Data Collection

To collect all the needed training data, I need to use mostly of the same methods I used to get the sample data. But instead of getting asll the reviews from one game, I need a wide variety of games. I start by collecting the appid's of the 750 most popular games on Steam. Then, I collect the 100 "most helpful" reviews from each game. This should collect 75,000 reviews, but many games have less than 100 reviews, leaving me with 73,096 total data points for this dataset.

In [1]:
```
from bs4 import BeautifulSoup
import pandas as pd
import requests
```

In [2]:
```
def get_reviews(appid, params):
        url_start = 'https://store.steampowered.com/appreviews/'
        response = requests.get(url=url_start+appid, params=params, headers={'Use
        return response.json() # return data extracted from the json response
```

In [3]:
```
def get_n_reviews(appid, n=100):
    reviews = []
    cursor = '*'
    params = { # https://partner.steamgames.com/doc/store/getreviews
            'json' : 1,
            'filter' : 'all', # sort by: recent, updated, all (helpfullness)
            'language' : 'english', # https://partner.steamgames.com/doc/store/lo
            'day_range' : 9223372036854775807, # shows reveiws from all time
            'review_type' : 'all', # all, positive, negative
            'purchase_type' : 'all', # all, non_steam_purchase, steam
        }
    while n > 0:
        params['cursor'] = cursor.encode() # for pagination
        params['num_per_page'] = min(100, n) # 100 is the max possible reviews in
        n -= 100

        response = get_reviews(appid, params)
        cursor = response['cursor']
        reviews += response['reviews']

        if len(response['reviews']) < 100: break

    return reviews
```

In [4]:
```
def get_n_appids(n=100, filter_by='topsellers'):
    appids = []
    url = f'https://store.steampowered.com/search/?category1=998&filter={filter_b
    page = 0

    while page*25 < n:
        page += 1
        response = requests.get(url=url+str(page), headers={'User-Agent': 'Mozil
        soup = BeautifulSoup(response.text, 'html.parser')
        for row in soup.find_all(class_='search_result_row'):
            appids.append(row['data-ds-appid'])

    return appids[:n]
```

In [5]:
```
reviews = []
appids = get_n_appids(750)
for appid in appids:
    reviews += get_n_reviews(appid, 100)
df = pd.DataFrame(reviews)[['review', 'voted_up']]
df
```

Out[5]:

	review	voted_up
0	I wanted to wait until I had 100 hours into th...	True
1	I don't know how these devs did it, but I have...	True
2	Has more game play, less bugs, and is polished...	True
3	I am very impressed with this game. Its worth...	True
4	Imagine if Rust and Runescape had a baby (with...	True
...	...	...
73091	70 hours in. No crashes, no slow-mo glitches, ...	True
73092	We still need a good GM mode.	False
73093	WWE 2K19 is a wrestling simulation game. It's ...	True
73094	Alrightie, where do I begin?\nI fell in love w...	False
73095	I dont think they even test these games before...	False

73096 rows × 2 columns

In [6]:
```
df.dropna(inplace=True)
df.reset_index(inplace=True)
df.voted_up.value_counts(normalize=True)
```

Out[6]:
```
True     0.805858
False    0.194142
Name: voted_up, dtype: float64
```

In [7]:
```
df.to_feather('../data/reviews_raw.feather')
```

These games were taken from the hot games section on Steam, which combines popularity and recency to come up with its list. Steam has over 50,000 games on it, so not every game can be taken into the dataset, but this is likely too restrictive. The games on the list or mostly popular and well-recieved games, which likely inflates the class imbalanace in favor of positive reviews. The games also represent current trends in gaming, which limits the model's ability to generalize to all games.

As well, a future improvement could be to get data from other sources in addition to Steam. Metacritic seems like a good choice, as its reviews come with scores. There are also storefronts, such as itch.io or GOG that cater to different types of games than the popular section of Steam or Metacritic, and so might help the model's ability to generalize.

However, at present, adding more data would just prevent my computer from running these models at all. If I want to increase the dataset size, I first need to get these notebooks running on a better computer, or on something like Amazon Sagemaker.

Data Processing

In addition to reading in the data from feather files (smaller file sizes than csv), I also perform the train-test split here. Perhaps this could have been done after the processing, but I wanted to make sure the different processed files were in the same order.

In [1]:
```
# This allows importing of scripts, which are stored in a folder one level up
import sys
sys.path.append('..')
```

In [2]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
```

In [4]:
```
df = pd.read_feather('../data/reviews_raw.feather').set_index('index')
df
```

Out[4]:

	review	voted_up
index		
0	I wanted to wait until I had 100 hours into th...	True
1	I don't know how these devs did it, but I have...	True
2	Has more game play, less bugs, and is polished...	True
3	I am very impressed with this game. Its worth...	True
4	Imagine if Rust and Runescape had a baby (with...	True
...	...	...
73091	70 hours in. No crashes, no slow-mo glitches, ...	True
73092	We still need a good GM mode.	False
73093	WWE 2K19 is a wrestling simulation game. It's ...	True
73094	Alrightie, where do I begin?\nI fell in love w...	False
73095	I dont think they even test these games before...	False

73096 rows × 2 columns

In [5]:
```
df_train, df_test = train_test_split(df, test_size=0.2, random_state=404)
X_train, y_train = df_train['review'].tolist(), df_train['voted_up'].tolist()
X_test, y_test = df_test['review'].tolist(), df_test['voted_up'].tolist()
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[5]:
```
(58476, 58476, 14620, 14620)
```

In [7]:
```
pd.DataFrame(y_train, columns=['voted_up']).to_feather('../data/processed/y_trair
pd.DataFrame(y_test, columns=['voted_up']).to_feather('../data/processed/y_test.f
```

Preprocessing

Most of the functions I use are the same as in the sample dataset. I've functionalized them in scripts/preprocessing.py, so I can later use them with the target data. These are seperated out instead of put in a pipeline to help with debugging errors. The large data size caused many errors and long runtime, so running these steps individually was the best way to make it work. Not using pipelines now may also allow me to not use scikit-learn in a final product, which could help in getting all the libraries I need loaded onto heroku.

In [9]:
```
from scripts import preprocessing
from nltk.corpus import stopwords
from string import punctuation
```

In [10]:
```
X_train_pre = list(map(preprocessing.remove_markdown, X_train))
X_test_pre = list(map(preprocessing.remove_markdown, X_test))
```

In [11]:
```
X_train_pre = list(map(preprocessing.remove_punctuation, X_train_pre))
X_test_pre = list(map(preprocessing.remove_punctuation, X_test_pre))
```

In [12]:
```
X_train_pre = list(map(preprocessing.tokenize, X_train_pre))
X_test_pre = list(map(preprocessing.tokenize, X_test_pre))
```

In [13]:
```
X_train_pre = list(map(preprocessing.lemmatize, X_train_pre))
X_test_pre = list(map(preprocessing.lemmatize, X_test_pre))
```

In [14]:
```
X_train_join = [' '.join(x) for x in X_train_pre]
X_test_join = [' '.join(x) for x in X_test_pre]
```

In [15]:
```
stopwords_list = stopwords.words('english') + list(punctuation) + ['`', '’', '…',
```

In [17]:
```
X_train_stopword = []
for review in X_train_pre:
    X_train_stopword.append([word for word in review if word not in stopwords_lis

X_test_stopword = []
for review in X_test_pre:
    X_test_stopword.append([word for word in review if word not in stopwords_list
```

In [18]:
```
pd.DataFrame([' '.join(x) for x in X_train_stopword], columns=['review']).to_feat
pd.DataFrame([' '.join(x) for x in X_test_stopword], columns=['review']).to_feath
```

Feature Engineering

I already know that TF-IDF performs the best, but I'm still interested to see howneural networks perform with the gensim document embeddings. These embeddings are much quicker and smaller than the TF-IDF vectorizers, so it isn't any trouble to run and save the data.

TF-IDF

In [19]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

In [20]:
```
tf = TfidfVectorizer(max_features=8000, stop_words=stopwords_list)
X_train_tf = pd.DataFrame(tf.fit_transform(X_train_join).todense(), columns=tf.ge
X_test_tf = pd.DataFrame(tf.transform(X_test_join).todense(), columns=tf.get_feat
```

In [21]:
```
X_train_tf.to_feather('../data/processed/X_train_tf.feather')
X_test_tf.to_feather('../data/processed/X_test_tf.feather')
```

TF-IDF with Bigrams

TF-IDF with Bigrams performed the best after running the models, so I pickled the vectorizer to use again later. When I get the ability to run bigger models and vectorizers, I may come back and try other levels of n-grams.

In [13]:
```
from pickle import dump
```

In [22]:
```
tf_bigram = TfidfVectorizer(max_features=8000, ngram_range=(1,2))
X_train_bigram = pd.DataFrame(tf_bigram.fit_transform(X_train_join).todense(), co
X_test_bigram = pd.DataFrame(tf_bigram.transform(X_test_join).todense(), columns=
```

In [23]:
```
X_train_bigram.to_feather('../data/processed/X_train_bigram.feather')
X_test_bigram.to_feather('../data/processed/X_test_bigram.feather')
```

In [16]:
```
dump(tf_bigram, open('../final_model/vectorizer.pk', 'wb'))
```

Document Embeddings

In [24]:
```
from gensim.sklearn_api import D2VTransformer
from sklearn.preprocessing import MinMaxScaler
```

In [25]:
```
d2v = D2VTransformer()
X_train_embed = d2v.fit_transform(X_train_pre)
X_test_embed = d2v.transform(X_test_pre)

scaler = MinMaxScaler((1, 2))
X_train_embed = pd.DataFrame(scaler.fit_transform(X_train_embed))
X_test_embed = pd.DataFrame(scaler.transform(X_test_embed))

X_train_embed.columns = X_train_embed.columns.astype(str)
X_test_embed.columns = X_test_embed.columns.astype(str)
```

In [26]:
```
X_train_embed.to_feather('../data/processed/X_train_embed.feather')
X_test_embed.to_feather('../data/processed/X_test_embed.feather')
```

Some of these final processed files are too alrge to upload to Github, so the entire data/processed folder has been added to .gitignore. You will need to run this script yourself to generate the same files. The raw data is still included in the Github upload.

Exploratory Data Analysis

This processed data is not uploaded to the Github repo, as some of the files are too large. Run notebook 2 in order to produce the same files.

In [3]:
```
import pandas as pd
```

In [4]:
```
X_train = pd.read_feather('../data/processed/X_train_preprocessed.feather')
y_train = pd.read_feather('../data/processed/y_train.feather')
data = pd.concat([X_train, y_train], axis=1)
data
```

Out[4]:

	review	voted_up
0	apparently fault whenever dont save teammate t...	True
1	get level 20farm potsget 88 die 1 shock denial...	True
2	played game almost 1000 hour seen go good exce...	False
3	play havent heard high elvesif play	True
4	personally found game frustrating thing loved ...	False
...	...	...
58471	first game kind ever ever enjoyed something al...	True
58472	overall decent game early accesspro great buil...	True
58473	want great sniper moment realism game purchase...	True
58474	game warhammer reskin new racesunits marked im...	True
58475	profound journey depth mean humanin era consci...	True

58476 rows × 2 columns

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 58476 entries, 0 to 58475
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   review    58476 non-null  object
 1   voted_up  58476 non-null  bool  
dtypes: bool(1), object(1)
memory usage: 514.1+ KB
```

Total Vocabulary

The dataset has over 200,000 tokens, and this is even before bigrams are factored in. I can't run a model on this many tokens, and few of them will be of any relative importance anyways, so only some of the most popular will be used in the final model.

In [6]:
```
total_vocabulary = []
for review in data['review'].tolist():
    total_vocabulary += review.split()
```

In [7]:
```
print('There are {} unique tokens in the dataset.'.format(len(set(total_vocabular
```

```
There are 223823 unique tokens in the dataset.
```

Frequency Distributuion

Many of the most common words are used in both classes. Words like "game" and "play" make sense, but even "like" has similar representation across the two classes. In order to get a truly good representation of the differences between the positive and negative reviews, I would need to remove words that have a certain level of representation in both classes. As well, including bigrams may yield intresting results here.

In [8]:
```
import matplotlib.pyplot as plt
from nltk import FreqDist
plt.style.use('seaborn')
plt.style.use('seaborn-talk')
```

In [9]:
```
data['voted_up'].value_counts(normalize=True)
```

Out[9]:
```
True     0.805903
False    0.194097
Name: voted_up, dtype: float64
```

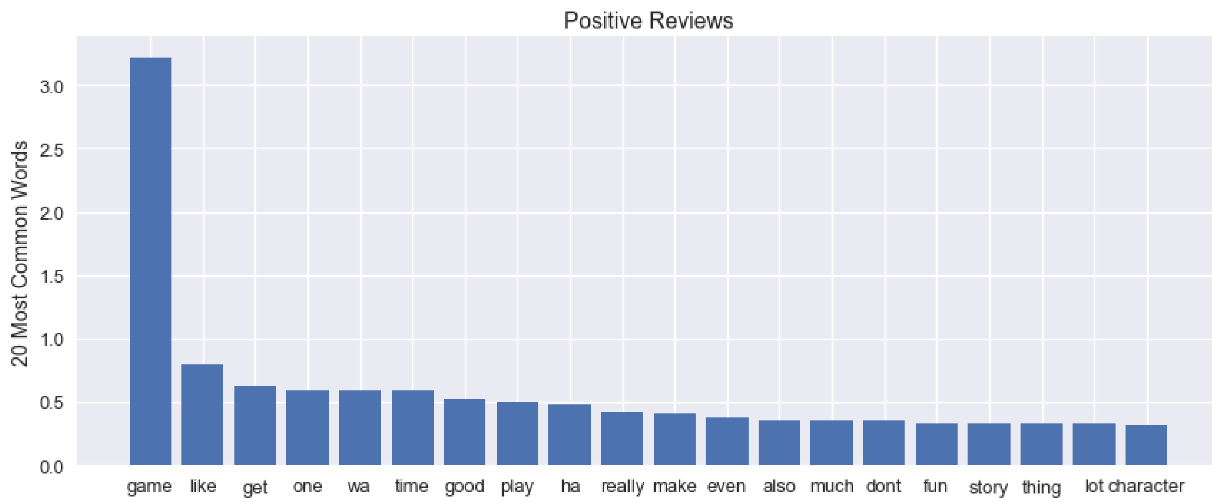
In [10]:
```
reviews_pos = data[data['voted_up']]['review']
reviews_neg = data[~data['voted_up']]['review']
```

In [11]:
```
vocab_pos = []
for review in reviews_pos.tolist():
    vocab_pos += review.split()

vocab_neg = []
for review in reviews_neg.tolist():
    vocab_neg += review.split()
```

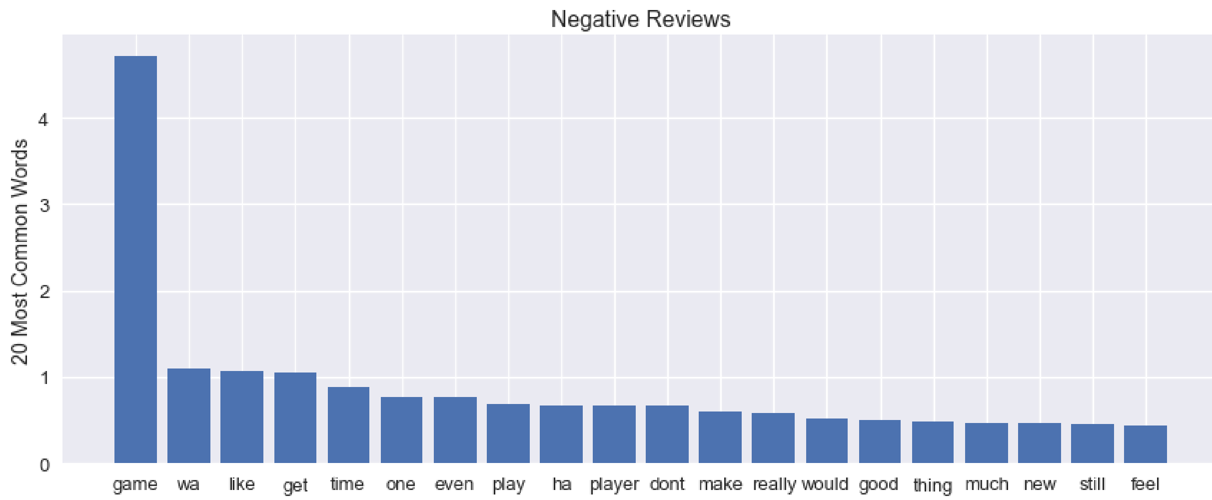
In [12]:
```
freqdist_pos = FreqDist(vocab_pos)
top_20_pos = freqdist_pos.most_common(20)

freqdist_neg = FreqDist(vocab_neg)
top_20_neg = freqdist_neg.most_common(20)
```

In [13]:
```
words_pos, values_pos = list(zip(*top_20_pos))
values_pos_norm = tuple(v/len(reviews_pos) for v in values_pos)

words_neg, values_neg = list(zip(*top_20_neg))
values_neg_norm = tuple(v/len(reviews_neg) for v in values_neg)
```

In [14]:
```
fig = plt.figure(figsize=(12, 5))
plt.bar(words_pos, values_pos_norm, figure=fig)
plt.ylabel('Average Occurances per Review')
plt.ylabel('20 Most Common Words')
plt.title('Positive Reviews')

plt.tight_layout()
plt.savefig('../visualizations/frequency-distribution-positive.png')
plt.show()
```



In [22]:
```
fig = plt.figure(figsize=(12, 5))
plt.bar(words_neg, values_neg_norm, figure=fig)
plt.ylabel('Average Occurances per Review')
plt.ylabel('20 Most Common Words')
plt.title('Negative Reviews')

plt.tight_layout()
plt.savefig('../visualizations/frequency-distribution-negative.png')
plt.show()
```



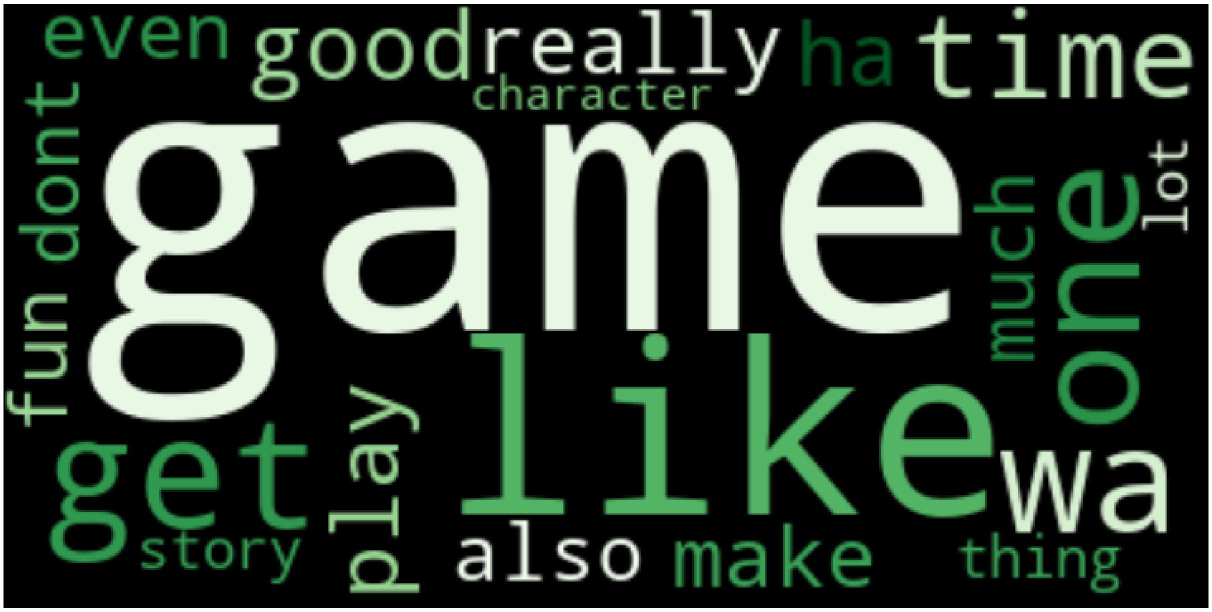
Word Clouds

These results are more or less the same as the frequency distribution results. A lot more work needs to be done here.

In [19]:
```
from wordcloud import WordCloud
```

In [18]:
```
positive_dict = dict(zip(words_pos, values_pos))
negative_dict = dict(zip(words_neg, values_neg))
```

In [32]:
```
wordcloud = WordCloud(colormap='Greens').generate_from_frequencies(positive_dict)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")

plt.tight_layout()
plt.savefig('../visualizations/wordcloud-positive.png')
plt.show()
```



In [34]:
```
wordcloud = WordCloud(colormap='Reds').generate_from_frequencies(negative_dict)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")

plt.tight_layout()
plt.savefig('../visualizations/wordcloud-negative.png')
plt.show()
```

Overall I am disappointed with this EDA. Most of these results are less than helpful, and work needs to be done in order to get real results. I ran out of time with this project, but I'd like to come back to this and get it working better.

Base Models

This processed data is not uploaded to the Github repo, as some of the files are too large. Run notebook 2 in order to produce the same files.

Baseline Models

Even though I ran this type of model analysis with the sample data, I want to run it again now that I've created bigrams. I'm also no longer using SVM, as it took way too long to run even on the smaller dataset. Once again, logistic regression is the best performing model. Also as expected, bigrams improved the accuracy of the models.

In [1]:
```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
```

In [2]:
```
def get_model_metrics(X_train, y_train, X_test, y_test, model, model_name, data_r

    model.fit(X_train, y_train)
    y_train_hat = model.predict(X_train)
    y_test_hat = model.predict(X_test)

    acc_train = accuracy_score(y_train, y_train_hat)
    pre_train = precision_score(y_train, y_train_hat)
    rec_train = recall_score(y_train, y_train_hat)

    acc_test = accuracy_score(y_test, y_test_hat)
    pre_test = precision_score(y_test, y_test_hat)
    rec_test = recall_score(y_test, y_test_hat)

    metrics = {'Model': model_name,
               'Processing': data_name,
               'Test Accuracy': acc_test,
               'Test Precision': pre_test,
               'Test Recall': rec_test,
               'Train Accuracy': acc_train,
               'Train Precision': pre_train,
               'Train Recall': rec_train}

    return metrics
```

In [ ]:
```
datasets = [('TF-IDF', 'tf'),
        ('TF-IDF with Bigrams', 'bigram'),
        ('Document Embeddings', 'embed')]
models = [('Logistic Regression', LogisticRegression(solver='saga')),
          ('Multinomial Naive Bayes', MultinomialNB()),
          ('Random Forest', RandomForestClassifier())]
metrics = []
```

In [ ]:
```
y_train = pd.read_feather('../data/processed/y_train.feather')['voted_up'].to_nur
y_test = pd.read_feather('../data/processed/y_test.feather')['voted_up'].to_numpy

for data_name, file in datasets:
    X_train = pd.read_feather(f'../data/processed/X_train_{file}.feather').to_nur
    X_test = pd.read_feather(f'../data/processed/X_test_{file}.feather').to_numpy
    for model_name, model in models:
        print(model_name, data_name)
        metrics.append(get_model_metrics(X_train, y_train, X_test, y_test, model,

metrics.append(get_model_metrics(X_train, y_train, X_test, y_test, DummyClassifie
```

In [ ]:
```
metrics_df = pd.DataFrame(metrics)
metrics_df.sort_values(by='Test Accuracy', ascending=False)
```

Gridsearch

Here I performed a gridsearch on the random forest and logistic regression models using just the bigram data, as it performed the best. Naive Bayes models do not have any hyperparameters to tune, and so there is no grid search to perform on it.

In [2]:
```
from sklearn.model_selection import GridSearchCV
```

In [3]:
```
y_train = pd.read_feather('../data/processed/y_train.feather')['voted_up'].to_nur
X_train = pd.read_feather('../data/processed/X_train_bigram.feather').to_numpy()
```

In [4]:
```
param_grid_lr = {'C': [0.1, 1, 10],
                 'class_weight': ['balanced', None],
                 'solver': ['saga']}
gs_lr = GridSearchCV(estimator=LogisticRegression(), param_grid=param_grid_lr, sc
gs_lr.fit(X_train, y_train)
gs_lr.best_params_
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
```

Out[4]:
```
{'C': 10, 'class_weight': None, 'solver': 'saga'}
```

In [5]:
```
param_grid_rf = {'n_estimators': [100, 250],
                 'max_features': ['auto', 150],
                 'class_weight': ['balanced', None]}
gs_rf = GridSearchCV(estimator=RandomForestClassifier(), param_grid=param_grid_rf
gs_rf.fit(X_train, y_train)
gs_rf.best_params_
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[CV 1/3] END class_weight=balanced, max_features=auto, n_estimators=100; tota
l time= 5.8min
[CV 2/3] END class_weight=balanced, max_features=auto, n_estimators=100; tota
l time= 5.8min
[CV 3/3] END class_weight=balanced, max_features=auto, n_estimators=100; tota
l time= 5.6min
[CV 1/3] END class_weight=balanced, max_features=auto, n_estimators=250; tota
l time=13.7min
[CV 2/3] END class_weight=balanced, max_features=auto, n_estimators=250; tota
l time=13.8min
[CV 3/3] END class_weight=balanced, max_features=auto, n_estimators=250; tota
l time=13.5min
[CV 1/3] END class_weight=balanced, max_features=150, n_estimators=100; total
time= 8.2min
[CV 2/3] END class_weight=balanced, max_features=150, n_estimators=100; total
time= 8.3min
[CV 3/3] END class_weight=balanced, max_features=150, n_estimators=100; total
time= 8.2min
```

Final Models

After comparing the best tuned models, logistic regression still has the best accuracy. I had expected random forest to improve performance more with tuning, but it seems not to be the case here.

In [3]:
```
y_train = pd.read_feather('../data/processed/y_train.feather')['voted_up'].to_nun
y_test = pd.read_feather('../data/processed/y_test.feather')['voted_up'].to_numpy
X_train = pd.read_feather('../data/processed/X_train_bigram.feather').to_numpy()
X_test = pd.read_feather('../data/processed/X_test_bigram.feather').to_numpy()
```

In [4]:
```
lr_final = LogisticRegression(C=10, solver='saga')
nb_final = MultinomialNB()
rf_final = RandomForestClassifier(max_features=150)

final_metrics = []
print('starting Logistic Regression model')
final_metrics.append(get_model_metrics(X_train, y_train, X_test, y_test, lr_final
print('starting Naive Bayes model')
final_metrics.append(get_model_metrics(X_train, y_train, X_test, y_test, nb_final
print('starting Random Forest model')
final_metrics.append(get_model_metrics(X_train, y_train, X_test, y_test, rf_final
print('completed models')

final_metrics_df = pd.DataFrame(final_metrics)
final_metrics_df.sort_values(by='Test Accuracy', ascending=False)
```

```
starting Logistic Regression model
starting Naive Bayes model
starting Random Forest model
completed models
```

Out[4]:

	Model	Processing	Test Accuracy	Test Precision	Test Recall	Train Accuracy	Train Precision	Train Recall
0	Logistic Regression	TF-IDF with Bigrams	0.914090	0.934010	0.961287	0.947295	0.956907	0.978674
1	Multinomial Naive Bayes	TF-IDF with Bigrams	0.871614	0.869202	0.989558	0.877557	0.874733	0.989815
2	Random Forest	TF-IDF with Bigrams	0.866963	0.864709	0.989727	0.998957	0.998940	0.999767

Save Model

Even though I will also be creating a neural network model, I still want to save this best logistic regression model. I can try to use it as a backup in case the neural network model is too big to upload to heroku.

In [4]:
```
import pickle
```

In [5]:
```
model = LogisticRegression(C=10, solver='saga')
model.fit(X_train, y_train)
```

Out[5]:
```
LogisticRegression(C=10, solver='saga')
```

In [7]:
```
pickle.dump(model, open('../final_model/sklearn-logreg/model.pk', 'wb'))
```

Neural Networks

In [1]:
```
import pandas as pd
```

In [2]:
```
y_train = pd.read_feather('../data/processed/y_train.feather')['voted_up'].to_nur
X_train = pd.read_feather(f'../data/processed/X_train_bigram.feather').to_numpy()
```

In [3]:
```
y_test = pd.read_feather('../data/processed/y_test.feather')['voted_up'].to_numpy
X_test = pd.read_feather(f'../data/processed/X_test_bigram.feather').to_numpy()
```

Baseline Neural Network

Here I build a simple sequential convolutional neural network. Although it only has dense and dropout layers, it stillperforms great.

In [4]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score
from tensorflow.keras.layers import Dense, Dropout, LSTM
from tensorflow.keras import Sequential
```

In [8]:
```
model = Sequential()

# hidden layers
model.add(Dense(500, input_dim=8000, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))

# output layer
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [9]:
```
model.fit(X_train, y_train, epochs=3, batch_size=32, validation_data=(X_test, y_t
model.evaluate(X_test, y_test)
```

```
Epoch 1/3
1828/1828 [==============================] - 61s 32ms/step - loss: 0.3386 - acc
uracy: 0.8589 - val_loss: 0.2210 - val_accuracy: 0.9097
Epoch 2/3
1828/1828 [==============================] - 53s 29ms/step - loss: 0.1787 - acc
uracy: 0.9328 - val_loss: 0.2159 - val_accuracy: 0.9190
Epoch 3/3
1828/1828 [==============================] - 54s 30ms/step - loss: 0.1305 - acc
uracy: 0.9540 - val_loss: 0.2229 - val_accuracy: 0.9180
457/457 [==============================] - 4s 9ms/step - loss: 0.2229 - accurac
y: 0.9180
```

Out[9]:
```
[0.2228986769914627, 0.9179890751838684]
```

0.9179890751838684

Top performing base model: 0.914090 test accuracy

Hyperparameter Tuning

Ths hyperparameter values, and in fact the structure of this network, is mostly guesswork. I don't have a good idea of what values to use here, or how many layers are needed. In addition to searching over a wider range of values, I'd like to grid search over varying numbers of layers and epochs. Again, I would need more time and processing power to try this.

In [5]:
```
from talos import Scan
```

In [6]:
```
def dense_network(x_train, y_train, x_val, y_val, params):
    model = Sequential()

    # hidden layers
    model.add(Dense(params['dense'], input_dim=8000, activation=params['activatio
    model.add(Dropout(params['dropout']))
    model.add(Dense(params['dense']*2, activation=params['activation1']))
    model.add(Dropout(params['dropout']))
    model.add(Dense(params['dense']*0.5, activation=params['activation1']))
    model.add(Dropout(params['dropout']))
    model.add(Dense(params['dense']*0.75, activation=params['activation1']))
    # output layer
    model.add(Dense(1, activation=params['activation2']))

    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accurac

    out = model.fit(x_train, y_train,
                    validation_data=(x_val, y_val),
                    epochs=5,
                    verbose=0)

    return out, model
```

In [7]:
```
params = {'dropout': [0.25, 0.5, 0.75],
          'dense': [10, 50, 100, 500],
          'activation1': ['relu', 'elu'],
          'activation2': ['sigmoid', 'tanh']}
```

In [8]:
```
results = Scan(X_train, y_train, params=params, model=dense_network, experiment_r
results.best_model(metric='accuracy')
```

```
100%|██████████| 48/48 [1:14:46<00:00, 93.47s/it]
```

Out[8]:
```
<tensorflow.python.keras.engine.sequential.Sequential at 0x165c9f4cb08>
```

In [9]:
```
pd.read_csv('grid/022421165029.csv').sort_values('val_accuracy', ascending=False)
```

Out[9]:

	round_epochs	loss	accuracy	val_loss	val_accuracy	activation1	activation2	dense
10	5	0.038711	0.988078	0.414048	0.911247	relu	sigmoid	500
22	5	0.107399	0.980114	0.465812	0.910050	relu	tanh	500
41	5	0.338274	0.902475	0.373536	0.909252	elu	tanh	50
11	5	0.139434	0.951653	0.270693	0.908853	relu	sigmoid	500
4	5	0.113932	0.960887	0.254306	0.908682	relu	sigmoid	50
23	5	0.192981	0.952728	0.379954	0.907712	relu	tanh	500
9	5	0.018463	0.994503	0.530807	0.907655	relu	sigmoid	500
44	5	0.345217	0.908875	0.448334	0.907484	elu	tanh	100
18	5	0.041759	0.990765	0.592345	0.907085	relu	tanh	100
26	5	0.318125	0.877287	0.241634	0.906743	elu	sigmoid	10
29	5	0.195315	0.930374	0.244876	0.906401	elu	sigmoid	50

Final Model

The top performing model looks very similar to the first model I made, and performs nearly the same as well. I would like to expand my gridsearch when I get a chance, but for now this model is my best, so I wil save it to use on the unlabeled data.

In [5]:
```python
model = Sequential()

# hidden layers
model.add(Dense(500, input_dim=8000, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(125, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(250, activation='relu'))

# output layer
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [6]:
```python
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.1)
```

```
Epoch 5/10
1645/1645 [==============================] - 50s 30ms/step - loss: 0.0404 - a
ccuracy: 0.9877 - val_loss: 0.3743 - val_accuracy: 0.90660s - loss: 0.0404 -
ac
Epoch 6/10
1645/1645 [==============================] - 51s 31ms/step - loss: 0.0297 - a
ccuracy: 0.9902 - val_loss: 0.3590 - val_accuracy: 0.9061
Epoch 7/10
1645/1645 [==============================] - 49s 30ms/step - loss: 0.0236 - a
ccuracy: 0.9929 - val_loss: 0.4836 - val_accuracy: 0.9075
Epoch 8/10
1645/1645 [==============================] - 49s 30ms/step - loss: 0.0214 - a
ccuracy: 0.9933 - val_loss: 0.5731 - val_accuracy: 0.9078
Epoch 9/10
1645/1645 [==============================] - 49s 30ms/step - loss: 0.0166 - a
ccuracy: 0.9953 - val_loss: 0.6054 - val_accuracy: 0.9073
Epoch 10/10
1645/1645 [==============================] - 49s 30ms/step - loss: 0.0173 - a
ccuracy: 0.9953 - val_loss: 0.5127 - val_accuracy: 0.9078
```

In [7]:
```python
model.evaluate(X_test, y_test)
```

```
457/457 [==============================] - 4s 9ms/step - loss: 0.4588 - accurac
y: 0.9112
```

Out[7]: [0.4588494598865509, 0.9112175107002258]

In [8]:
```python
model.save('../final_model/model.h5')
model.save_weights('../final_model/model_weights.h5')
```

```python
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import RegexpTokenizer
from re import sub
from string import punctuation
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')

def remove_markdown(x):
    # remove markdown tags, only needed for Steam reviews
    # todo: check if tag is a link, and remove url in parenthesis
    # and keep the text in the brackets (without the brackets)
    # link format: [text to keep](www.urltoremove.com)
    # remove links as well
    return sub(r'\[.*?\]', '', x)

def remove_punctuation(x):
    # remove all punctuation, which is often freely not use on these user reviews
    punctuation_list = list(punctuation) + ['`', '’', '…', '\n']
    return x.translate(str.maketrans('', '', ''.join(punctuation_list)))

def tokenize(x):
    # tokenize words with only numbers and latin characters
    # also turns everything to lowercase
    # input is a single string, output is a list of strings
    tokenizer = RegexpTokenizer(r'[a-zA-Z0-9]+')
    return tokenizer.tokenize(x.lower())

def lemmatize(x):
    # expects list of strings as input
    lemmatizer = WordNetLemmatizer()
    return list(map(lemmatizer.lemmatize, x))

def make_bigrams(x):
    # expects list of strings as input
    # adds bigrams onto existing tokens
    grams = []
    for i in range(len(x)-(n-1)):
        gram = []
        for j in range(i, i+n):
            gram.append(x[j])
        grams.append(' '.join(gram))
    return x + grams

def remove_stopwords(x):
    # expects list of strings as input
    stopwords_list = stopwords.words('english') + self.punctuation_list
    return [word for word in x if word not in stopwords_list]

def unsplit(x):
    # recombines list of strings into single string
    # needed for TF-IDF vectorizer
    # not needed with doc2vec or make_bigrams
    return ' '.join(x)
```

```python
from scripts.config import reddit_api
from scripts import preprocessing

from tensorflow.keras.models import load_model
import pandas as pd
from pickle import load
from praw import Reddit
import twint

model = load_model('final_model/cnn-bigrams/model.h5')
model.load_weights('final_model/cnn-bigrams/model_weights.h5')
vectorizer = load(open('final_model/cnn-bigrams/vectorizer.pk', 'rb'))
#model = load(open('final_model/sklearn-logreg/model.pk', 'rb'))
#vectorizer = load(open('final_model/sklearn-logreg/vectorizer.pk', 'rb'))

def get_tweets(search, limit=1000): # get ~1000 most recent tweets from hashtag
    c = twint.Config()
    c.Limit = limit*2 # searching by language does not work, reducing to English-only reduces amount by about half
    c.Min_likes = 5
    c.Pandas = True
    c.Lang = 'en'
    c.Hide_output = True
    c.Search = search

    twint.run.Search(c)
    tweets = twint.storage.panda.Tweets_df
    tweets = tweets.loc[tweets['language']=='en']

    return tweets['tweet'].to_list()

def get_comments(url): # get all top-level coments from reddit thread
    reddit = Reddit(client_id=reddit_api['client_id'],
                    client_secret=reddit_api['client_secret'],
                    user_agent=reddit_api['user_agent'])

    submissionId = url[url.find('comments'):].split('/')[1]
    submission = reddit.submission(submissionId)
    submission.comments.replace_more(limit=None)
    comments = []
    for comment in submission.comments:
        comments.append(comment.body)

    return comments

def process_data(X): # run all preprocessing functions
    X_pre = list(map(preprocessing.remove_markdown, X))
    X_pre = list(map(preprocessing.remove_punctuation, X_pre))
    X_pre = list(map(preprocessing.tokenize, X_pre))
    X_pre = list(map(preprocessing.lemmatize, X_pre))
```

```python
from scripts import predictions

def get_examples():
    examples = {'twitter': ['#stateofplay',
                            '#pokemonpresents',
                            '#MonsterHunter'
                           ],
                'reddit': ['https://www.reddit.com/r/nintendo/comments/lm6obv/project_triangle_strategy_announcement_trailer/',
                           'https://www.reddit.com/r/nintendo/comments/lru33p/the_animal_crossing_new_horizons_free_update_is/',
                           'https://www.reddit.com/r/Games/comments/ls6qlh/bravery_default_ii_review_thread/',
                           'https://www.reddit.com/r/Games/comments/lw3wtu/aliens_fireteam_official_announcement_trailer/'
                          ]
               }

    return examples

def get_web_output(source, limit=1000, samples=5):
    df = predictions.get_predictions(source, limit)
    value_counts = df.positive.value_counts(normalize=True)
    pos_percentage = round(value_counts[True]*100)
    neg_percentage = round(value_counts[False]*100)
    pos_samples = df[df['positive']].sample(5)['review'].tolist()
    neg_samples = df[~df['positive']].sample(5)['review'].tolist()

    return {'pos_percentage': pos_percentage,
            'neg_percentage': neg_percentage,
            'pos_samples': pos_samples,
            'neg_samples': neg_samples}
```

```python
from flask import Flask, request, render_template

from scripts import WebInterface

app = Flask(__name__)
app.config['DEBUG'] = True

@app.route('/', methods = ['GET'])
def render_homepage():
    examples = WebInterface.get_examples()
    return render_template('homepage.html', data=examples)

@app.route('/', methods = ['POST'])
def predict():
    search = request.form.get('search')
    data = WebInterface.get_web_output(search)
    return render_template('output.html', data=data, source=search)

if __name__ == '__main__':
    app.run()
```