
sentiment["clean_text"]=sentiment["text"].apply(parse)
election["clean_text"]=election["text"].apply(parse)
```

1.2.2 Html character codes (i.e., &…;) are replaced with an ASCII equivalent. Using html.unescape() function to unescape html entities in a string.

[15]:
```
def to_ascii(text):
    text = html.unescape(text)
    return text
```

[16]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(to_ascii)
election["clean_text"]=election["clean_text"].apply(to_ascii)
```

1.2.3 Remove all URLs Using regular expression operations for urls format with re.sub method to remove URLs. Once it finds urls in the text, it will automatically replace those matches with "".

[17]:
```
def remove_urls(text):
    text = re.sub(r"http\S+", "", text, flags=re.MULTILINE)
    return text
```

[18]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(remove_urls)
election["clean_text"]=election["clean_text"].apply(remove_urls)
```

1.2.4 All characters in the text are in lowercase

[19]:
```
def lower_case(text):
    return text.lower() #Convert all characters to lowercase
```

[20]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(lower_case)
election["clean_text"]=election["clean_text"].apply(lower_case)
```

1.2.5 Remove the word after @ remove the word after @ using re.compile() function.

[21]:
```
def remove_at(text):
    text = re.compile('rt @').sub('@', str(text))
    at = re.compile('@(?=\w+)\w+')
    text = re.sub(at,'', str(text))
    return text
```

[22]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(remove_at)
election["clean_text"]=election["clean_text"].apply(remove_at)
```

1.2.6 Remove non-alphabetic words, whitespaces and puctuation Remove the non-letter words and punctuations using re.compile through matching anything that is not Alphabet and not white spaces. Delete punctuations by using string.translate method to remove punctuation.

[23]:
```
def remove_noletter_ws_puc(text):
    try:
        text = re.sub(re.compile(r"[^A-Za-z]+"), " ",text) #remove non-letter
        text = re.sub(re.compile(r'\s+')," ",text) #remove whitespaces
        text = text.translate(string.maketrans("",""), string.punctuation)␣
↪#remove punctuation
    except:
        text=text
    return text
```

[24]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(remove_noletter_ws_puc)
election["clean_text"]=election["clean_text"].apply(remove_noletter_ws_puc)
```

1.2.7 Remove all stop words Stopwords refer to a set of commonly used words without unique meanings. I first define all stopwords using nltk(natural language tool kit) package and then remove them by matching with the tweets.

[25]:
```
stopword = stopwords.words('english')
def remove_stopwords(text):
    word=text.split()
    text=" ".join([temp for temp in word if temp not in stopword])
    return text
```

[26]:
```
sentiment["clean_text"]=sentiment["clean_text"].apply(remove_stopwords)
election["clean_text"]=election["clean_text"].apply(remove_stopwords)
```

1.2.8 Stemming and Lemmatization Stemming is a process of reducing inflecred words to their word stem, while lemmatization is the process of grouping together inflected forms of a word so that those words can be analyzed as a single item.

[27]:
```
ps =PorterStemmer()
lm=WordNetLemmatizer()
election["clean_text"]=election["clean_text"].apply(lambda x: " ".join([ps.
 ↪stem(w)for w in x.split()]))
election["clean_text"]=election["clean_text"].apply(lambda y: " ".join([lm.
 ↪lemmatize(w,"v")for w in y.split()]))
```

The cleaned datasets are shown as following:

[28]:
```
sentiment.head(3)
```

[28]:
```
             ID                                                text  label  \
0  7.680980e+17  Josh Jenkins is looking forward to TAB Breeder…      1
1  7.680980e+17  RT @MianUsmanJaved: Congratulations Pakistan o…      1
2  7.680980e+17  RT @PEPalerts: This September, @YESmag is taki…      1

                                          clean_text
0  josh jenkins looking forward tab breeders crow…
1  congratulations pakistan becoming testteam wor…
2  september taking maine mendoza surprise thanks…
```

[29]:
```
election.head(3)
```

[29]:
```
                                                text  sentiment  \
0  @robreiner so afraid of Nov, Dec, and Jan! Eve…          0
1  RT @SueCOOK: Lord Sumption launches Recovery -…          0
2  RT @WalidPhares: Uber Timing: after #Biden adv…          0

  negative_reason                                         clean_text
0         covid19  afraid nov dec jan even bidenharri win frump m…
1          others  lord sumption launch recoveri new initi promot…
```

```
2         covid19  uber time biden advisor say would put pressur …
```

[30]:
```
#check whether dataset contains empty tweet
print(sentiment["clean_text"].isnull().sum())
print(election["clean_text"].isnull().sum())
```

```
0
0
```

0.2 2. Exploratory Analysis

0.2.1 2.1 Determine the political party

Procedure: **2.1.1 Step 1:** I first collect all hashtags and mentions from the original text data, and combine both lists to generate a wordcloud. After that, I create two lists of keywords for republican and democratic parties based on word frequency generated by the wordcloud. **2.1.2 Step 2:** If we label the party only by the party name or mentioned leader, the result may be biased or mislabelled since some tweets does not contain those keywords but still express their selections. Therefore, I define a function to label three parties (Republican, democratic, and others) based on the frequency of keyword occurrence. The most common parties mentioned will be selected for the tweet with more than one party keyword mentioned. If the number of keywords mention are the same, I categorize it into a new category: "Republican/Democratic". Otherwise, it will belong to "other". **2.1.3 Step 3:** Visualize the political affiliations of the tweets and discuss the findings

2.1.1 Find keywords

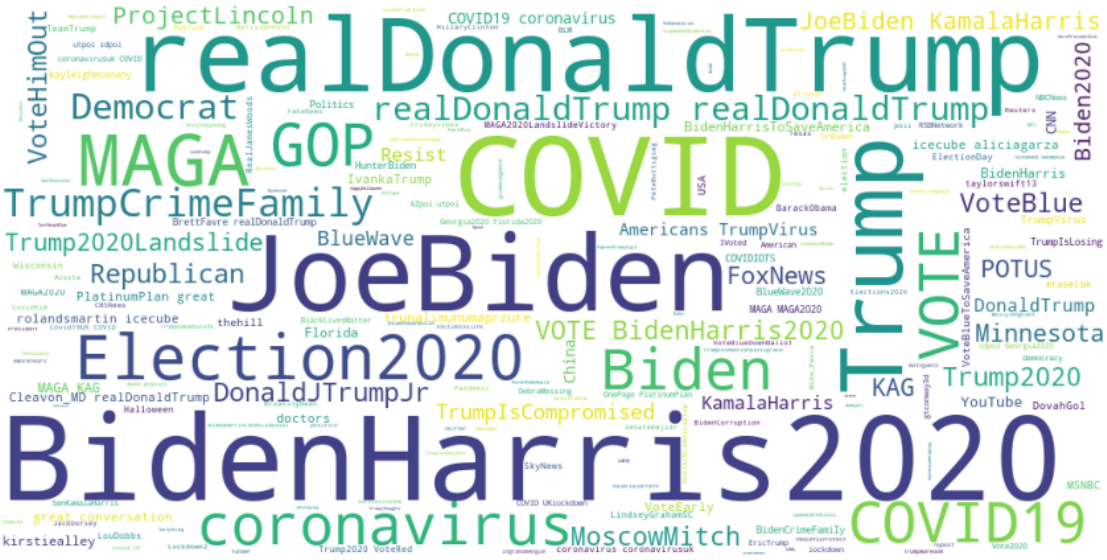
[31]:
```
election1=election.copy()
hashtag = election1['text'].map(lambda x: re.findall('#(?=\w+)\w+',x))
at = election1['text'].map(lambda x: re.findall('@(?=\w+)\w+',x))
```

[32]:
```
#combine all lists together
new_hashtag= list(itertools.chain(*hashtag))
new_at= list(itertools.chain(*at))
#remove # and @
new_hashtag = [i[1:] for i in new_hashtag]
new_at = [i[1:] for i in new_at]
taglist=new_hashtag+new_at
```

Hashtags and mentions visualization for US election dataset

[33]:
```
#convert list to string and generate
unique_string=(" ").join(taglist)
#generate a wordcloud using hashtags
wordcloud = WordCloud(width = 1000, height = 500,background_color='white').
 ↪generate(unique_string)
plt.figure(figsize=(15,8))
plt.imshow(wordcloud)
plt.axis("off")
plt.savefig("your_file_name"+".png", bbox_inches='tight')
```

```
plt.show()
plt.close()
```



2.1.2 Define a function to label the party

[34]:
```
republican_key = ['DonaldTrump', "DonaldJTrumpJr","realDonaldTrump",'Trump',␣
 ↪"Trump","republicanparty", "republican",␣
 ↪"MAGA","makeamericagreatagain","GOP", "ivankaTrump", "ericTrump","TeamTrump"]
democratic_key =␣
 ↪["democraticparty","democrats","Democrat","TheDemocrats","JoeBiden","Biden","KamalaHarris",
def label_party(text):
    party=[]
    for i in range(len(text)):
        word_tokens = word_tokenize(text[i])
        #keyword count
        rep_count = 0
        demo_count = 0

        rep_count = len([w for w in word_tokens if w in republican_key])
        demo_count = len([w for w in word_tokens if w in democratic_key])

        if rep_count > demo_count:
            party.append('Republican')

        elif demo_count > rep_count:
            party.append('Democratic')

        elif rep_count == demo_count != 0:
```

```
            party.append('Republican/Democratic')
        else:
            party.append('Other')

    return party
```

[35]:
```
#apply the label_party function
party = label_party(election1["text"].tolist())
election1['political_party'] = party
```

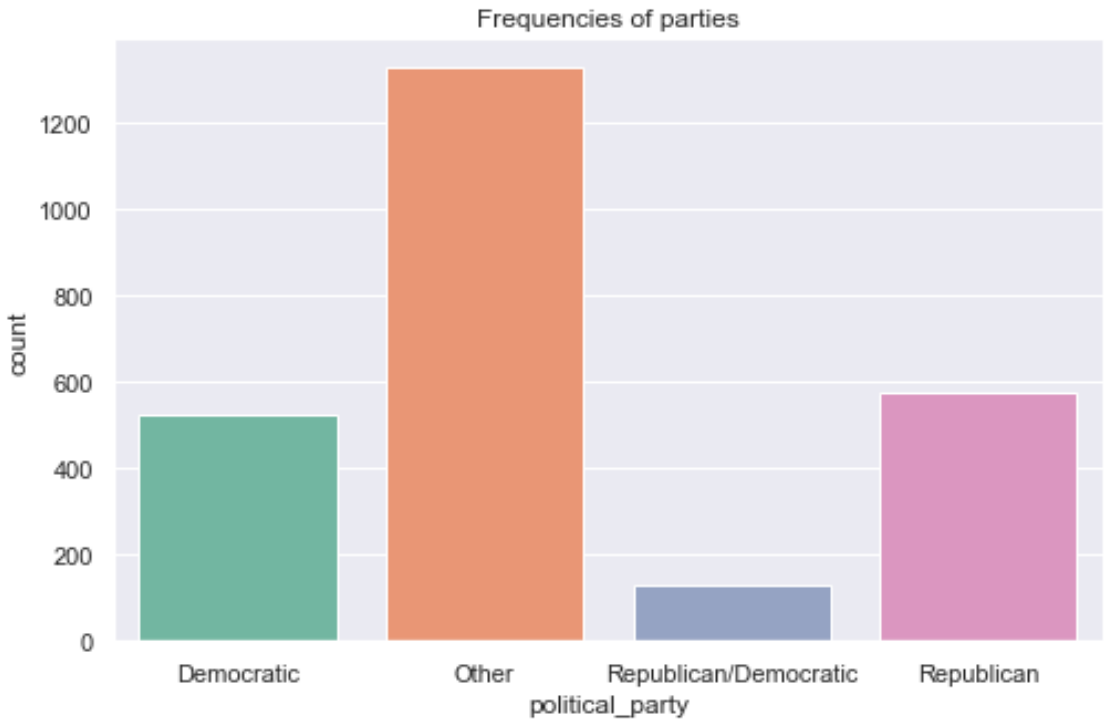
[36]:
```
election1.head(3)
```

[36]:
```
                                                text  sentiment  \
0  @robreiner so afraid of Nov, Dec, and Jan! Eve…          0
1  RT @SueCOOK: Lord Sumption launches Recovery -…          0
2  RT @WalidPhares: Uber Timing: after #Biden adv…          0

  negative_reason                                         clean_text  \
0         covid19  afraid nov dec jan even bidenharri win frump m…
1          others  lord sumption launch recoveri new initi promot…
2         covid19  uber time biden advisor say would put pressur …

  political_party
0      Democratic
1           Other
2      Democratic
```

2.1.3 Visualize the political affiliations of the tweets

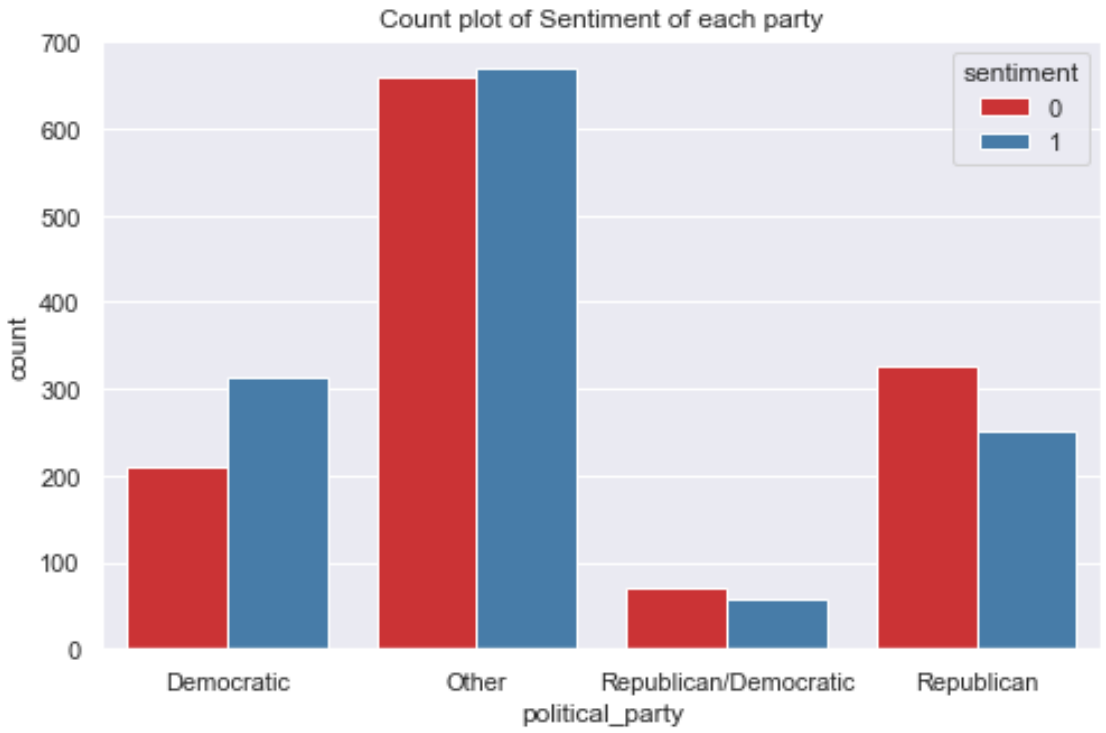
distributions of the political affiliations of the tweets The majority of tweets are classified as "other," and only a small number of people vote for both parties. One possible reason for many "other" groups is that people do not explicitly mention the selected keywords or send tweets without hashtags or mentions. Also, I did not include some neutral words such as "covid" or "vote" because it is hard to identify which party it belongs to based on those words. Also, the number of people who vote for republican and democratic are very close. Indeed, the published US election result indicates a close vote for those two parties. Furthermore, when labeling the parties, I did not consider the sentiment result, which might lead to biased results.

[37]:
```
parties = ['Republican', 'Democratic', 'Republican/Democratic', 'Other']
sns.set(rc={'figure.figsize':(8,5)})
sns.countplot(x='political_party', label=parties,␣
 ↪data=election1,palette="Set2").set_title('Frequencies of parties')
plt.show()
```



Count plot of Sentiment of each party According to each party's count plot of sentiment shown below, it is worth noticing that two classes of sentiment are imbalanced for democratic and republican parties. For the republican party, the number of negative tweets is more than positive tweets; in contrast, the positive tweets for democratic parties are more than negative ones, which is reasonable as Trump always posts offensive twitter speech.

[38]:
```
sns.countplot(x='political_party', hue="sentiment",
              data=election1,palette="Set1").set_title('Count plot of Sentiment␣
 ↪of each party')
plt.show()
```



0.2.2 2.2 Graphic figures for sentiment analysis and US election datasets

- 2.2.1 Sentiment analysis tweet
- 2.2.2 U.S. election tweet

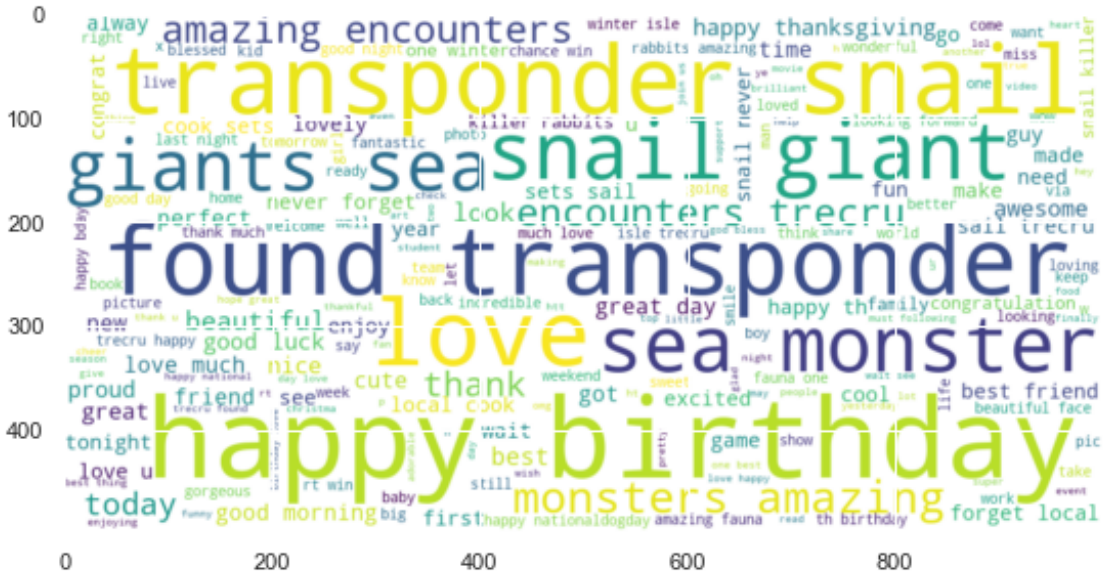
2.2.1 sentiment analysis

I grouped the clean text by two different sentiment labels in the sentiment analysis CSV file to generate a wordcloud. The wordcloud shown below shows that the neural words occur in both positive and negative tweets. The positive tweets contain positive words such as “love” and “happy”, while the negative ones have many negative words. Also, it is interesting to find “trump” in the word cloud of negative tweets. Those words would help us to decide the sentiment of the text.

[39]:
```
senti_label0=sentiment[sentiment["label"]==0]["clean_text"]
senti_label1=sentiment[sentiment["label"]==1]["clean_text"]
```

[40]:
```
sentiment_string=(" ").join([x for x in senti_label1])
wordcloud = WordCloud(width = 1000, height = 500,background_color='white').
 ↪generate(sentiment_string)
plt.figure(figsize=(20,10))
plt.imshow(wordcloud)
plt.axis("off")
plt.savefig("your_file_name"+".png", bbox_inches='tight')
ax1 = plt.subplot(121)
```

```
ax1.imshow(wordcloud,interpolation="bilinear")
plt.show()

sentiment_string=(" ").join([x for x in senti_label0])
wordcloud2 = WordCloud(width = 1000, height = 500,background_color='white').
 ↪generate(sentiment_string)
plt.figure(figsize=(20,10))
plt.imshow(wordcloud)
plt.axis("off")
plt.savefig("your_file_name"+".png", bbox_inches='tight')
ax2 = plt.subplot(122)
ax2.imshow(wordcloud2,interpolation="bilinear")
plt.show()
plt.close()
```



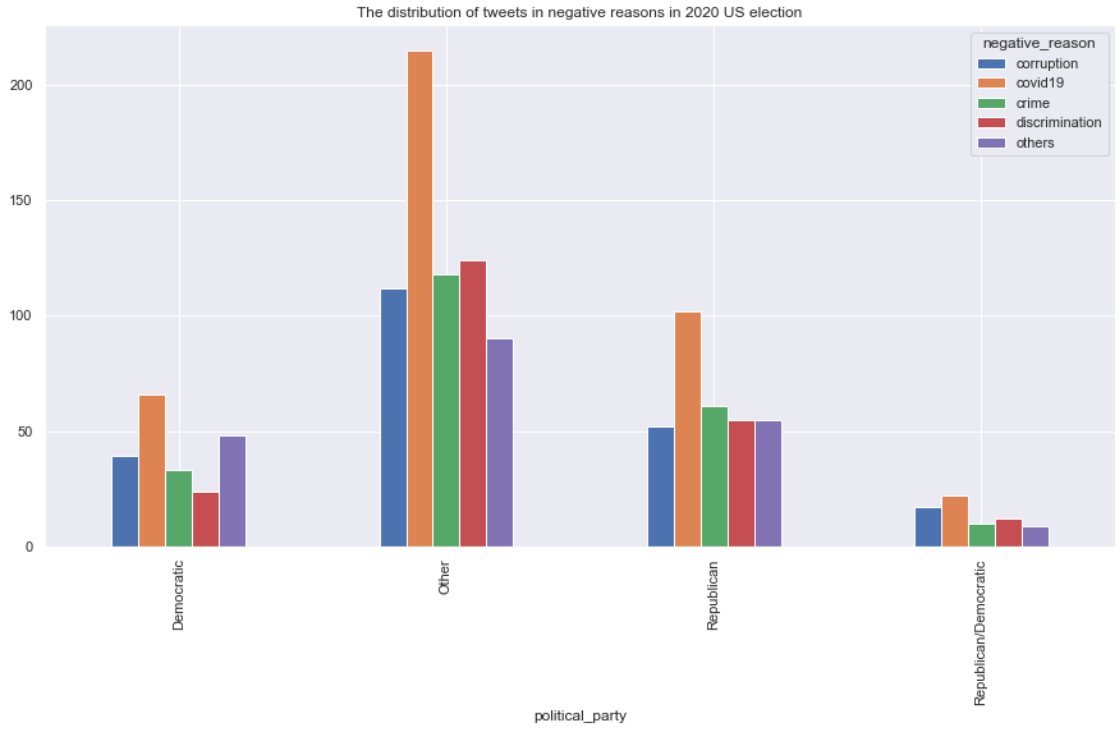
2.2.2 US election tweet

From the tweets' distribution plot in negative reason below, the covid-19 is the main reason for negative tweets. Besides, other than covid-19, people wrote negative tweets to attack the republican party due to crime and discrimination issues, while people wrote to attack the democratic party due to others and corruption reasons.

[41]:
```
neg_reason=election1[["sentiment","negative_reason","political_party"]]
neg_reason=neg_reason.dropna(axis=0)
neg_reason=neg_reason.drop(columns=["sentiment"])
new= pd.
 ↪pivot_table(neg_reason,index=["negative_reason"],columns="political_party",aggfunc=len)
```

[42]:
```
new.reset_index().set_index("negative_reason").T.plot(kind='bar',figsize=(15,7.
 ↪5),stacked=False,
                                                       title="The distribution of␣
 ↪tweets in negative reasons in 2020 US election")
```

[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7face53c5580>
```



0.3 3. Model Preparation

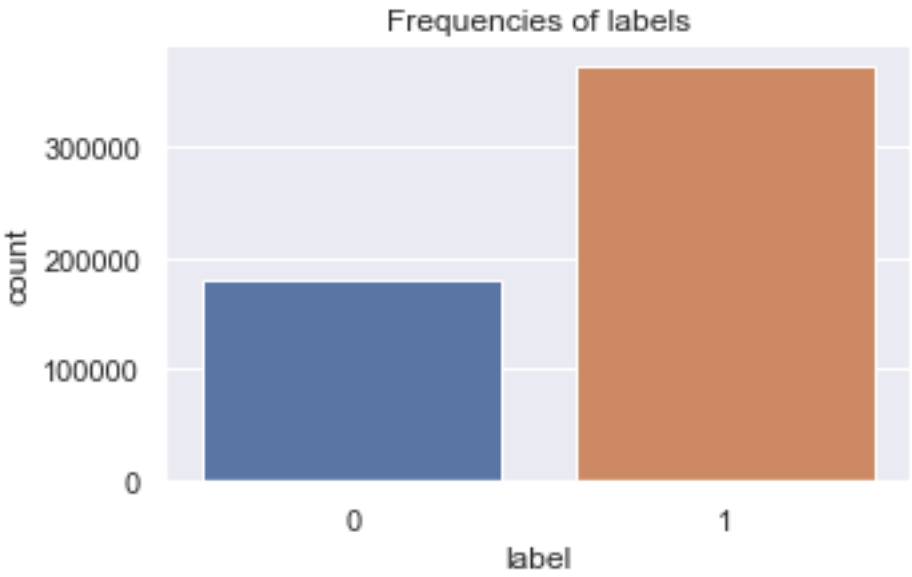
- Split the generic tweets randomly into training data (70%) and test data (30%).
- Prepare the data to try multiple classification algorithms (logistic regression, k-NN, Naive Bayes, SVM, decision trees, ensembles (RF, XGBoost)), where each tweet is considered a single observation/example. In these models, the target variable is the sentiment value, which is either positive or negative.
- Try **two different types of features, Bag of Words (word frequency) and TF-IDF.** (Hint: Be careful about when to split the dataset into training and testing set)

0.3.1 3.1 Sampling data

To avoid a long running time, I only select 20% of the dataset. Since the number of labels in the sentiment CSV file is imbalanced, I decide to undersampling the sentiment dataset. Also, since the label is already enconded as 0 and 1, there is no need to perform label encoding.

[43]:
```
#Randomly select 20% of dataset
sentiment1=sentiment.sample(frac = 0.2)
```

[44]:
```
plt.figure(figsize=(5,3))
sns.countplot(x='label', data=sentiment).set_title('Frequencies of labels')
plt.show()
```



[45]:
```
#undersampling
label1 = len(sentiment1[sentiment1['label'] == 0])
label1_index = sentiment1[sentiment1.label == 1].index
random_indices = np.random.choice(label1_index,label1, replace=False)
label0_indices = sentiment1[sentiment1.label == 0].index
under_sample_indices = np.concatenate([label0_indices,random_indices])
sentiment2 = sentiment1.loc[under_sample_indices]
```

[46]:
```
target_name= ["label"]
target=pd.DataFrame(sentiment1,columns=target_name)
feature=sentiment1["clean_text"]
```

0.3.2 3.2 Bag of words (Word frequency)

Word frequency is a word frequency counting technique in which a sorted list of words and their frequencies are generated, where the frequency is the frequency of occurrence in a given composition. In this project, the CountVectorizer function is used. It converts a collection of text documents to a matrix of token counts. After that, I split the sentiment tweets randomly into 70% and 30%.

[47]:
```
corpus=pd.read_csv("corpus.txt",sep="\t",header=None)
corpus_word=corpus[0].tolist()
corpus_score=corpus[1].tolist()
```

[48]:
```
# Apply CountVectorizer function
BoW_Vector = CountVectorizer(analyzer="word",vocabulary=corpus_word)
BoW_feature=BoW_Vector.transform(feature)
```

[49]:
```
# Split it into training and test dataset
Xsenti_train, Xsenti_test, ysenti_train,ysenti_test =␣
↪train_test_split(BoW_feature,
                                                                    target,
                                                                   ␣
↪test_size=0.3,
                                                                   ␣
↪stratify=target,
                                                                   ␣
↪random_state=75544)
```

[50]:
```
# Get the shape for training sets
print(Xsenti_train.shape)
print(ysenti_train.shape)
```

```
(77054, 2477)
(77054, 1)
```

0.3.3 3.3 TF-IDF

Tf-idf is a statistical method used to assess the importance of a word to one of the documents in a document set or corpus. The importance of a word increases proportionally with the frequency of its occurrence in the document, but decreases inversely with the frequency of its occurrence in the corpus. In this part, I use TFIDF to convert a collection of raw documents to a matrix of TF-IDF features.

[51]:
```
# Apply TdidfVectorizer function without normalization, and maximum 300 features
TFIDF_vector = TfidfVectorizer(norm=None,max_features=300)
TFIDF_feature=TFIDF_vector.fit_transform(feature)
```

[52]:
```
# Split the train and test based on the top 300 common words on TFIDF
TFIDF_x_train, TFIDF_x_test, TFIDF_y_train, TFIDF_y_test =␣
↪train_test_split(TFIDF_feature,
                                                        target,
                                                        test_size=0.
↪3,stratify=target)
```

[53]:
```
# Get the shapes of trainging sets
print(TFIDF_x_train.shape)
print(TFIDF_y_train.shape)
```

```
(77054, 300)
(77054, 1)
```

0.4 4. Model implementation

In this section, I divide the question into three parts: - 4.1 Choosing the model has the best performance by tuning the hyperparameters - Apply the baseline models for both bag of word

and TFIDF features in the following order: logistic regression, k-NN, Naive Bayes, SVM, decision trees, random forest, and XGBoost, and tune each model with 5 fold cross validation to select the model perform the best. - 4.2 Apply the optimal model from sentiment training set to evaluate the 2020 US election - Feature selection - Implement the best model - Visualize the results and discussion - 4.3 Train Multiclass Classification to predict the reasons for negative tweets - Select the obsrvations with negative reasons - Feature selection - Model implementation (Logistic regression, random forest, and XGBoost)

0.4.1 4.1 Choosing the model has the best performance by tuning the hyperparameters

The hyperparater I selected to tune are: * LR: penalty C, solver * KNN: n_neighbors * NB: alpha * SVM: C * DT: criterion * RF: n_jobs=-1,criterion * XGB: learning_rate

The overall results are shown in the following two tables:

	Train Accuracy	Test Accuracy	Mean cross validation score after hyperparameter tuning
Logistic Regression	92.194%	92.023%	91.941%
KNN	91.704%	90.961%	90.490%
Naive Bayes	89.437%	89.198%	89.250%
SVM	92.432%	91.972%	91.878%
Decision Trees	93.425%	91.151%	91.022%
Random Forest	93.425%	91.591%	91.463%
XGBoost	89.875%	90.107%	89.776%

Bag of word:

	Train Accuracy	Test Accuracy	Mean cross validation score after hyperparameter tuning
Logistic Regression	89.824%	89.359%	89.605%
KNN	87.797%	83.736%	84.643%
Naive Bayes	83.070%	82.882%	82.535%
SVM	89.724%	89.281%	89.527%
Decision Trees	94.512%	88.139%	88.332%
Random Forest	94.666%	89.114%	89.356%
XGBoost	89.500%	88.823%	89.065%

TF-IDF According to the tables shown above, the **logistic regression using the word frequency (Bag of words) features with hyperparameter{penalty:'l2', C: 10, solver:'liblinear'}** performs the best with the training accuracy of 92%, and I will use this model for further investigation on US election dataset. The relative codes for this part are shown as following:

4.1.1 Baseline models

[54]:
```
classifiers = [LogisticRegression(),
               KNeighborsClassifier(),
               MultinomialNB(),
               svm.LinearSVC(),
               DecisionTreeClassifier(),
               RandomForestClassifier(),
               XGBClassifier()]
```

Baseline models (Bag of Words)

[55]:
```
log_cols=["Classifier", "Train Accuracy", "Test Accuracy"]
log = pd.DataFrame(columns=log_cols)

for clf in classifiers:
    clf.fit(Xsenti_train, ysenti_train)
    name = clf.__class__.__name__
    print("-"*20)
    print(name)
    print('****Performance****')
    train_predictions = clf.predict(Xsenti_train)
    train_acc = accuracy_score(ysenti_train, train_predictions)
    print("Train Accuracy: {:.5%}".format(train_acc))
    test_predictions = clf.predict(Xsenti_test)
    test_acc = accuracy_score(ysenti_test, test_predictions)
    print("Test Accuracy: {:.5%}".format(test_acc))
    log_entry = pd.DataFrame([[name, train_acc*100, test_acc]],␣
 ↪columns=log_cols)
    log = log.append(log_entry)
print("-"*20)
```

```
--------------------
LogisticRegression
****Performance****
Train Accuracy: 92.19379%
Test Accuracy: 92.02398%
--------------------
KNeighborsClassifier
****Performance****
Train Accuracy: 91.70452%
Test Accuracy: 90.96112%
--------------------
MultinomialNB
****Performance****
Train Accuracy: 89.43728%
Test Accuracy: 89.19876%
--------------------
LinearSVC
```

```
****Performance****
Train Accuracy: 92.43258%
Test Accuracy: 91.97250%
--------------------
DecisionTreeClassifier
****Performance****
Train Accuracy: 93.42539%
Test Accuracy: 91.15189%
--------------------
RandomForestClassifier
****Performance****
Train Accuracy: 93.42539%
Test Accuracy: 91.59096%
--------------------
XGBClassifier
****Performance****
Train Accuracy: 89.87593%
Test Accuracy: 90.10719%
--------------------
```

Baseline models (TFIDF)

[56]:
```
for clf in classifiers:
    clf.fit(TFIDF_x_train, TFIDF_y_train)
    name = clf.__class__.__name__
    print("-"*20)
    print(name)
    print('****Performance****')
    train_predictions = clf.predict(TFIDF_x_train)
    train_acc = accuracy_score(TFIDF_y_train, train_predictions)
    print("Train Accuracy: {:.3%}".format(train_acc))
    test_predictions = clf.predict(TFIDF_x_test)
    test_acc = accuracy_score(TFIDF_y_test, test_predictions)
    print("Test Accuracy: {:.3%}".format(test_acc))
    log_entry = pd.DataFrame([[name, train_acc*100, test_acc]],␣
 ↪columns=log_cols)
    log = log.append(log_entry)
print("-"*20)
```

```
--------------------
LogisticRegression
****Performance****
Train Accuracy: 89.824%
Test Accuracy: 89.359%
--------------------
KNeighborsClassifier
****Performance****
Train Accuracy: 87.797%
```

```
Test Accuracy: 83.736%
--------------------
MultinomialNB
****Performance****
Train Accuracy: 83.070%
Test Accuracy: 82.882%
--------------------
LinearSVC
****Performance****
Train Accuracy: 89.724%
Test Accuracy: 89.281%
--------------------
DecisionTreeClassifier
****Performance****
Train Accuracy: 94.512%
Test Accuracy: 88.139%
--------------------
RandomForestClassifier
****Performance****
Train Accuracy: 94.509%
Test Accuracy: 89.114%
--------------------
XGBClassifier
****Performance****
Train Accuracy: 89.500%
Test Accuracy: 88.823%
--------------------
```

4.1.2 Apply seven models with hyperparameter tuning and 5 fold cross validation

Logisitic regression Tuning with 5 fold cv

[57]:
```
def LRTuning(xtrain, ytrain):
    #defining classifier
    logReg = LogisticRegression()
    #hyperparameters to be tuned
    penalty = ['l1', 'l2']
    C =[0.01, 0.1, 1, 10, 100]
    solver=['liblinear','lbfgs', 'newton-cg']
    hyperparameters = dict(penalty=penalty, C=C,solver=solver)

    #grid search with 5-fold cv
    clf = GridSearchCV(logReg, hyperparameters, cv=5, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best penalty for Logistic Regression:', best_model.best_estimator_.
 ↪get_params()['penalty'])
```

```
    print('Best C for Logistic Regression:', best_model.best_estimator_.
↪get_params()['C'])
    print('Best score:', clf.best_score_)

    log_penalty = best_model.best_estimator_.get_params()['penalty']
    log_C = best_model.best_estimator_.get_params()['C']
    solver= best_model.best_estimator_.get_params()['solver']
    best=clf.best_score_

    return log_penalty, log_C,solver,best
```

[58]:
```
LRTuning(Xsenti_train, ysenti_train)
```

```
Best penalty for Logistic Regression: l2
Best C for Logistic Regression: 10
Best score: 0.9194072026306163
```

[58]:
```
('l2', 10, 'liblinear', 0.9194072026306163)
```

[59]:
```
LRTuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best penalty for Logistic Regression: l1
Best C for Logistic Regression: 0.1
Best score: 0.8960469263739617
```

[59]:
```
('l1', 0.1, 'liblinear', 0.8960469263739617)
```

KNN Tuning with 5 fold cv

[60]:
```
def KNNtuning(xtrain,ytrain):
    #Define the classifier:
    knn = KNeighborsClassifier()
    #hyperparameter to be tuned
    hyperparameters = dict(n_neighbors=[5,10,25,100,500,5000])

    #gridsearch with 5-fold cv
    clf = GridSearchCV(knn, hyperparameters, cv=5)
    #Fit the model
    best_model = clf.fit(xtrain,ytrain)
    #Print The value of best Hyperparameters

    print('Best n_neighbors:', best_model.best_estimator_.
↪get_params()['n_neighbors'])
    print('Best score:', clf.best_score_)

    neighbors=best_model.best_estimator_.get_params()['n_neighbors']
    best=clf.best_score_
```

```
    return neighbors,best
```

[61]:
```
KNNtuning(Xsenti_train, ysenti_train)
```

```
Best n_neighbors: 25
Best score: 0.9048978583818303
```

[61]:
```
(25, 0.9048978583818303)
```

[62]:
```
KNNtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best n_neighbors: 500
Best score: 0.8464323186060371
```

[62]:
```
(500, 0.8464323186060371)
```

Naive Bayes tuning with 5 fold cv

[63]:
```
def NBtuning(xtrain,ytrain):
    #Define the classifier:
    NB = MultinomialNB()
    #hyperparameter to be tuned
    hyperparameters = {'alpha':[0.01,0.1,1,5,20,100]}

    #gridsearch with 5-fold cv
    clf = GridSearchCV(NB, hyperparameters, cv=5)
    #Fit the model
    best_model = clf.fit(xtrain,ytrain)
    #Print The value of best Hyperparameters

    print('Best alpha:', best_model.best_estimator_.get_params()['alpha'])
    print('Best score:', clf.best_score_)

    alpha=best_model.best_estimator_.get_params()['alpha']
    best=clf.best_score_

    return alpha,best
```

[64]:
```
NBtuning(Xsenti_train, ysenti_train)
```

```
Best alpha: 0.1
Best score: 0.8925040083835716
```

[64]:
```
(0.1, 0.8925040083835716)
```

[65]:
```
NBtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best alpha: 0.1
Best score: 0.8295350409803189
```

[65]:
```
(0.1, 0.8295350409803189)
```

SVM tuning with 5 fold cv

[66]:
```
def SVMtuning(xtrain,ytrain):
    #Define the classifier:
    SVM = svm.LinearSVC()
    #hyperparameter to be tuned
    hyperparameters = {'C': [0.1,1, 10, 100]}

    #gridsearch with 5-fold cv
    clf = GridSearchCV(SVM, hyperparameters, cv=5)
    #Fit the model
    best_model = clf.fit(xtrain,ytrain)
    #Print The value of best Hyperparameters

    print('Best C:', best_model.best_estimator_.get_params()['C'])
    print('Best score:', clf.best_score_)

    c=best_model.best_estimator_.get_params()['C']
    best=clf.best_score_

    return c,best
```

[67]:
```
SVMtuning(Xsenti_train, ysenti_train)
```

```
Best C: 1
Best score: 0.9187842532729956
```

[67]:
```
(1, 0.9187842532729956)
```

[68]:
```
SVMtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best C: 1
Best score: 0.8952682365188218
```

[68]:
```
(1, 0.8952682365188218)
```

Decision tree tuning with 5-fold cv

[69]:
```
def DTtuning(xtrain, ytrain):

    # Defining classifier
    tree = DecisionTreeClassifier()
```

```
    # Measurement criteria
    criterion = ['gini', 'entropy']

    hyperparameters = dict(criterion=criterion)

    # Grid search with 5-fold cv
    clf = GridSearchCV(tree, hyperparameters, cv=5, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best criterion for Decision Tree:', best_model.best_estimator_.
↪get_params()['criterion'])
    print('Best score:', clf.best_score_)

    tree_criterion = best_model.best_estimator_.get_params()['criterion']
    best=clf.best_score_

    return tree_criterion,best
```

[70]:
```
DTtuning(Xsenti_train, ysenti_train)
```

```
Best criterion for Decision Tree: entropy
Best score: 0.9102188206667486
```

[70]:
```
('entropy', 0.9102188206667486)
```

[71]:
```
DTtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best criterion for Decision Tree: entropy
Best score: 0.8833155851536809
```

[71]:
```
('entropy', 0.8833155851536809)
```

Random Forest tuning with 5-fold cv

[72]:
```
def RFtuning(xtrain, ytrain):
    #defining classifier
    randforest = RandomForestClassifier(n_jobs=-1)
    # Number of trees in random forest
    #n_estimators = [int(x) for x in np.linspace(start = 100, stop = 500, num =␣
↪3)]
    # Measurement criteria
    criterion = ['gini', 'entropy']

    hyperparameters = dict(criterion=criterion)

    # cross validation is not used to save time
    clf = GridSearchCV(randforest, hyperparameters, verbose=0)
```

```
    best_model = clf.fit(xtrain, ytrain)

    #print('Best n_estimators for Random Forest:', best_model.best_estimator_.
↪get_params()['n_estimators'])
    print('Best criterion for Random Forest:', best_model.best_estimator_.
↪get_params()['criterion'])
    print('Best score:', clf.best_score_)

    #randforest_n_estimators = best_model.best_estimator_.
↪get_params()['n_estimators']
    randforest_criterion = best_model.best_estimator_.get_params()['criterion']
    best=clf.best_score_

    return randforest_criterion ,best
```

[73]:
```
RFtuning(Xsenti_train, ysenti_train)
```

```
Best criterion for Random Forest: entropy
Best score: 0.9146313080853486
```

[73]:
```
('entropy', 0.9146313080853486)
```

[74]:
```
RFtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best criterion for Random Forest: entropy
Best score: 0.8935681403731991
```

[74]:
```
('entropy', 0.8935681403731991)
```

XGBoost tuning with 5-fold cv

[75]:
```
def XGBtuning(xtrain, ytrain):
    #defining classifier
    xgb=XGBClassifier()
    learning_rate= [0.05, 0.10, 0.15, 0.20, 0.25, 0.30]
    hyperparameters = dict(learning_rate=learning_rate)

    # cross validation is not used to save time
    clf = GridSearchCV(xgb, hyperparameters, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best learning_rate:', best_model.best_estimator_.
↪get_params()['learning_rate'])
    print('Best score:', clf.best_score_)

    learning_rate = best_model.best_estimator_.get_params()['learning_rate']
    best=clf.best_score_
```

```
    return learning_rate,best
```

[76]:
```
XGBtuning(Xsenti_train, ysenti_train)
```

```
Best learning_rate: 0.3
Best score: 0.8977600246855035
```

[76]:
```
(0.3, 0.8977600246855035)
```

[77]:
```
XGBtuning(TFIDF_x_train,TFIDF_y_train)
```

```
Best learning_rate: 0.3
Best score: 0.8906480774180909
```

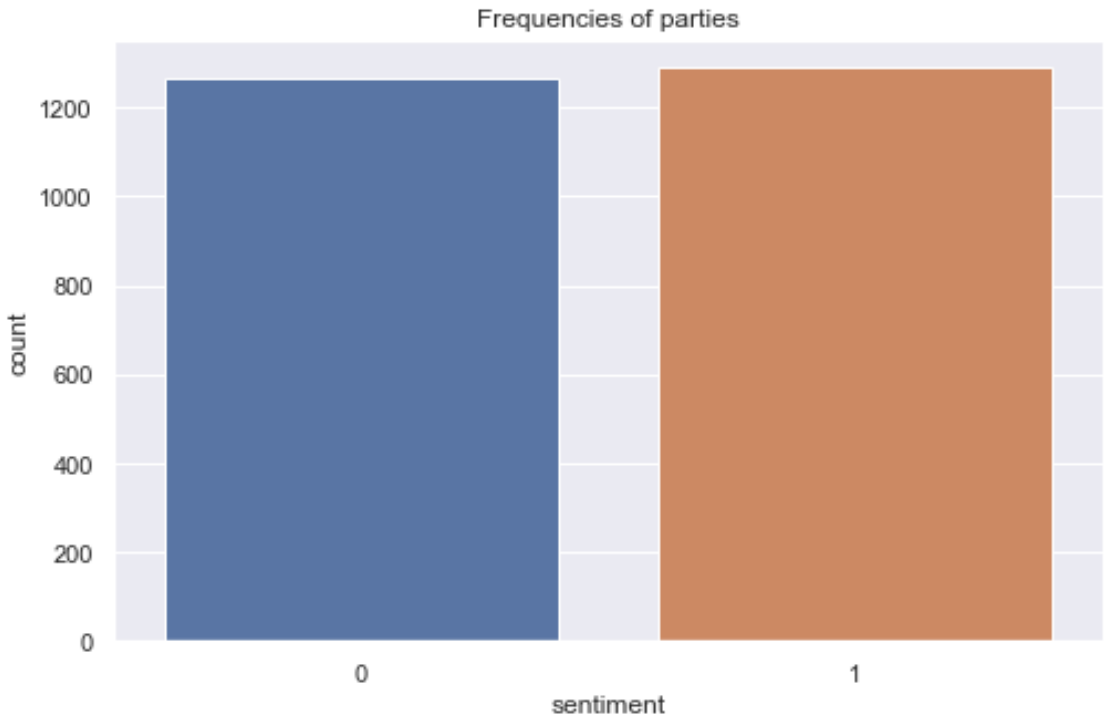
[77]:
```
(0.3, 0.8906480774180909)
```

0.4.2 4.2 Apply the optimal model from sentiment training set to evaluate the 2020 US election

Section 4.2 are analyzed with the following three steps: * 4.2.1 Feature selection * 4.2.2 Implement the best model * 4.2.3 Visualize the results and discussion

4.2.1 Feature Selection The dataset of US election is much smaller than the sentiment analysis csv file, and so I decide not to sample the dataset. Also, since the class of target variable "sentiment" is balanced, there is no need to resample the model.

[78]:
```
election2=election.copy()
sns.countplot(x='sentiment', data=election2).set_title('Frequencies of parties')
plt.show()
#There is no need to resample the model
```



[79]:
```
elect_y_name= ["sentiment"]
elect_y=pd.DataFrame(election2,columns=elect_y_name)
elect_x=election2["clean_text"]
```

[80]:
```
#Apply BoW methos
BoW_Vector = CountVectorizer(analyzer="word",vocabulary=corpus_word)
#Train model on election data
BoW_elect_feature=BoW_Vector.transform(elect_x)
```

[81]:
```
#splitting tweet and sentiment values into 70% train and 30% test
elect_x_train, elect_x_test, elect_y_train, elect_y_test =␣
↪train_test_split(BoW_elect_feature, elect_y,
                                                                    ␣
↪test_size=0.3, random_state=75544)
```

4.2.2 Implement the best model

[82]:
```
bow_lr = LogisticRegression(penalty = 'l2',
                            C=10,
                            solver='liblinear')
bow_lr.fit(elect_x_train, elect_y_train)
prediction_elect = bow_lr.predict(elect_x_test)
```

[83]:
```
print('The accuracy of the Logistic regression model for the 2020 US Election␣
↪is:',
      metrics.accuracy_score(prediction_elect, elect_y_test))
```

```
The accuracy of the Logistic regression model for the 2020 US Election is:
0.7389033942558747
```

[84]:
```
print(confusion_matrix(elect_y_test, prediction_elect))
print(classification_report(elect_y_test, prediction_elect))
```

```
[[238 145]
 [ 55 328]]
              precision    recall  f1-score   support

           0       0.81      0.62      0.70       383
           1       0.69      0.86      0.77       383

    accuracy                           0.74       766
   macro avg       0.75      0.74      0.74       766
weighted avg       0.75      0.74      0.74       766
```

Append the prediction result to the election dataset

[85]:
```
df_election_test = election1.loc[elect_y_test.index]
#append prediction result to the dataset with classified political party
df_election_test['prediction result'] = prediction_elect
```

[86]:
```
elect_rep = df_election_test[df_election_test['political_party']=='Republican']
elect_demo = df_election_test[df_election_test['political_party']=='Democratic']
acc_rep = accuracy_score(elect_rep['sentiment'],elect_rep['prediction result'])
acc_demo = accuracy_score(elect_demo['sentiment'],elect_demo['prediction␣
↪result'])
print("Accuracy for republican: {:.3%}".format(acc_rep))
print("Accuracy for democratic: {:.3%}".format(acc_demo))
```

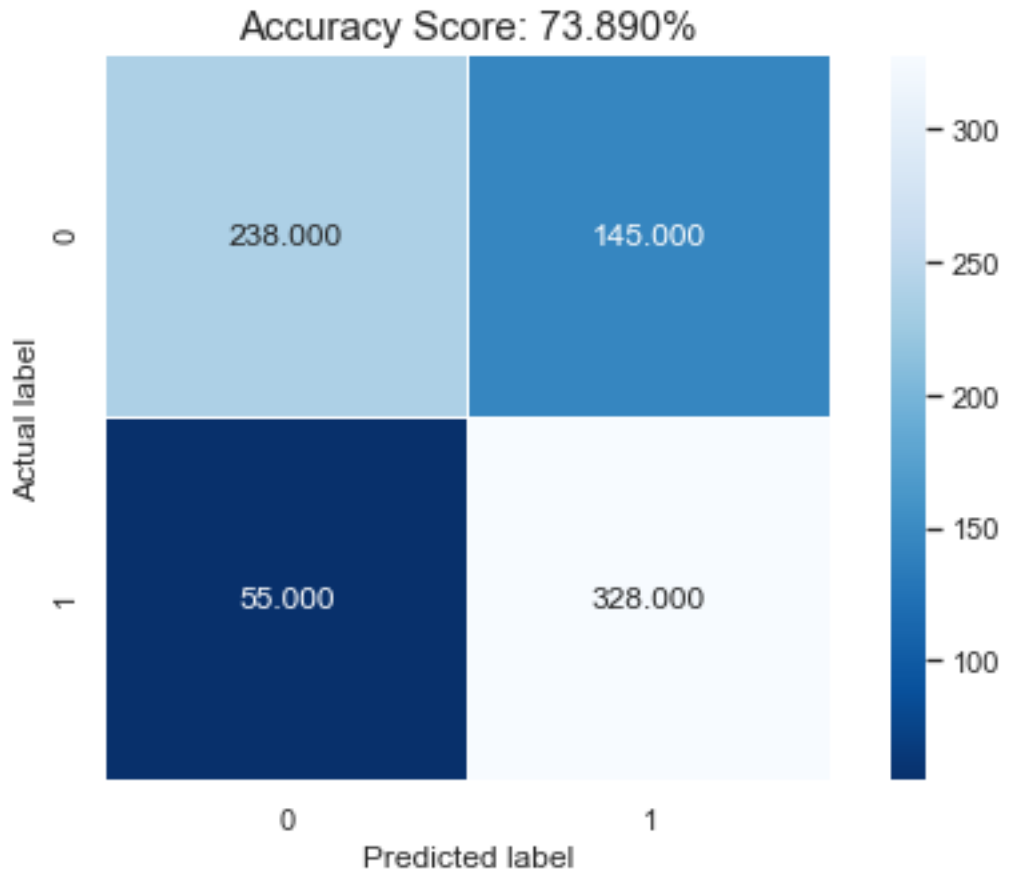
```
Accuracy for republican: 78.443%
Accuracy for democratic: 77.273%
```

I implement the logisitc regression model with tuned hyperparameter, and have obtain 73.890% overall accuracy, f1-score of 71% for negative tweets, and fi-score of 77% for positive tweets. The accuracy for republican and democratic party are 78.443 and 77.273% respectively.

Party	Accuracy
Republican	78.443%
Democratic	77.273%
Overall	73.890%

4.2.3 Visualize the Results visualization of confusion matrix The visualization of confusion matrix is shown as following:

[87]:
```
cm = metrics.confusion_matrix(elect_y_test, prediction_elect)
score = bow_lr.score(elect_x_test, elect_y_test)
sns.heatmap(cm, annot=True, fmt=".3f", linewidths=.5, square = True, cmap =␣
↪'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
sample_title = 'Accuracy Score: {:.3%}'.format(score)
plt.title(sample_title, size = 15);
```



After that, I plot the distributions of sentiment Predictions and true sentiments for both political parties separately.

Visualization of sentiment prediction for different parties

[88]:
```
#include sentiment, negative reason, political party and prediction result into␣
↪the dataset
political_result = df_election_test[["sentiment","negative_reason",
```

```
                                        "political_party","prediction result"]]
#count the number of tweets based on prediction result and group by political␣
 ↪party
political_result=political_result.groupby(['political_party', 'prediction␣
 ↪result']).count()
# reset the index of the dataframe
political_result=political_result.reset_index()
political_result=political_result[(political_result['political_party'] ==␣
 ↪"Democratic")|
                                  (political_result['political_party'] ==␣
 ↪"Republican")]
```

[89]:
```
pio.renderers.default='notebook'
fig = go.Figure()
fig.add_trace(go.Bar(
    x=political_result[political_result["prediction␣
 ↪result"]==0]["political_party"],
    y=political_result[political_result["prediction result"]==0]["sentiment"],
    name='negative',
    marker_color='firebrick'
))
fig.add_trace(go.Bar(
    x=political_result[political_result["prediction␣
 ↪result"]==1]["political_party"],
    y=political_result[political_result["prediction result"]==1]["sentiment"],
    name='positive',
    marker_color='steelblue'
))
fig.update_layout(barmode='group',
                  xaxis_title="Political party",
                  yaxis_title="Number of Tweets",
                  legend_title="Prediction result",
                  title="Sentiment Predictions for both political parties")
fig.show()
```

Visualization of true sentiment for two parties

[90]:
```
true_result = df_election_test[["sentiment","negative_reason",
                                        "political_party","prediction result"]]
#count the number of tweets based on true sentiment result and group by␣
 ↪political party
true_result=true_result.groupby(['political_party', 'sentiment']).count()
# reset the index of the dataframe
true_result=true_result.reset_index()
true_result=true_result[(true_result['political_party'] == "Democratic")|
                        (true_result['political_party'] == "Republican")]
```

[91]:
```
pio.renderers.default='notebook'
fig = go.Figure()
fig.add_trace(go.Bar(
    x=true_result[true_result["sentiment"]==0]["political_party"],
    y=true_result[true_result["sentiment"]==0]["prediction result"],
    name='negative',
    marker_color='firebrick'
))
fig.add_trace(go.Bar(
    x=true_result[true_result["sentiment"]==1]["political_party"],
    y=true_result[true_result["sentiment"]==1]["prediction result"],
    name='positive',
    marker_color='steelblue'
))
fig.update_layout(barmode='group',
                  xaxis_title="Political party",
                  yaxis_title="Number of Tweets",
                  legend_title="True result",
                  title="True sentiment for both political parties")
fig.show()
```

Discussion: According to the distributions plots of sentiment prediction and true sentiment for both parties, there are several exciting findings: 1. The overall trend and number of tweets involving democratic and republican parties are similar. 2. The democratic party has a higher positive sentiment result, while the Republican has a higher negative sentiment result. 3. The democratic party has more positive sentiments than negative sentiments under sentiment prediction and true emotion. However, the republican party shows a reversal trend of forecast compared to the true result. The negative tweets are much more than the positive ones, but the prediction shows the opposite result. 4. Even though both parties' overall accuracy scores are similar, the model predicts the result for the democratic party more accurately. In a nutshell, the logistic regression model captures the overall trend of the actual data for the democratic party but not for the republican party, which implies that the model could not predict the negative sentiment correctly. In reality, the democratic party wins the 2020 US election. The sentiment prediction result could represent the election result, and the NLP analysis can be used to observe the election trend. However, people still should not rely too much on the model to predict the results.

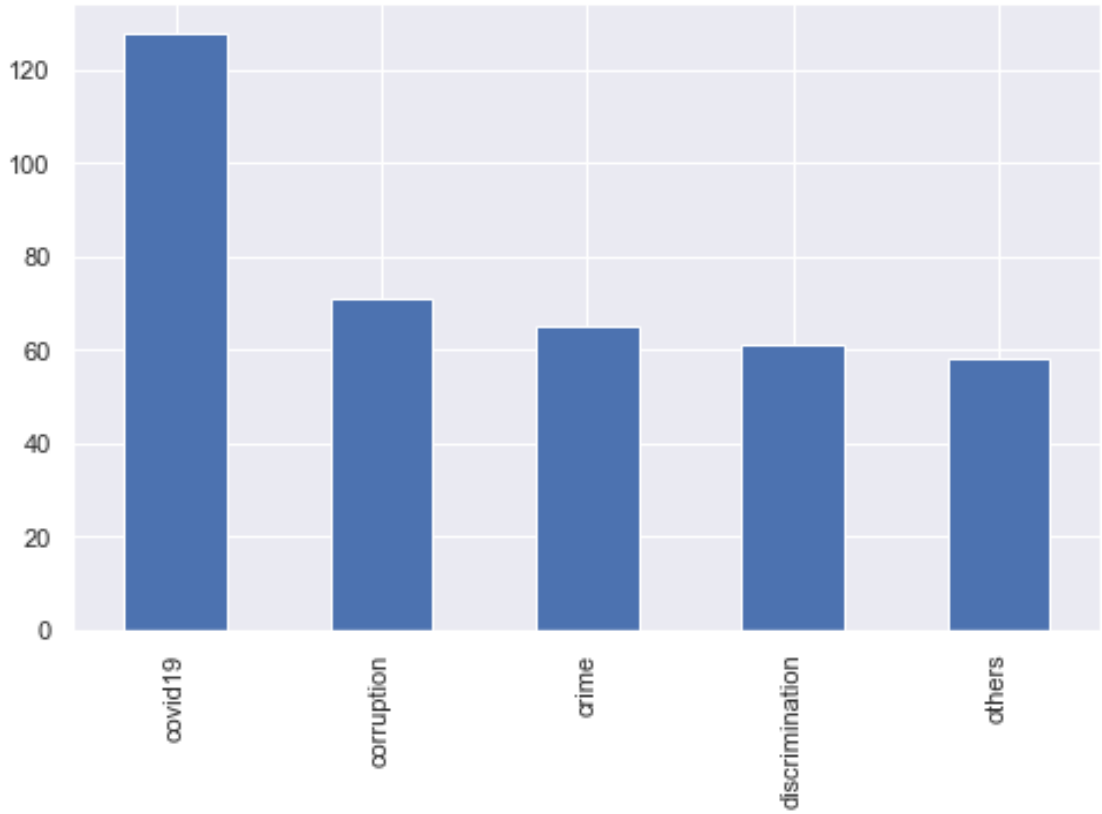
0.4.3 4.3 Train Multiclass Classification to predict the reasons for negative tweets

Section 4.3 are analyzed with the following three steps: * 4.3.1 Select the obsrvations with negative reasons * 4.3.2 Feature selection * 4.3.3 Model implementation

4.3.1 Select the observations with negative reasons

[92]:
```
neg_elect=df_election_test[df_election_test['sentiment'] == 0]
#There are five negative reasons
print(neg_elect["negative_reason"].value_counts().plot(kind="bar"))
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
```



Since all of those five reasons are different, I decide not to group those reasons. However, label encoding is required as the target values are a list of string.

Label encoding

[93]:
```
#Using label encoding
le = preprocessing.LabelEncoder()
neg_elect['negative_reason_target'] = le.
 ↪fit_transform(neg_elect['negative_reason'].tolist())
#corruption:0, covid19:1, crime:2,discrimination:3,other:4
```

[94]:
```
neg_elect.head()
```

[94]:
```
                                                  text  sentiment  \
484   POV: Im just your average Joe pulling up on so…          0
2055  RT @civilrightsorg: Delayed election results a…          0
1157  RT @Msdesignerlady: How cool is this America, …          0
2159  While terrorist @BorisJohnson wants to finish …          0
2002  Trump lied to Americans about covid. He failed…          0

     negative_reason                                    clean_text  \
```

```
484       corruption   pov im averag joe pull racist trumper think want
2055         covid19  delay elect result noth sign system work everi…
1157           crime  cool america detroit steviewond perform event …
2159         covid19  terrorist want finish uk economi terrorist gre…
2002  discrimination  trump lie american covid fail institut nation …

     political_party  prediction result  negative_reason_target
484            Other                  0                       0
2055           Other                  0                       1
1157           Other                  0                       2
2159           Other                  1                       1
2002      Republican                  0                       3
```

4.3.2 Feature selection (Bag of Words)

[95]:
```
neg_elect_y_name= ["negative_reason_target"]
neg_elect_y=pd.DataFrame(neg_elect,columns=neg_elect_y_name)
neg_elect_x=neg_elect["clean_text"]
```

[96]:
```
#Apply BoW and train model on negative election data
BoW_elect_feature=BoW_Vector.transform(neg_elect_x)

#splitting negative tweet and negative reasons into 70% train and 30% test
neg_elect_x_train, neg_elect_x_test, neg_elect_y_train, neg_elect_y_test =␣
↪train_test_split(BoW_elect_feature,
                                                                        ␣
↪            neg_elect_y,
                                                                       ␣
↪test_size=0.3, random_state=75544)
```

4.3.3 Model implementation

1. Logistic regression with one vs rest Algorithms such as the Logistic Regression and Support Vector Machines were designed for binary classification and do not natively support classification tasks with more than two classes. And so one approach for using One-vs-Rest classification algorithms for multi-classification problems is to split the dataset into multiple binary classification datasets and fit a binary classification model on each.

[97]:
```
def negLRTuning(xtrain, ytrain):
    #defining classifier
    neg_lr = LogisticRegression()
    #hyperparameters to be tuned
    penalty = ['l1', 'l2']
    C =[0.01, 0.1, 1, 10, 100]
    solver=['liblinear','lbfgs', 'newton-cg']
    multi_class=['ovr']
```

```
    hyperparameters = dict(penalty=penalty, C=C,solver=solver,␣
↪multi_class=multi_class)
    #grid search with 5-fold cv
    clf = GridSearchCV(neg_lr, hyperparameters, cv=5, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best penalty:', best_model.best_estimator_.get_params()['penalty'])
    print('Best C for:', best_model.best_estimator_.get_params()['C'])
    print('Best solver:', best_model.best_estimator_.get_params()['solver'])
    print('Best score:', clf.best_score_)

    return None
```

[98]:
```
negLRTuning(neg_elect_x_train,neg_elect_y_train)
```

```
Best penalty: l1
Best C for: 1
Best solver: liblinear
Best score: 0.3323549965059399
```

2. Random Forest

[99]:
```
def negRFtuning(xtrain, ytrain):
    #defining classifier
    randforest = RandomForestClassifier(n_jobs=-1)
    # Number of trees in random forest
    n_estimators = [int(x) for x in np.linspace(start = 100, stop = 500, num =␣
↪3)]
    # Measurement criteria
    criterion = ['gini', 'entropy']
    hyperparameters = dict(criterion=criterion,n_estimators=n_estimators)

    # cross validation is not used to save time
    clf = GridSearchCV(randforest, hyperparameters, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best criterion:', best_model.best_estimator_.
↪get_params()['criterion'])
    print('Best n_estimators:', best_model.best_estimator_.
↪get_params()['n_estimators'])
    print('Best score:', clf.best_score_)

    return None
```

[100]:
```
negRFtuning(neg_elect_x_train,neg_elect_y_train)
```

```
Best criterion: entropy
Best n_estimators: 500
```

```
Best score: 0.24276729559748428
```

3. XGBoost

[101]:
```
def negXGBtuning(xtrain, ytrain):
    #defining classifier
    xgb=XGBClassifier()
    max_depth=[range(3,10,2)]
    min_child_weight=[range(1,6,2)]
    learning_rate= [0.05, 0.10, 0.15, 0.20, 0.25, 0.30]
    hyperparameters = dict(learning_rate=learning_rate)

    # cross validation is not used to save time
    clf = GridSearchCV(xgb, hyperparameters, verbose=0)
    best_model = clf.fit(xtrain, ytrain)

    print('Best max_depth:', best_model.best_estimator_.
 ↪get_params()['max_depth'])
    print('Best min_child_weight:', best_model.best_estimator_.
 ↪get_params()['min_child_weight'])
    print('Best learning_rate:', best_model.best_estimator_.
 ↪get_params()['learning_rate'])
    print('Best score:', clf.best_score_)

    return None
```

[102]:
```
negXGBtuning(neg_elect_x_train,neg_elect_y_train)
```

```
Best max_depth: 6
Best min_child_weight: 1
Best learning_rate: 0.05
Best score: 0.2951083158630328
```

[103]:
```
neglr = LogisticRegression(penalty="l1", C=1,solver="liblinear",␣
 ↪multi_class="ovr")
neglr.fit(neg_elect_x_train,neg_elect_y_train)
prediction_neglr = neglr.predict(neg_elect_x_test)
```

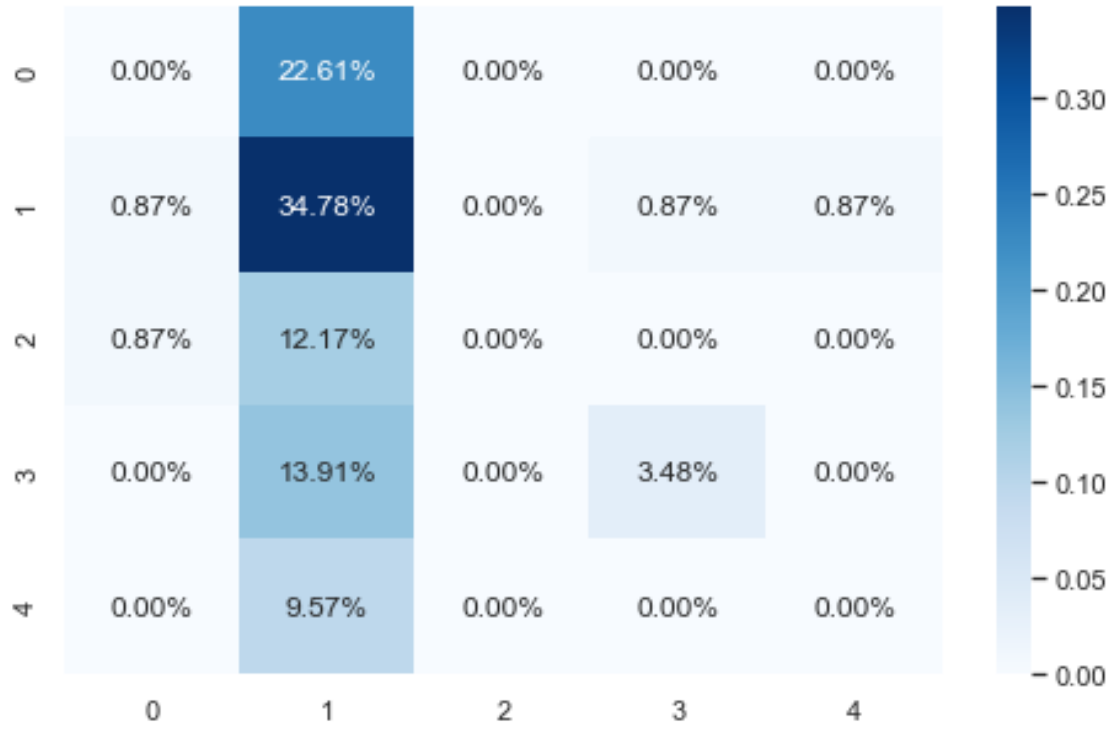
[104]:
```
print(classification_report(neg_elect_y_test, prediction_neglr))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        26
           1       0.37      0.93      0.53        43
           2       0.00      0.00      0.00        15
           3       0.80      0.20      0.32        20
           4       0.00      0.00      0.00        11
```

```
              accuracy                           0.38       115
             macro avg       0.23      0.23      0.17       115
          weighted avg       0.28      0.38      0.26       115
```

[105]:
```
cm2 = confusion_matrix(neg_elect_y_test, prediction_neglr)
fig = sns.heatmap(cm2/np.sum(cm2), annot=True, fmt='.2%', cmap='Blues')
xlabels = [0, 1,2, 3, 4]
ylabels = [0, 1,2, 3, 4]
fig.set_xticklabels(xlabels)
fig.set_yticklabels(ylabels)
```

[105]:
```
[Text(0, 0.5, '0'),
 Text(0, 1.5, '1'),
 Text(0, 2.5, '2'),
 Text(0, 3.5, '3'),
 Text(0, 4.5, '4')]
```



	Accuracy
Logistic regression	38%
Random Forest	25%
XGBoost	30%

Same, the logistic regression performs the best with the highest accuracy of 38%. According to the visualization of a confusion matrix for the multiclass logistic regression model, class 1 (covid-19) has the highest correct prediction rate, while class 0(corruption), 2(crime), 4(other) perform the worst. One possible solution could be using **Word2Vec**, a group of related models used to produce word embeddings, to perform feature selection. Word2vec takes as its input a large corpus of text. With Word2Vec, we take the relationship between word weight and actual context, which contains the word, into consideration. It will improve model accuracy as the context is crucial for Twitter data.

0.5 5. Results

Sentiment analysis

- Model performs the best: logistic regression by using the word frequency (Bag of words) features
- Hyperparameter tuning verifies that {penalty:'l2', C: 10, solver:'liblinear'} yields the best result
- Model accuracy on sentiment label: 91.961%

US election dataset

- Accuracy of logisitic regression on sentiment:73.890%
  - Hyperparameter tuning verifies that {penalty:'l', C: 1, solver:'liblinear'} yields the best result
  - Accuracy for republican: 78.443%
  - Accuracy for democratic: 77.273%
- Accuracy of logisitic regression on negative reasons: 38%

5.1 Sentiment prediction on US election data

- According to the visualization of true sentiment for each party, people tweet the democratic party with more positive emotion while tweeting the republican party with more negative emotion. The distribution plots for negative reasons indicate that the main reason for negative tweets is covid-19. For both parties, the number of reasons "covid-19" is almost twice more than each of the other negative reasons. Even so, the number of positive labels is more than the number of negative labels.

- The first model predicts the emotional trends among all labeled parties. Since the logistic regression model with a bag of words performs the best on the given sentiment analysis dataset, I apply it to the US election dataset and yield 73.89% accuracy. After visualizing the result for the Republican and the Democratic parties, it is worth noticing that the model correctly predicts the election result that the democratic party has more positive tweets, while it fails to predict that the republican party has more negative tweets. Through this model, one could conclude that people could use the NLP analysis to observe the election trends but should not rely too much on predicting the results.

- Compared to the prediction result on the sentiment analysis dataset, the accuracy reduces from 91.961% to 73.89%. There are two possible reasons: First, the US election dataset is much smaller than the sentiment analysis dataset. Second, most tweets are classified into

"other" and "republican/democratic", and those results are not connected with the sentiment. Third, due to limited computational time, the tuning of each model is restricted.

- Possible improved methods are:
  - Reducing the number of features to avoid overfitting
  - Increasing the size of the dataset
  - Classifying the features by connecting the sentiment label.
  - Exploring more hyperparameters that can be tuned for those seven models

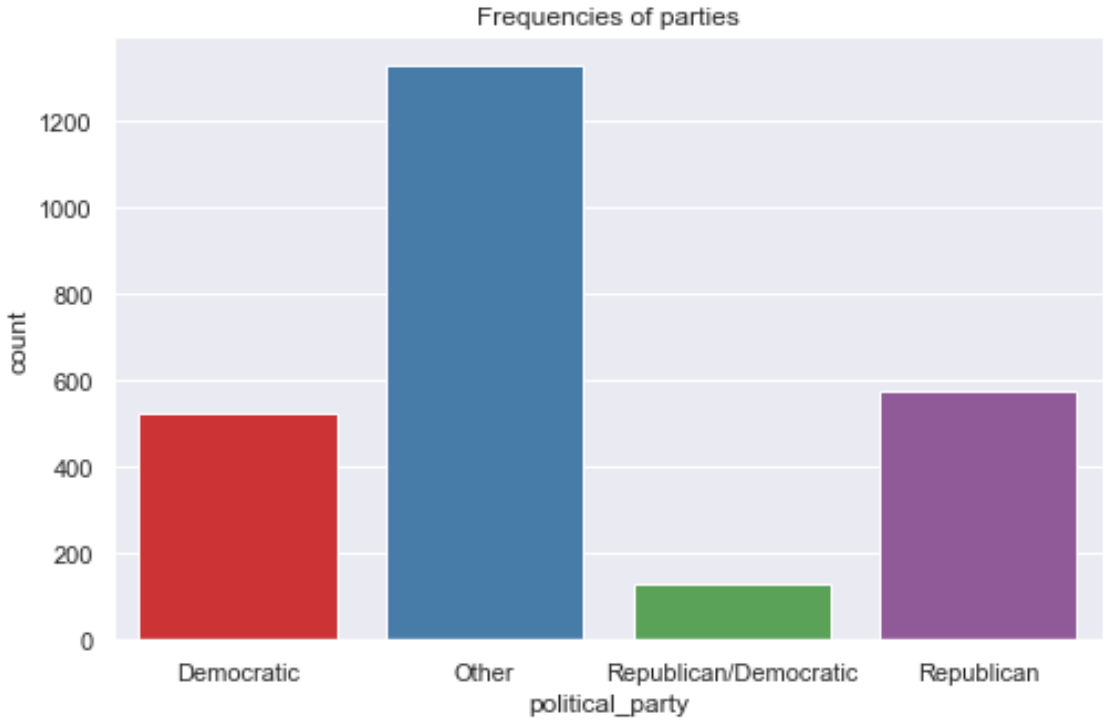
5.1 Reason codes: the US election dataset is much smaller than the sentiment analysis dataset

[106]:
```
print("The size of training set for sentiment analysis csv file:
 ↪",len(ysenti_train))
print("The size of training set for US election csv file:",len(elect_y_train))
```

```
The size of training set for sentiment analysis csv file: 77054
The size of training set for US election csv file: 1786
```

the majority of tweets are classified into "other" and "republican/democratic", and those results are not connected with the sentiment

[107]:
```
parties = ['Republican', 'Democratic', 'Republican/Democratic', 'Other']
sns.set(rc={'figure.figsize':(8,5)})
sns.countplot(x='political_party', label=parties,␣
 ↪data=election1,palette="Set1").set_title('Frequencies of parties')
plt.show()
```



5.2 Negative reason prediction on US election data

- According to the result of negative reasons prediction on the US election dataset, logistic regression still performs the best with an accuracy of 38%. However, the classification report implies overfitting as the accuracy for the training set is higher than the accuracy for the test set. Also, the size of the training set is really small, which is only 268. Also, refer to the visualization of negative reasons for each party, the "covid-19" counts for the majority, the classes of target values for the negative reason prediction are imbalanced.
- Possible improved methods:
  - Scraping more data from Twitter or combine similar negative reasons
  - Increasing 5-fold cross-validation to 10-fold cross-validation.
  - Using word embedding (Word2Vec) or N-gram for feature selection instead of Bag of Words and TF-IDF
  - Using resampling methods (oversampling/undersampling/SMOTE) to distribute the amount of data in each class uniformly

5.2 Reason codes Issue of overfitting

[108]:
```
prediction_neglr_train = neglr.predict(neg_elect_x_train)
print("Train accuracy for negative reasons:",accuracy_score(neg_elect_y_train,␣
 ↪prediction_neglr_train))
print("Test accuracy for negative reasons:",accuracy_score(neg_elect_y_test,␣
 ↪prediction_neglr))
```

```
Train accuracy for negative reasons: 0.4216417910447761
Test accuracy for negative reasons: 0.3826086956521739
```

Small training set

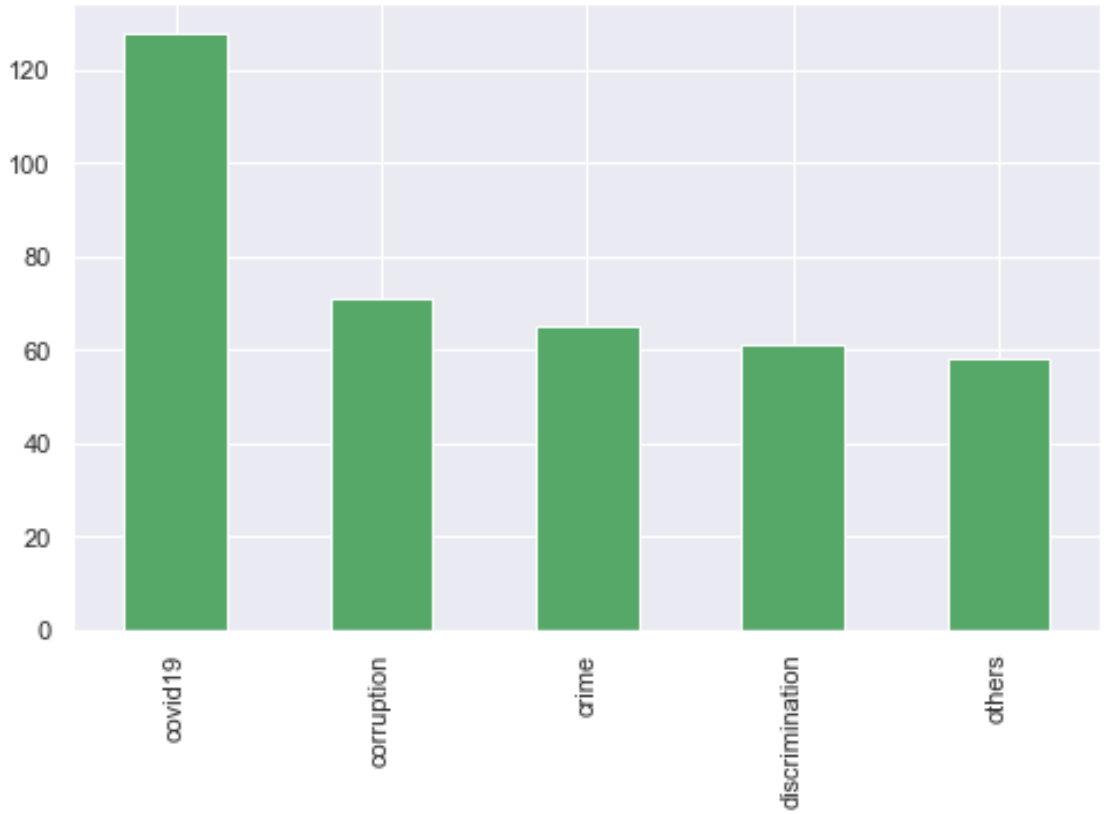
[109]:
```
len(neg_elect_y_train)
```

[109]: 268

Imbalanced dataset

[110]:
```
print(neg_elect["negative_reason"].value_counts().plot(kind="bar",color="g"))
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
```


120
100
80
60
40
20
0
covid19
corruption
crime
discrimination
others