Business Problem

Unlike most other reviewing websites which calculate the average of review scores (i.e. Amazon 4.5/5 stars, IMDB 8.2/10 rating), Rotten Tomatoes scores media on a percentage out of 100. This percentage is aggregated completely based on whether the review is positive or negative.

Rotten Tomatoes currently aggregates reviews in two ways based on:

1. Opinions of film and television critics
2. Opinions of users

Rotten Tomatoes wants to explore what the rest of the internet is saying about film and television. They want to have the ability to efficiently analyze publicly available comments/chatter (i.e. Tweets, Reddit message boards, Facebook comments) and derive a percentage score for movies and television programming.

This project will focus on just the first step of solving this business problem. Here is our proposed hypothesis:

'A model can be derived to input opinionated language material written about a film or show, so that it can reliably predict positive or negative sentiment.'

Data Understanding

Our hypothesis will be tested on this IMDB data set (https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews) from kaggle. It contains 50,000 "highly polar" movie/show reviews and their positive or negative sentiment.

Data Preparation

Imports

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px

from bs4 import BeautifulSoup

import re

import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorize
import string
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score, precision_score, f1_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import roc_auc_score, plot_roc_curve
```

In [2]:
```
df = pd.read_csv('data/IMDB_Dataset.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

Check for duplicates

In [4]:
```
df.describe()
```

Out[4]:

	review	sentiment
count	50000	50000
unique	49582	2
top	Loved today's show!!! It was a variety and not...	positive
freq	5	25000

A small amount of duplicates. Since the data does not distinguish the media associated with the review, we will not remove any duplicates.

Check Target Distribution

In [5]:
```
df['sentiment'].value_counts()
```

Out[5]:
```
positive    25000
negative    25000
Name: sentiment, dtype: int64
```

This is a balanced dataset

Check Null Values

In [6]:
```
df.isna().sum()
```

Out[6]:
```
review       0
sentiment    0
dtype: int64
```

Explore a single review

In [7]:
```
df.loc[0][0]
```

Out[7]:
```
"One of the other reviewers has mentioned that after watching just 1 Oz e
pisode you'll be hooked. They are right, as this is exactly what happened
with me.<br /><br />The first thing that struck me about Oz was its bruta
lity and unflinching scenes of violence, which set in right from the word
GO. Trust me, this is not a show for the faint hearted or timid. This sho
w pulls no punches with regards to drugs, sex or violence. Its is hardcor
e, in the classic use of the word.<br /><br />It is called OZ as that is
the nickname given to the Oswald Maximum Security State Penitentary. It f
ocuses mainly on Emerald City, an experimental section of the prison wher
e all the cells have glass fronts and face inwards, so privacy is not hig
h on the agenda. Em City is home to many..Aryans, Muslims, gangstas, Lati
nos, Christians, Italians, Irish and more....so scuffles, death stares, d
odgy dealings and shady agreements are never far away.<br /><br />I would
say the main appeal of the show is due to the fact that it goes where oth
er shows wouldn't dare. Forget pretty pictures painted for mainstream aud
iences, forget charm, forget romance...OZ doesn't mess around. The first
episode I ever saw struck me as so nasty it was surreal, I couldn't say I
was ready for it, but as I watched more, I developed a taste for Oz, and
got accustomed to the high levels of graphic violence. Not just violence,
but injustice (crooked guards who'll be sold out for a nickel, inmates wh
o'll kill on order and get away with it, well mannered, middle class inma
tes being turned into prison bitches due to their lack of street skills o
r prison experience) Watching Oz, you may become comfortable with what is
uncomfortable viewing....thats if you can get in touch with your darker s
ide."
```

Text Cleaning

In [8]:
```
def text_preprocessor(review):
    # Strip html text
    soup = BeautifulSoup(review, "lxml")
    data = soup.get_text()
    # Lower case all text
    data = data.lower()
    # Remove edge cases with multiple periods
    pattern = re.compile(r'\s+')
    data = re.sub(pattern, ' ', data.replace('.', ' '))
    # Remove punctuation and other special characters
    pattern = r'[^a-zA-Z\s]'
    data = re.sub(pattern, '', data)
    # Tokenize
    data = word_tokenize(data)
    # Get list of nltk's stopwords
    stopwords_list = stopwords.words('english')
    # Remove stop words
    data = [word for word in data if word not in stopwords_list]
    # Initialize a PortStemmer object
    stemmer = PorterStemmer()
    # Convert the tokens into their stem
    data = [stemmer.stem(token) for token in data]
    # Convert the list of words back into
    # a string by joining each word with a space
    data = ' '.join(data)
    # Remove double spaces
    data = data.replace('  ', ' ')
    # Remove opening and trailing spaces
    data = data.strip()
    # Return the cleaned text data
    return data
```

In [9]:
```
df.review = df.review.apply(text_preprocessor)
```

In [10]:
```
print(df.review[:5])
```

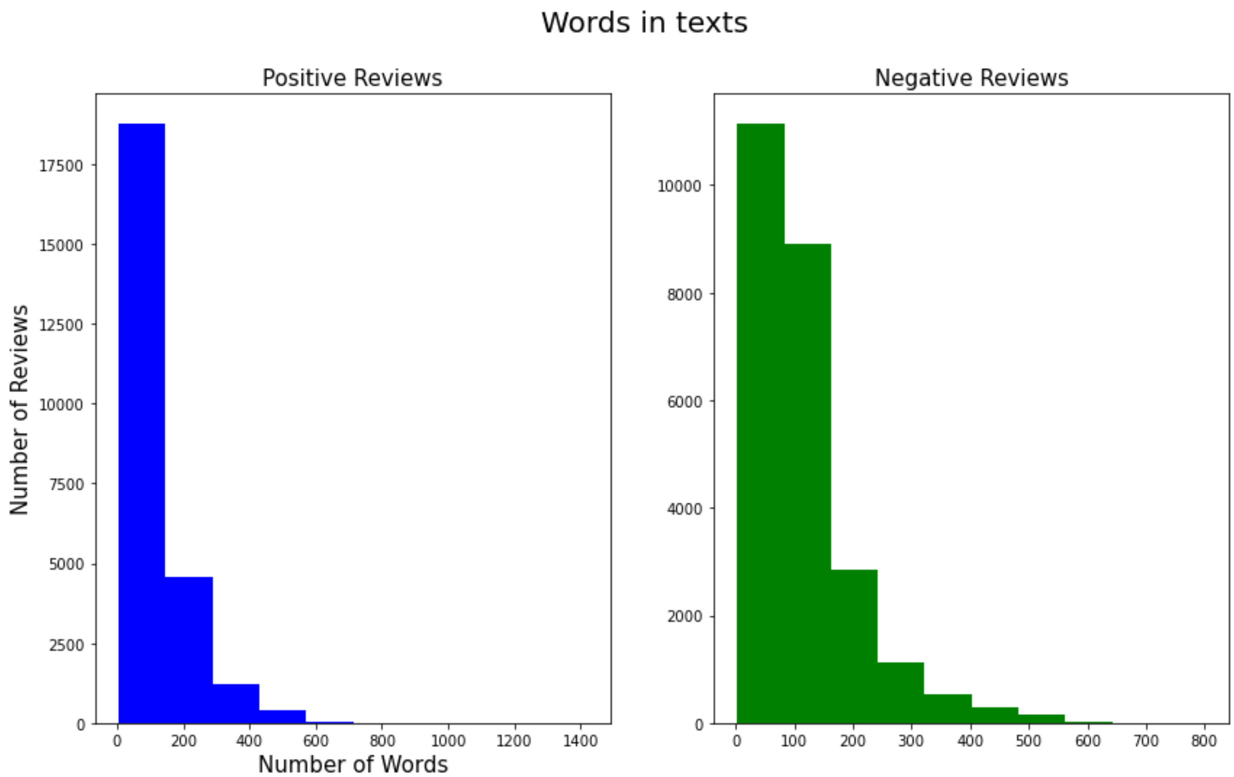
```
0    one review mention watch oz episod youll hook ...
1    wonder littl product film techniqu unassum old...
2    thought wonder way spend time hot summer weeke...
3    basic there famili littl boy jake think there ...
4    petter mattei love time money visual stun film...
Name: review, dtype: object
```

Data Exploration

Now that we have cleaned our text, we will analyze some of our data's characteristics:

Number of words per review

In [11]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 8))
length_good_reviews = df[df['sentiment'] == 'positive']['review'].str.split
ax1.hist(length_good_reviews, color='blue')
ax1.set_title('Positive Reviews', fontsize=15)
ax1.set_ylabel('Number of Reviews', fontsize=15)
ax1.set_xlabel('Number of Words', fontsize=15)
length_bad_reviews = df[df['sentiment'] == 'negative']['review'].str.split(
ax2.hist(length_bad_reviews, color='green')
ax2.set_title('Negative Reviews', fontsize=15)
fig.suptitle('Words in texts', fontsize=20)
plt.show()
```



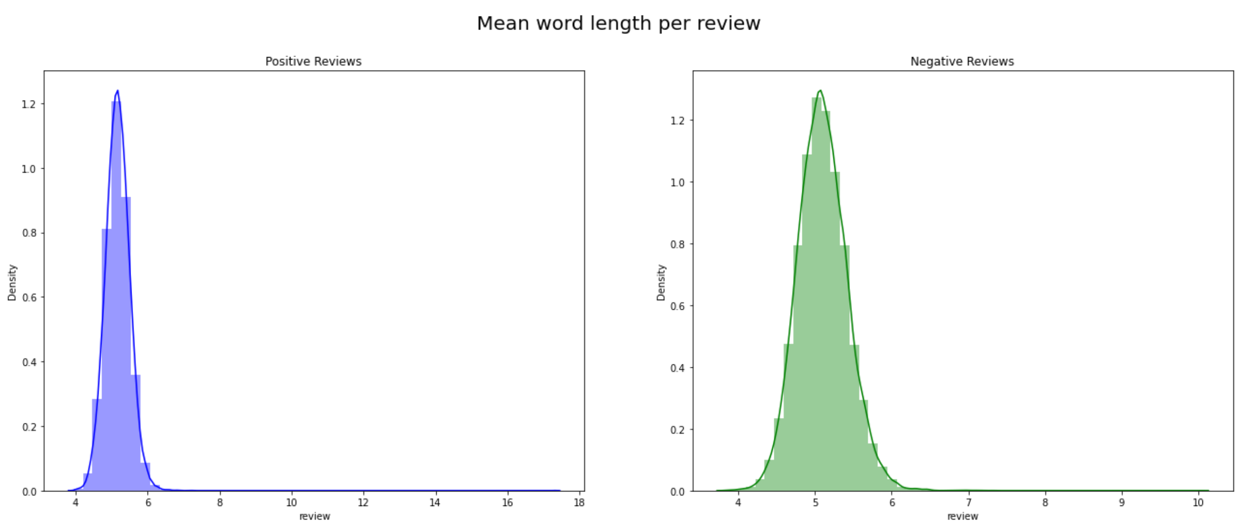
It seems that negative reviews are generally more verbose than positive reviews.

Mean word length per review

```
In [12]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(22, 8))
pos_word = df[df['sentiment'] == 'positive']['review'].str.split().apply(la
    len(i) for i in x])
sns.distplot(pos_word.map(lambda x: np.mean(x)), ax=ax1, color='blue')
ax1.set_title('Positive Reviews')
neg_word = df[df['sentiment'] == 'negative']['review'].str.split().apply(la
    len(i) for i in x])
sns.distplot(neg_word.map(lambda x: np.mean(x)), ax=ax2, color='green')
ax2.set_title('Negative Reviews')
fig.suptitle('Mean word length per review', fontsize=20)
```

```
/Users/dan/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/seabo
rn/distributions.py:2551: FutureWarning: `distplot` is a deprecated funct
ion and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) or
`histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
/Users/dan/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/seabo
rn/distributions.py:2551: FutureWarning: `distplot` is a deprecated funct
ion and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) or
`histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

```
Out[12]: Text(0.5, 0.98, 'Mean word length per review')
```



Word length in positive and negative reviews are relatively equal.

For further exploration, we must retrieve a corpus of all words:

Grab all words in reviews

In [13]:
```
def get_all_words(text):
    words = []
    for x in text:
        for y in x.split():
            words.append(y.strip())
    return words
all_words = get_all_words(df.review)
all_words[:10]
```

Out[13]:
```
['one',
 'review',
 'mention',
 'watch',
 'oz',
 'episod',
 'youll',
 'hook',
 'right',
 'exactli']
```

Count all words in reviews

In [14]:
```
from collections import Counter
counter = Counter(all_words)
most_common = counter.most_common(10)
most_common = dict(most_common)
most_common
```

Out[14]:
```
{'movi': 101385,
 'film': 94157,
 'one': 53941,
 'like': 44144,
 'time': 30810,
 'good': 29423,
 'make': 28675,
 'charact': 28038,
 'see': 27867,
 'get': 27820}
```

Create a function to find most common n-grams

credit: https://www.kaggle.com/madz2000/sentiment-analysis-cleaning-eda-bert-88-acc/notebook (https://www.kaggle.com/madz2000/sentiment-analysis-cleaning-eda-bert-88-acc/notebook)

In [15]:
```
def get_top_text_ngrams(text, num_words, ngram):
    vec = TfidfVectorizer(ngram_range=(ngram, ngram)).fit(text)
    bag_of_words = vec.transform(text)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx])
                  for word, idx in vec.vocabulary_.items()]
    words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)
    return words_freq[:num_words]
```

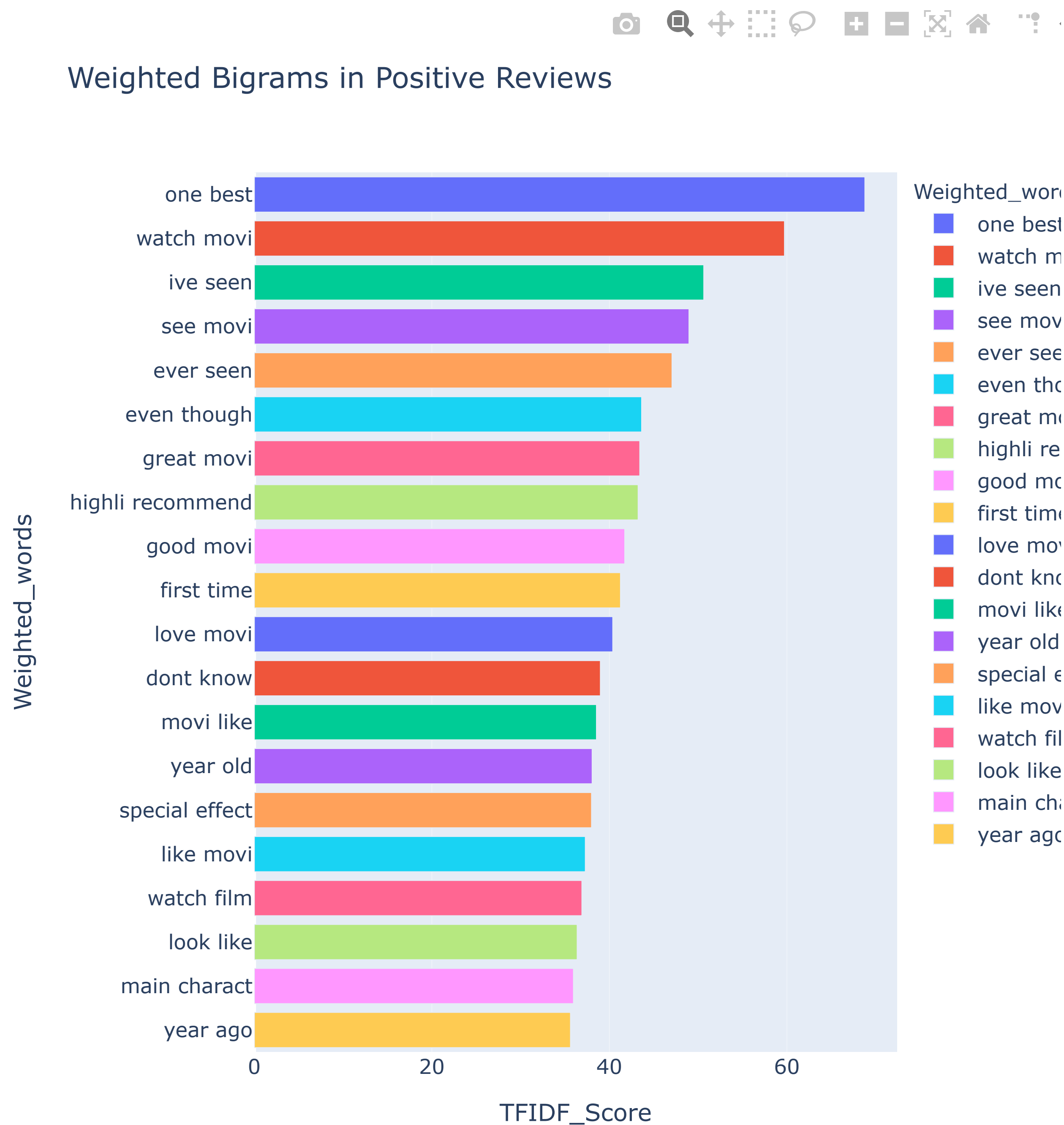
Unigram Analysis

In [16]:
```
most_common_uni_pos = get_top_text_ngrams(df.review[df.sentiment=='positive
most_common_uni_pos = dict(most_common_uni_pos)
temp = pd.DataFrame(columns=["Weighted_words", 'TFIDF_Score'])
temp["Weighted_words"] = list(most_common_uni_pos.keys())
temp["TFIDF_Score"] = list(most_common_uni_pos.values())
fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Wor
             width=700, height=700, color='Weighted_words')
fig.show()
```

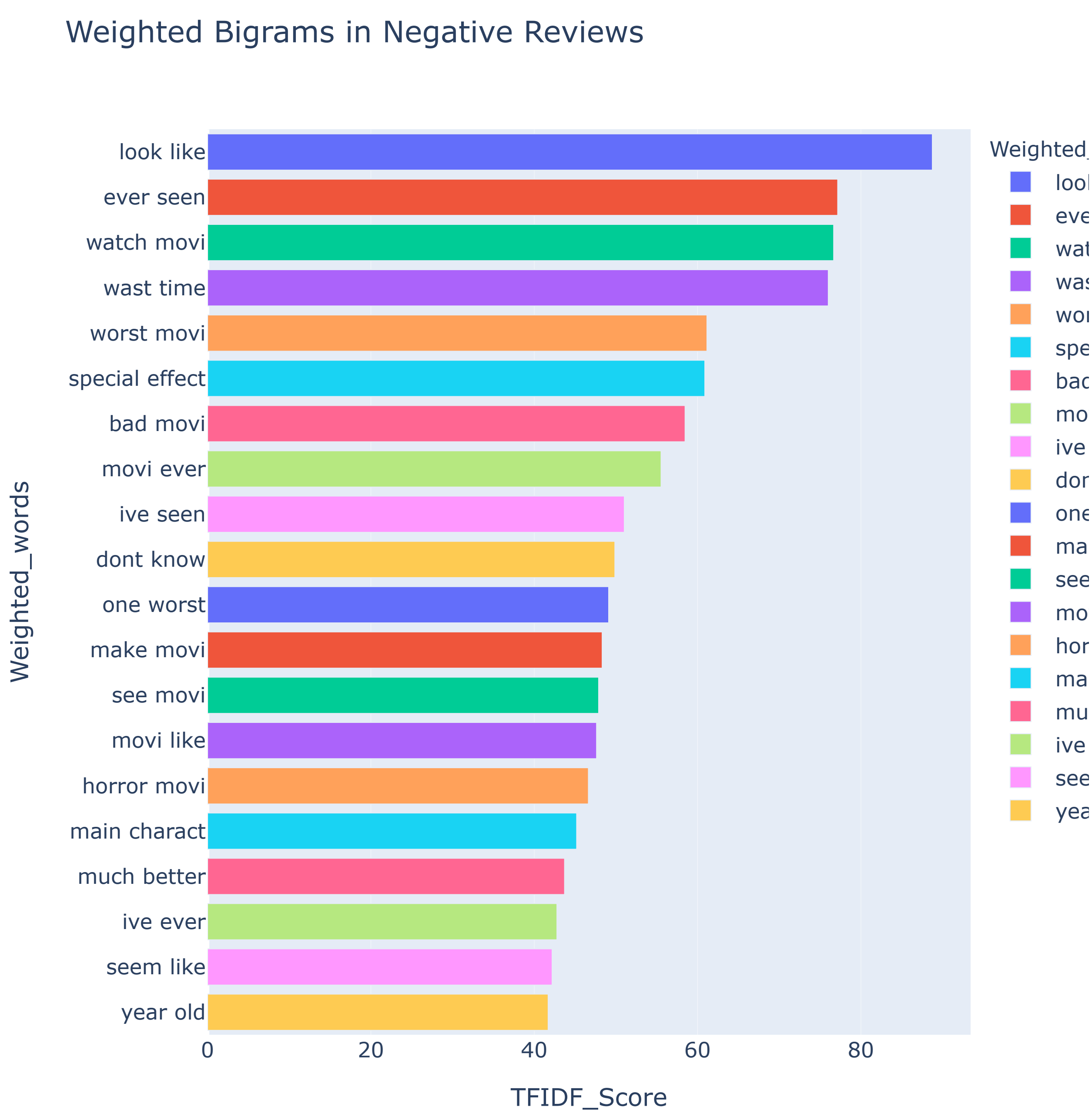
```
In [17]: most_common_uni_neg = get_top_text_ngrams(df.review[df.sentiment=='negative
         most_common_uni_neg = dict(most_common_uni_neg)
         temp = pd.DataFrame(columns=["Weighted_words", 'TFIDF_Score'])
         temp["Weighted_words"] = list(most_common_uni_neg.keys())
         temp["TFIDF_Score"] = list(most_common_uni_neg.values())
         fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Wor
                      width=700, height=700, color='Weighted_words')
         fig.show()
```

Bigram Analysis

In [18]:
```
most_common_bi_pos = get_top_text_ngrams(df.review[df.sentiment=='positive'
most_common_bi_pos = dict(most_common_bi_pos)
temp = pd.DataFrame(columns = ["Weighted_words" , 'TFIDF_Score'])
temp["Weighted_words"] = list(most_common_bi_pos.keys())
temp["TFIDF_Score"] = list(most_common_bi_pos.values())
fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Big
             width=700, height=700,color='Weighted_words')
fig.show()
```



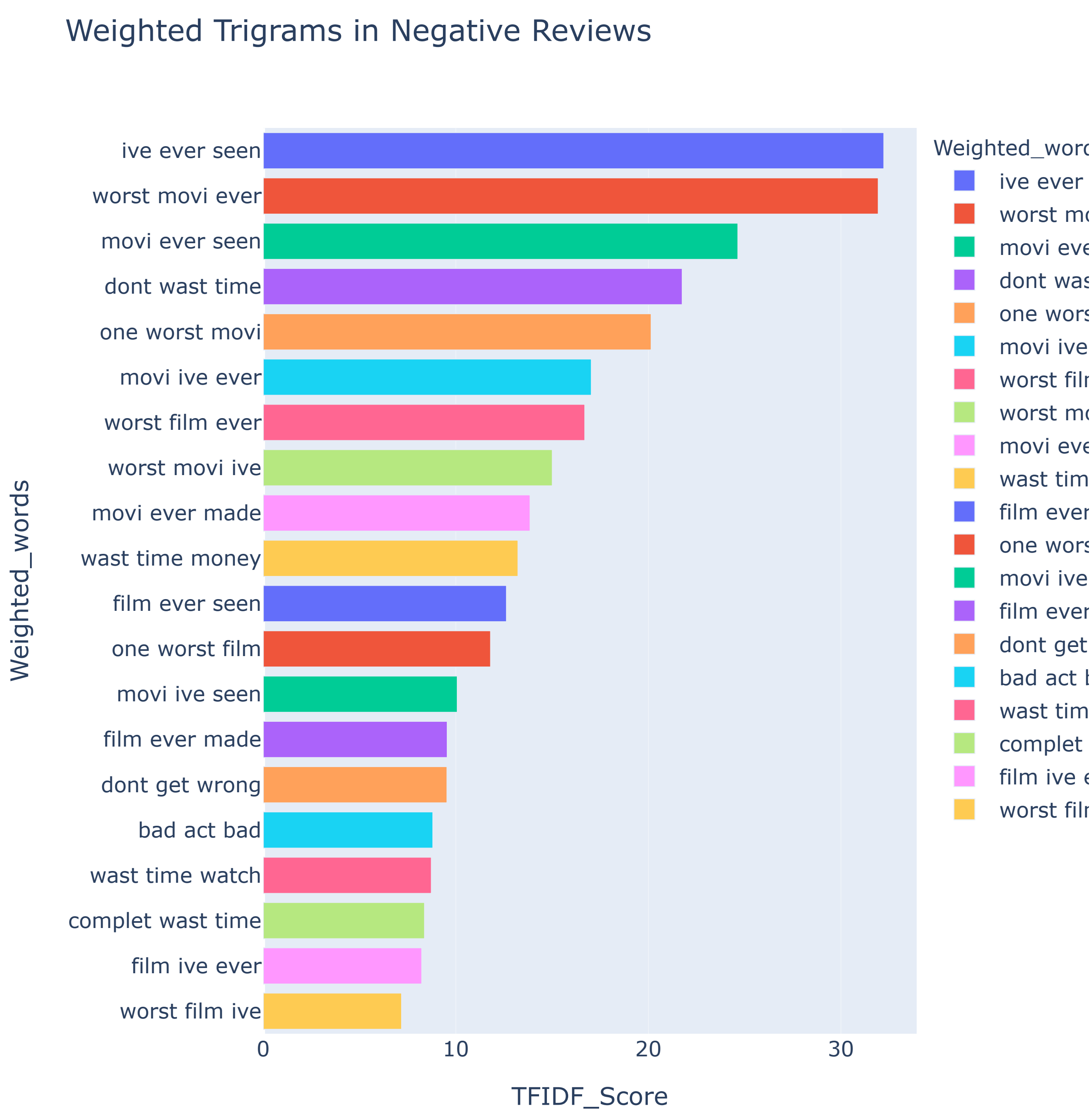
In [19]:
```
most_common_bi_neg = get_top_text_ngrams(df.review[df.sentiment=='negative'
most_common_bi_neg = dict(most_common_bi_neg)
temp = pd.DataFrame(columns = ["Weighted_words" , 'TFIDF_Score'])
temp["Weighted_words"] = list(most_common_bi_neg.keys())
temp["TFIDF_Score"] = list(most_common_bi_neg.values())
fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Big
             width=700, height=700,color='Weighted_words')
fig.show()
```



Trigram Analysis

In [20]:
```
most_common_tri_pos = get_top_text_ngrams(df.review[df.sentiment=='positive
most_common_tri_pos = dict(most_common_tri_pos)
temp = pd.DataFrame(columns = ["Weighted_words" , 'TFIDF_Score'])
temp["Weighted_words"] = list(most_common_tri_pos.keys())
temp["TFIDF_Score"] = list(most_common_tri_pos.values())
fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Tri
             width=700, height=700,color='Weighted_words')
fig.show()
```

In [21]:
```
most_common_tri_neg = get_top_text_ngrams(df.review[df.sentiment=='negative
most_common_tri_neg = dict(most_common_tri_neg)
temp = pd.DataFrame(columns = ["Weighted_words" , 'TFIDF_Score'])
temp["Weighted_words"] = list(most_common_tri_neg.keys())
temp["TFIDF_Score"] = list(most_common_tri_neg.values())
fig = px.bar(temp, x="TFIDF_Score", y="Weighted_words", title='Weighted Tri
             width=700, height=700,color='Weighted_words')
fig.show()
```



Modeling

Model-less Baseline

In [22]:
```
df['sentiment'].value_counts()
```

Out[22]:
```
positive    25000
negative    25000
Name: sentiment, dtype: int64
```

Our model-less baseline says we can accurately predict the each class 50% success rate if we only guess one class every time.

Encode Target Column

In [23]:
```
df.sentiment.replace("positive" , 1 , inplace = True)
df.sentiment.replace("negative" , 0 , inplace = True)
```

Train test split

In [24]:
```
X_train, X_test, y_train, y_test = train_test_split(df[['review']], df.sent
```

Initialize count and tfidf vectorizers

In [25]:
```
count = CountVectorizer()
tfidf = TfidfVectorizer(max_features=12500)
```

Create pipelines containing each vectorizer

In [26]:
```
models = {'lr_count': make_pipeline(count, LogisticRegression(max_iter=375,
          'dt_count': make_pipeline(count, DecisionTreeClassifier(random_st
          'rf_count': make_pipeline(count, RandomForestClassifier(random_st
          'lr_tfidf': make_pipeline(tfidf, LogisticRegression(max_iter=375,
          'dt_tfidf': make_pipeline(tfidf, DecisionTreeClassifier(random_st
          'rf_tfidf': make_pipeline(tfidf, RandomForestClassifier(random_st
```

In [59]:
```
models = {'lr_tfidf': make_pipeline(tfidf, LogisticRegression(max_iter=375,
```

We will prioritize F1 score in model selection for well balanced correctness

In [27]:
```
baseline_scores = {}

for model in models:
    score = cross_val_score(models[model], X_train.iloc[:,0], y_train, scor
    baseline_scores[model] = score.mean()

baseline_scores
```

Out[27]:
```
{'lr_count': 0.8756128986644571,
 'dt_count': 0.7171327201737887,
 'rf_count': 0.8513636098212689,
 'lr_tfidf': 0.8900022620962762,
 'dt_tfidf': 0.7106417503189231,
 'rf_tfidf': 0.8460468712274147}
```

Fit all training data on logistic regression model with tfidf vectorization

In [28]:
```
lr_pipeline = models['lr_tfidf']
```

In [29]:
```
lr_pipeline.fit(X_train.iloc[:,0], y_train)
```

Out[29]:
```
Pipeline(steps=[('tfidfvectorizer', TfidfVectorizer(max_features=12500)),
                ('logisticregression',
                 LogisticRegression(max_iter=375, random_state=42))])
```

Evaluate the model

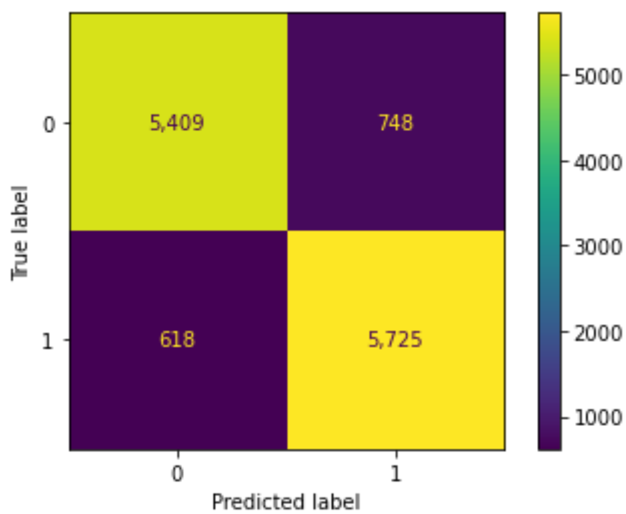
In [30]:

```python
def evaluate(estimator, X_train, X_test, y_train, y_test, roc_auc='proba'):
    '''
    Evaluation function to show a few scores for both the train and test se
    Also shows a confusion matrix for the test set

    roc_auc allows you to set how to calculate the roc_auc score:
    'dec' for decision_function or 'proba' for predict_proba
    If roc_auc == 'skip', then it ignores calculating the roc_auc_score

    Function takes in:
    'estimator' a fit sklearn model object
    'X_train' dataframe
    'X_test' dataframe
    'y_train' series
    'y_test' series
    'roc_auc' string that defines how the score is calculated
    '''
    # grab predictions
    train_preds = estimator.predict(X_train)
    test_preds = estimator.predict(X_test)

    # output needed for roc_auc_score
    if roc_auc == 'skip':  # skips calculating the roc_auc_score
        train_out = False
        test_out = False
    elif roc_auc == 'dec':  # not all classifiers have decision_function
        train_out = estimator.decision_function(X_train)
        test_out = estimator.decision_function(X_test)
    elif roc_auc == 'proba':
        train_out = estimator.predict_proba(
            X_train)[:, 1]  # proba for the 1 class
        test_out = estimator.predict_proba(X_test)[:, 1]
    else:
        raise Exception(
            "The value for roc_auc should be 'skip', 'dec' or 'proba'.")

    # print scores
    print("Train Scores")
    print("------------")
    print(f"Accuracy: {accuracy_score(y_train, train_preds)}")
    print(f"Precision: {precision_score(y_train, train_preds)}")
    print(f"Recall: {recall_score(y_train, train_preds)}")
    print(f"F1 Score: {f1_score(y_train, train_preds)}")
    if type(train_out) == np.ndarray:  # checking for roc_auc
        print(f"ROC-AUC: {roc_auc_score(y_train, train_out)}")
    print("----" * 5)
    print("Test Scores")
    print("-----------")
    print(f"Accuracy: {accuracy_score(y_test, test_preds)}")
    print(f"Precision: {precision_score(y_test, test_preds)}")
    print(f"Recall: {recall_score(y_test, test_preds)}")
    print(f"F1 Score: {f1_score(y_test, test_preds)}")
    if type(test_out) == np.ndarray:
        print(f"ROC-AUC: {roc_auc_score(y_test, test_out)}")

    # plot test confusion matrix
```

```
    plot_confusion_matrix(estimator, X_test, y_test, values_format=',.5g')
    plt.grid(False)
    plt.show()
```

In [31]:
```
evaluate(lr_pipeline, X_train.iloc[:,0], X_test.iloc[:,0], y_train, y_test,
```

```
Train Scores
------------
Accuracy: 0.9219733333333333
Precision: 0.9143444134225359
Recall: 0.9303210591199014
F1 Score: 0.9222635494155155
ROC-AUC: 0.9754639240177175
--------------------
Test Scores
-----------
Accuracy: 0.89072
Precision: 0.8844430712189093
Recall: 0.9025697619422985
F1 Score: 0.893414481897628
ROC-AUC: 0.9592380531179882
```



The model is slightly overfit, but overall performed well on all metric scores. Further optimizations can be made on:

1. Vectorizer 'max features' hyperparameter
2. Inverse of regularization strength 'C' hyperparameter

Grab Feature Importances

In [32]:
```
coefs = lr_pipeline.steps[1][1].coef_
```

Transform test data with fit tfidf vectorizer

In [33]:
```
# the fit tfidf vectorizer
transformer = lr_pipeline.steps[0][-1]
```

In [34]:
```
X_transformed = transformer.transform(X_test.iloc[:,0]).toarray()
```

Analyze coefficients for feature importance

Words in negative reviews

In [63]:
```
# Zip the names of the features with the features importance
coef_magnitudes_neg = zip(transformer.get_feature_names(), coefs.squeeze().
```

In [64]:
```python
# Sort the features in descending order by magnitude
top_100_neg = sorted(coef_magnitudes_neg, key=lambda x: x[1], reverse=False
top_100_neg
```

Out[64]:
```
[('worst', -9.979429095516444),
 ('wast', -8.897572147775568),
 ('aw', -7.69853494545529),
 ('bad', -7.499133681247096),
 ('bore', -6.970010664565677),
 ('poor', -6.241712171490241),
 ('terribl', -5.836886145433135),
 ('noth', -5.700127568440428),
 ('disappoint', -5.662935050120046),
 ('fail', -5.333368612472158),
 ('wors', -5.003899037007581),
 ('horribl', -4.909406708383349),
 ('poorli', -4.805561215512351),
 ('dull', -4.663559073580145),
 ('suppos', -4.587736246090588),
 ('lack', -4.520590601432766),
 ('save', -4.369268990324432),
 ('minut', -4.257373737396317),
 ('ridicul', -4.154887819324281),
 ('annoy', -4.1489427378412955),
 ('unfortun', -4.08775159102494),
 ('stupid', -3.9511499799907797),
 ('instead', -3.9476888858738),
 ('script', -3.8761877053583014),
 ('lame', -3.8244380780766156),
 ('mediocr', -3.59748632075463),
 ('mess', -3.5907856582381092),
 ('oh', -3.463797552047383),
 ('sorri', -3.4013464970450125),
 ('attempt', -3.374591976446817),
 ('avoid', -3.366955694201297),
 ('unless', -3.3534971942596212),
 ('would', -3.352055300496341),
 ('even', -3.3071164884309168),
 ('pointless', -3.2872214235273014),
 ('laughabl', -3.275749975751101),
 ('embarrass', -3.2472211299561224),
 ('mstk', -3.222147313971 2034),
 ('unfunni', -3.1689461333045124),
 ('pathet', -3.1635056839400915),
 ('forgett', -3.099705735731295),
 ('badli', -3.0794857972828695),
 ('couldnt', -3.058838375942007),
 ('clich', -3.041269085640917),
 ('weak', -2.9821310573662867),
 ('redeem', -2.9752543895109658),
 ('tri', -2.955249073424983),
 ('reason', -2.84835483761114),
 ('least', -2.824204264922108),
 ('money', -2.7921741545090266),
 ('cheap', -2.790914505548369),
 ('predict', -2.7530521488818906),
 ('pretenti', -2.728398775779035),
```

```
 ('idea', -2.721485270466022),
 ('neither', -2.717534644293169),
 ('look', -2.704203244427797),
 ('insult', -2.586591666586116),
 ('director', -2.5780439001501687),
 ('bother', -2.5576400708520146),
 ('stereotyp', -2.5435016446263132),
 ('tediou', -2.51282670236836),
 ('lousi', -2.510557932691127),
 ('suck', -2.5088161434253764),
 ('could', -2.5084057834982434),
 ('plot', -2.494387325283256),
 ('uninterest', -2.4794902451081144),
 ('none', -2.4653383319337916),
 ('garbag', -2.4601921530295576),
 ('wooden', -2.458294588474702),
 ('mildli', -2.4511246173785466),
 ('crap', -2.426146898260964),
 ('dread', -2.406355012024241),
 ('turkey', -2.3940695260134404),
 ('unintent', -2.379997239726048),
 ('far', -2.3770522403760572),
 ('seem', -2.37049640505965),
 ('potenti', -2.3633426716516737),
 ('silli', -2.316531737463489),
 ('miscast', -2.311969201564295),
 ('flat', -2.3030399678730293),
 ('problem', -2.2655435802052653),
 ('bland', -2.2425715312295553),
 ('enough', -2.236322351586425),
 ('pain', -2.192616361628102),
 ('materi', -2.190796043145468),
 ('stinker', -2.184538444692382),
 ('hour', -2.183738560768064),
 ('shallow', -2.171984850818519),
 ('nonsens', -2.165151288596038),
 ('seagal', -2.1449484825305825),
 ('uninspir', -2.139798404759646),
 ('irrit', -2.1378972603014796),
 ('effort', -2.136816009953017),
 ('amateurish', -2.131094478525395),
 ('dumb', -2.112446945273682),
 ('half', -2.0956442183228545),
 ('unbeliev', -2.0669049305979774),
 ('sit', -2.0597838653986424),
 ('write', -2.059154970182335),
 ('skip', -2.0486711796827706)]
```

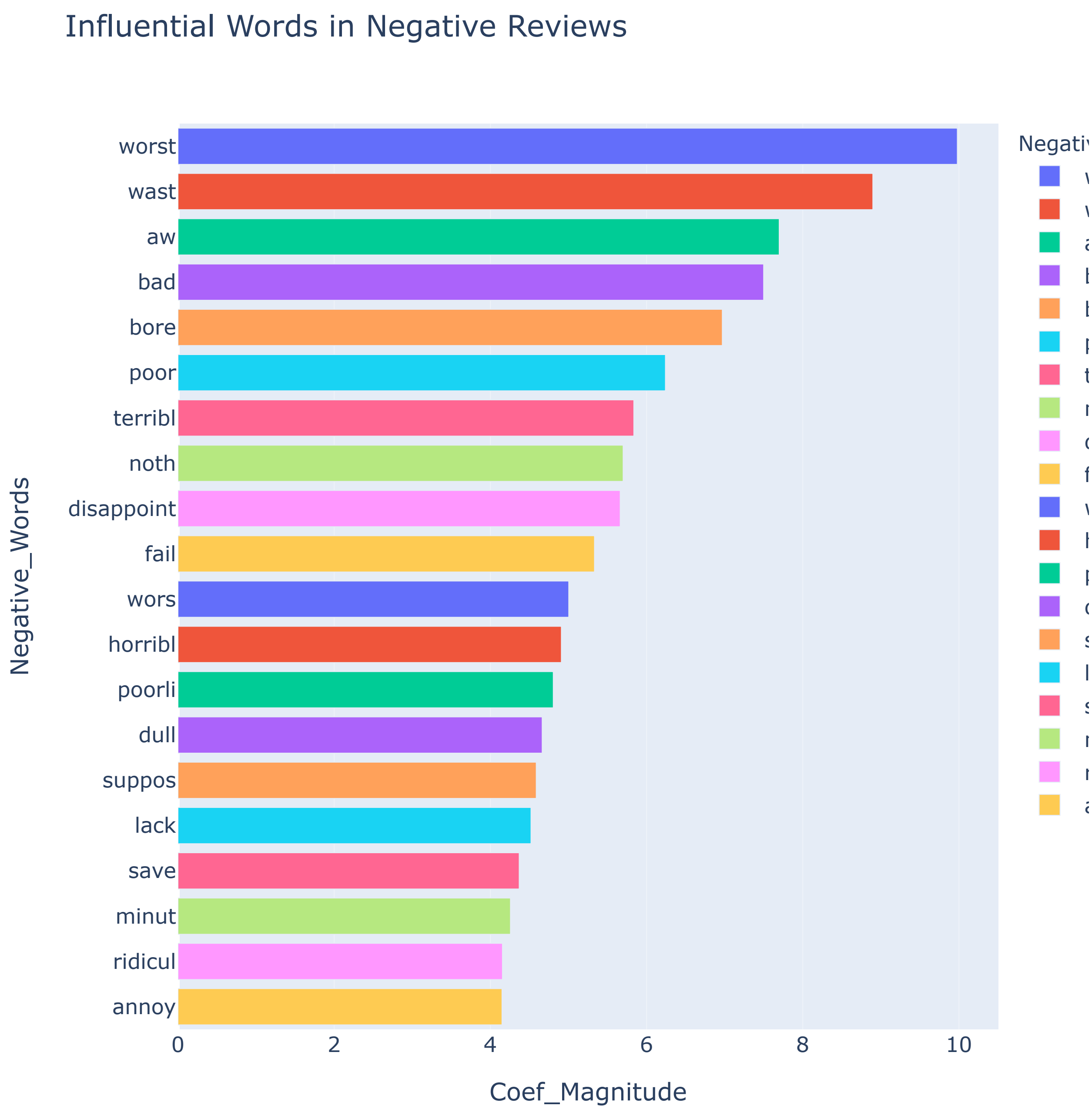
In [68]:

```
coef_magnitudes_neg = zip(transformer.get_feature_names(), coefs.squeeze().
neg_sorted = sorted(coef_magnitudes_neg, key=lambda x: x[1], reverse=False)
```

In [69]:
```
neg_sorted
```

Out[69]:
```
[('worst', -9.979429095516444),
 ('wast', -8.897572147775568),
 ('aw', -7.69853494545529),
 ('bad', -7.499133681247096),
 ('bore', -6.970010664565677),
 ('poor', -6.241712171490241),
 ('terribl', -5.836886145433135),
 ('noth', -5.700127568440428),
 ('disappoint', -5.662935050120046),
 ('fail', -5.333368612472158),
 ('wors', -5.003899037007581),
 ('horribl', -4.909406708383349),
 ('poorli', -4.805561215512351),
 ('dull', -4.663559073580145),
 ('suppos', -4.587736246090588),
 ('lack', -4.520590601432766),
 ('save', -4.369268990324432),
 ('minut', -4.25737737396317),
 ('ridicul', -4.154887819324281),
 ('annoy', -4.1489427378412955)]
```

In [70]:
```
temp = pd.DataFrame(neg_sorted, columns = ["Negative_Words", 'Coef_Magnitud
temp['Coef_Magnitude'] = temp['Coef_Magnitude'].abs() # Apply absolute valu
fig = px.bar(temp, x="Coef_Magnitude", y="Negative_Words", title='Influenti
             width=700, height=700,color='Negative_Words')
fig.show()
```



Words in positive reviews

In [71]:

```
# Zip the names of the features with the features importance
coef_magnitudes_pos = zip(transformer.get_feature_names(), coefs.squeeze().
```

In [72]:
```python
# Sort the features in descending order by magnitude
top_100_pos = sorted(coef_magnitudes_pos, key=lambda x: x[1], reverse=True)
top_100_pos
```

Out[72]:
```
[('great', 7.333116055961179),
 ('excel', 7.0107668593828425),
 ('enjoy', 5.461692705850696),
 ('perfect', 5.453807875926427),
 ('love', 5.004253278488765),
 ('best', 4.910473880629905 5),
 ('brilliant', 4.852488902344012),
 ('favorit', 4.62837021507443),
 ('amaz', 4.53613492456775),
 ('hilari', 4.212978064617924),
 ('beauti', 4.029569690260818),
 ('today', 3.954248738102151),
 ('highli', 3.913240544320295),
 ('fun', 3.8195655684280467),
 ('definit', 3.8006248798368327),
 ('fantast', 3.759831580093193),
 ('well', 3.683343325715259),
 ('superb', 3.6393337837101436),
 ('entertain', 3.493478317205275),
 ('surpris', 3.469791543086244),
 ('perfectli', 3.391571581526942),
 ('still', 3.3670483766058434),
 ('funniest', 3.329525911624191),
 ('recommend', 3.2406623820935834),
 ('refresh', 3.2083419246120535),
 ('touch', 3.184655907131137),
 ('uniqu', 3.133547532597266),
 ('gem', 3.0735696192589246),
 ('strong', 3.017431167622673),
 ('classic', 2.9872924895192656),
 ('realist', 2.9305204918788785),
 ('awesom', 2.9076940365129396),
 ('simpl', 2.8838150794414887),
 ('especi', 2.8772420956042963),
 ('subtl', 2.8668667522935767),
 ('appreci', 2.7682067093353506),
 ('terrif', 2.722485042419419),
 ('bit', 2.7164610891540124),
 ('good', 2.715784487385702),
 ('greatest', 2.7109445329761157),
 ('alway', 2.6913266144408308),
 ('dvd', 2.656792842075479),
 ('delight', 2.59290663986185),
 ('fascin', 2.5900236426574557),
 ('also', 2.5511719987930284),
 ('thank', 2.546770277237754),
 ('underr', 2.517240014760455),
 ('see', 2.4906795214130297),
 ('masterpiec', 2.4781289092552186),
 ('marvel', 2.4595572614358368),
 ('solid', 2.452166259474853),
 ('wonder', 2.3941688819612827),
 ('human', 2.363460336292522),
```

```
 ('outstand', 2.3477891531399773),
 ('job', 2.3269732291877188),
 ('world', 2.295604701918964),
 ('atmospher', 2.2719403353736127),
 ('perform', 2.246272594777399),
 ('differ', 2.2292345651803553),
 ('surprisingli', 2.219885415377982),
 ('life', 2.138420142715407),
 ('everyon', 2.126899513103413),
 ('rare', 2.1141027828008863),
 ('memor', 2.1114262183989707),
 ('finest', 2.10269320462038),
 ('brilliantli', 2.096652945797733),
 ('favourit', 2.057850841309165),
 ('intens', 1.9995002200892762),
 ('lot', 1.9834016718375036),
 ('power', 1.9446961329513008),
 ('nevertheless', 1.9409529880737946),
 ('nice', 1.918148352743646),
 ('true', 1.913120013881596),
 ('complaint', 1.9122732880137063),
 ('innoc', 1.9076845648885745),
 ('sweet', 1.8941332616901039),
 ('tear', 1.8904384632132085),
 ('unexpect', 1.8787664840084743),
 ('emot', 1.8569343099612954),
 ('time', 1.8389073829044953),
 ('chanc', 1.837408749734956),
 ('heart', 1.8255005524932897),
 ('noir', 1.8151173261403368),
 ('seen', 1.802153148541798),
 ('keep', 1.778642165417919),
 ('delici', 1.7765334642567352),
 ('tale', 1.7432399670081011),
 ('glad', 1.7359438291401044),
 ('edg', 1.7353695960057385),
 ('stun', 1.7349184092432715),
 ('think', 1.7048208048559648),
 ('chill', 1.6873663991020564),
 ('dream', 1.6831148441257804),
 ('sometim', 1.6803712939956506),
 ('journey', 1.672462859452585),
 ('incred', 1.6639426332867486),
 ('deal', 1.6618218604198307),
 ('extraordinari', 1.6598446237628557),
 ('seat', 1.659237111281219),
 ('pleasantli', 1.6417542553999815)]
```

In [73]:
```
coef_magnitudes_pos = zip(transformer.get_feature_names(), coefs.squeeze().
pos_sorted = sorted(coef_magnitudes_pos, key=lambda x: x[1], reverse=True)[
```

In [74]:
```
temp = pd.DataFrame(pos_sorted, columns = ["Positive_Words", 'Coef_Magnitud
fig = px.bar(temp, x="Coef_Magnitude", y="Positive_Words", title='Influenti
             width=700, height=700,color='Positive_Words')
fig.show()
```

Evaluation

Our originally stated hypothesis was:

'A model can be derived to input opinionated language material written about a film or show, so that it can reliably predict positive or negative sentiment.'

After text preprocessing and model selection, the resulting findings are:

1. We can reliably predict movie/show sentiment at a rate of 89.3%. This is significantly better than our model-less baseline of 50%.
2. We have identified an abundance of words with the most significant predictive power from our dataset of 50,000 reviews. With testing, it will be interesting to see if these feature importances are applicable to review material outside of IMDB's database.

Next Steps

As mentioned above, we may be able to tune for better performance by optimizing the following hyperparameters:

1. Vectorizer 'max features' hyperparameter
2. Inverse of regularization strength 'C' hyperparameter

In [ ]:

```

```