Predicting Brand Sentiment on Twitter

- Nick Gigliotti
- ndgigliotti@gmail.com

Business Problem

Apple has asked me to create a strong predictive model for detecting positive, negative, and neutral sentiment in tweets. They are primarily concerned with tweets about their company and products, but also might want to know what people are saying about competitors. They intend to use the model to classify new, never-before-seen, tweets, in order to conduct their research. My goals are:

1. Create an accurate classifier which can classify **novel tweets** as positive, negative, or neutral.
2. Find out what people are saying about Apple (at South by Southwest, 2011).
3. Make some PR recommendations for the period immediately following the event.

Imports

Because there are so many of them, I've created a separate section.

Standard Library and External

In [1]:

```
import re
import string
import json
from pprint import pprint
from functools import partial
from operator import itemgetter, attrgetter
from os.path import normpath
from typing import Callable

import joblib
import matplotlib.pyplot as plt
import nltk
import numpy as np
import pandas as pd
import seaborn as sns
from gensim.parsing.preprocessing import STOPWORDS
from sacremoses import MosesTokenizer, MosesTruecaser
from sklearn.base import clone
from sklearn.compose import (
    ColumnTransformer,
    make_column_selector,
    make_column_transformer,
)
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import (
    VarianceThreshold,
    SelectKBest,
    SelectPercentile,
    GenericUnivariateSelect,
)
```

```python
from sklearn.ensemble import StackingClassifier, VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import (
    LogisticRegression,
    LogisticRegressionCV,
    PassiveAggressiveClassifier,
    RidgeClassifier,
    RidgeClassifierCV,
    SGDClassifier,
)
from sklearn.naive_bayes import (
    BernoulliNB,
    CategoricalNB,
    ComplementNB,
    GaussianNB,
    MultinomialNB,
)
from sklearn.svm import LinearSVC, NuSVC, OneClassSVM, SVC
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    RepeatedStratifiedKFold,
    StratifiedKFold,
    train_test_split,
)
from sklearn.pipeline import FeatureUnion, Pipeline, make_pipeline
from sklearn.preprocessing import (
    Binarizer,
    FunctionTransformer,
    MaxAbsScaler,
    MinMaxScaler,
    Normalizer,
    PowerTransformer,
    QuantileTransformer,
    RobustScaler,
    StandardScaler,
    PolynomialFeatures,
)

# Set Seaborn theme and default palette
sns.set_theme(font_scale=1.25, style="darkgrid")
sns.set_palette("deep", desat=0.85, color_codes=True)

# Turn on inline plotting
%matplotlib inline

# Load Black auto-formatter
%load_ext nb_black

# Enable automatic reloading
%load_ext autoreload
%autoreload 2
```

My `tools` Package

I put a lot of time and energy into developing my own tools for analysis. It's probably my favorite part of this kind of work, and I (admittedly) tend to get carried away with it. I developed a lot in `tools.language`, `tools.sklearn.vectorizers`, and `tools.sklearn.selection` for this project in particular.

Caching

Some computationally expensive functions in `tools.language` implement caching, allowing them to save the results of previous calls and reuse them. This **dramatically increases their performance** when being called over and over again as part of a preprocessing pipeline. Essentially, after the function has been called once with certain parameters, every subsequent call with those parameters is fulfilled instantly. This is a highly non-trivial development, which increases the speed of parameter searches (e.g. with `GridSearchCV`) and makes model development more efficient in general.

Polymorphism

I've designed the raw-text processing functions in `tools.language` to be polymorphic: capable of handling both a single string document and various types of iterables of documents. This level of flexibility is arguably overkill for the present task.

FreqVectorizer

I extended Scikit-Learn's `TfidfVectorizer` to be capable of much more advanced preprocessing out of the box. In addition to the many new text filters, there's built-in stemming and lemmatization, better stopwords selection, and the option to mark negation or parts of speech. See My `FreqVectorizer` and what comes after for more details.

VaderVectorizer

Another notable development is the `VaderVectorizer`, which extracts VADER (Valence Aware Dictionary and Sentiment Reasoner) polarity scores from documents and turns them into short vectors of shape (n_samples, 4). This is essentially just a fancy wrapper around the VADER tools from NLTK, which integrates them with the Scikit-Learn API and implements caching. It proved very useful for the current project. See Add `VaderVectorizer` and what follows.

See also:

My `sweep` Function for my generic parameter-space searching function.

In [2]:

```
# Import my modules
from tools import cleaning, plotting, language as lang, utils
from tools.sklearn.vectorizers import FreqVectorizer, VaderVectorizer
from tools.sklearn.classification import diagnostics as diag
from tools.sklearn import selection

# Run time-consuming grid searches
RUN_SWEEPS = False

# Set my default MPL settings
plt.rcParams.update(plotting.MPL_DEFAULTS)

# RandomState for reproducibility
rando = np.random.RandomState(9547)
```

Overview of Dataset

Since Apple is interested in sentiment analysis on Twitter, I've found a Twitter dataset with crowdsourced sentiment labels. It comes from CrowdFlower, which has released other similar datasets.

The tweets are related to South by Southwest, an annual conference and arts festival in Austin, Texas. They are from 2011, when Apple launched the iPad 2.

It has only three features: the tweet text, the brand object of the sentiment, and the sentiment. It has only about 9,100 tweets.

In [3]:
```
df = pd.read_csv(normpath("data/crowdflower_tweets.csv"))
df.head()
```

Out[3]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or_product
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative emotion
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive emotion
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive emotion
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative emotion
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive emotion

The dataset contains one text feature and two categorical features, one of which has a lot of null values. The feature names are very long and wordy, presumably to reflect the actual language used by CrowdFlower in crowdsourcing the dataset. I'm going to rename those before I do anything else.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9093 entries, 0 to 9092
Data columns (total 3 columns):
 #   Column                                              Non-Null Count  Dtype 
---  ------                                              --------------  ----- 
 0   tweet_text                                          9092 non-null   object
 1   emotion_in_tweet_is_directed_at                     3291 non-null   object
 2   is_there_an_emotion_directed_at_a_brand_or_product  9093 non-null   object
dtypes: object(3)
memory usage: 213.2+ KB
```

Cleaning

Renaming

In [5]:
```
# Assign new column names
df.columns = ["text", "object_of_emotion", "emotion"]
df.head()
```

Out[5]:

	text	object_of_emotion	emotion
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative emotion
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive emotion
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive emotion
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative emotion
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive emotion

Next, I take a look at the values of the categorical variables. The categories make sense, although the names are longer than necessary. I'm going to shorten some of them as well.

In [6]:

```
cleaning.show_uniques(df)
```

object_of_emotion	emotion
iPhone	Negative emotion
iPad or iPhone App	Positive emotion
iPad	No emotion toward brand or product
Google	I can't tell
Android	
Apple	
Android App	
Other Google product or service	
Other Apple product or service	

First, I convert the categorical columns to `CategoricalDtype`. This will make it easier to rename the categories, and is a convenient way to differentiate the categorical features from the text column.

In [7]:

```
# Convert categorical columns to categorical dtype
cat_cols = ["emotion", "object_of_emotion"]
df[cat_cols] = df.loc[:, cat_cols].astype("category")

# Delete temp variable
del cat_cols

# Display results
display(df["emotion"].head(3), df["object_of_emotion"].head(3))
```

```
0    Negative emotion
1    Positive emotion
2    Positive emotion
Name: emotion, dtype: category
Categories (4, object): ['I can't tell', 'Negative emotion', 'No emotion toward brand or product',
'Positive emotion']
0              iPhone
1    iPad or iPhone App
2                iPad
Name: object_of_emotion, dtype: category
Categories (9, object): ['Android', 'Android App', 'Apple', 'Google', ..., 'Other Google product or
service', 'iPad', 'iPad or iPhone App', 'iPhone']
```

Next, I rename the categories for both categorical features.

I use a single `dict` mapping old category names to new ones. I only need one `dict` for both features because the method `Series.cat.rename_categories(...)` ignores irrelevant keys.

In [8]:

```
# Create mapping of old categories to new ones
new_cats = {
    # New 'emotion' categories
    "Negative emotion": "Negative",
    "Positive emotion": "Positive",
    "No emotion toward brand or product": "Neutral",
    "I can't tell": "Uncertain",
    # New 'object_of_emotion' categories
```

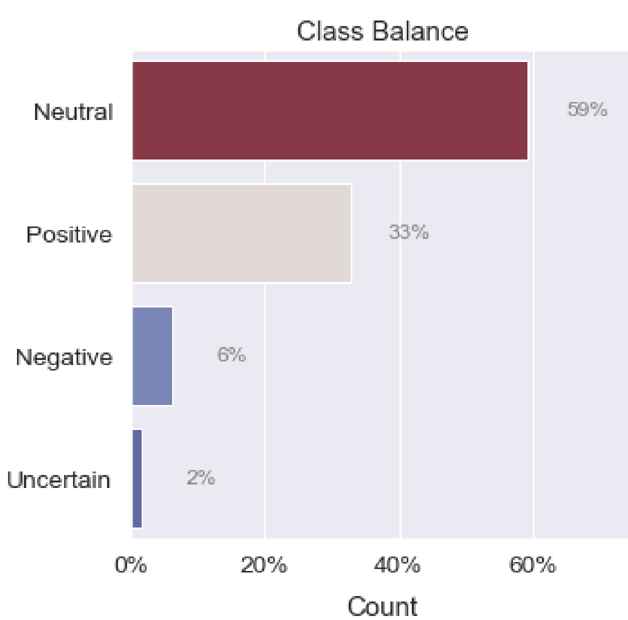
```
    "iPad or iPhone App": "iOS App",
    "Other Google product or service": "Other Google Product",
    "Other Apple product or service": "Other Apple Product",
}

# Rename categories in-place (ignores irrelevant keys)
df["emotion"].cat.rename_categories(new_cats, inplace=True)
df["object_of_emotion"].cat.rename_categories(new_cats, inplace=True)

# Delete renaming dict
del new_cats

# Show results
cleaning.show_uniques(df)
```

object_of_emotion	emotion
iPhone	Negative
iOS App	Positive
iPad	Neutral
Google	Uncertain
Android	
Apple	
Android App	
Other Google Product	
Other Apple Product	

The 'Neutral' category dominates the distribution, and 'Negative' is very underrepresented. 'Uncertain' is fortunately a very small 2% of the samples. That's good, because it's completely useless to me.

In [9]:
```
ax = plotting.countplot(df["emotion"], normalize=True)
ax.set(title="Class Balance")
ax.set_xlim((0, 0.75))
```

Out[9]: (0.0, 0.75)



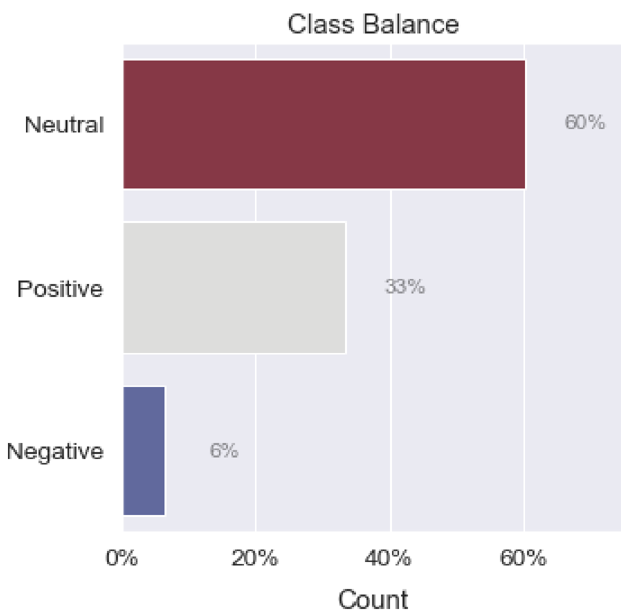
I drop the uncertain category, which doesn't have any clear value. I will have to cope with this imbalance later.

In [10]:

```
# Remove 'Uncertain' category
df.emotion.cat.remove_categories("Uncertain", inplace=True)

# Plot class balance
ax = plotting.countplot(df.emotion, normalize=True)
ax.set(title="Class Balance")
ax.set_xlim((0, 0.75))
plotting.save(ax.figure, "images/class_balance.svg")
```

Out[10]:

```
'images\\class_balance.svg'
```



Missing Values

According to the table below, there are a lot of missing values in the 'object_of_emotion' category. I bet, however, that these NaN values correspond to the 'Neutral' category. If a tweet doesn't express a brand-emotion, then there shouldn't be any brand in the 'object_of_emotion' column.

There's also one null 'text' row, and a bunch of null 'emotion' rows where the 'Uncertain' category used to be.

In [11]:

```
cleaning.info(df)
```

Out[11]:

	null	null_%	uniq	uniq_%	dup	dup_%
object_of_emotion	5802	63.81	9	0.10	22	0.24
emotion	156	1.72	3	0.03	22	0.24
text	1	0.01	9065	99.69	22	0.24

I'll go ahead and drop the nulls in the 'text' and 'emotion' columns first.

In [12]:

```
df.dropna(subset=["text", "emotion"], inplace=True)
cleaning.info(df)
```

Out[12]:

	null	null_%	uniq	uniq_%	dup	dup_%

	null	null_%	uniq	uniq_%	dup	dup_%
object_of_emotion	5654	63.27	9	0.10	22	0.25
text	0	0.00	8909	99.70	22	0.25
emotion	0	0.00	3	0.03	22	0.25

In [13]:

```
null_rows = cleaning.null_rows(df)
lang.readable_sample(null_rows["text"], random_state=rando)
```

	text
5140	RT @mention @mention New iPad Apps For Speech Therapy And Communication Are Showcased At #SXSW Conference {link} #sxswi #hcsm #sxswh
509	Please RT Follow the next big #college social network @mention chance to win an #iPad at 7,000 followers #socialmedia #SXSW
4916	millions of iPhone cases at #SXSW trade show but can any of them double as shuffleboard wax sprinklers? I think not. #fail (CC @mention
6384	RT @mention not launching any products at #SXSW but we're doing plenty else. {link}
790	Google to Launch Major New Social Network Called Circles, Possibly Today {link} #sxsw"
8793	Google giving Social another go? {link} Google Circles, let's see what the guys at #SXSW make of it
8452	@mention The unofficial #SXSW torrents are a great way to hear what you can expect this year {link}
3645	U gotta fight for yr right to party & to privacy ACLU/google #sxsw #partylikeits1986
61	#futuremf @mention {link} spec for recipes on the web, now in google search: {link} #sxsw
4081	Hope people ask the tough questions. RT @mention Reminder: Android and Chrome TTS talk @mention 1 PM today! {link} #sxsw

Looks like some of the NaN values don't line up with the 'Neutral' category. Also, it's important to note that some retweets, e.g. 64, 68, do have sentimental content beyond that of the original tweet.

In [14]:

```
emotion_without_object = null_rows.loc[null_rows.emotion != "Neutral"]

# Delete variable
del null_rows

display(emotion_without_object.head(), emotion_without_object.shape)
```

	text	object_of_emotion	emotion
46	Hand-Held ���Hobo�; Drafthouse launches ���Ho...	NaN	Positive
64	Again? RT @mention Line at the Apple store is ...	NaN	Negative
68	Boooo! RT @mention Flipboard is developing an ...	NaN	Negative
103	Know that "dataviz" translates to &q...	NaN	Negative
112	Spark for #android is up for a #teamandroid aw...	NaN	Positive

```
(357, 3)
```

These are positive tweets which are missing a brand label. Many of them seem positive, some towards a brand and some not. The original features names were 'emotion_in_tweet_is_directed_at' and 'is_there_an_emotion_directed_at_a_brand_or_product', which is not consistent with brandless positivity. But this is data science, and in data science, nothing is consistent.

In [15]:

```
lang.readable_sample(
    emotion_without_object.groupby("emotion").get_group("Positive").text,
    random_state=456,
)
```

	text
6606	RT @mention RT @mention Shiny new @mention @mention @eightbit apps, a new @garyvee book, pop-up iPad 2 stores... #SXSW is Christmas for nerds.
4164	Mad long line for Google party at Maggie Mae's. Hope it's worth it.. but with 80s theme I am very optimistic #sxsw
3020	Apple offers original iPad donation program {link} #entry #friends #house #sxsw
8114	#touchingstories giving us the background to STARTING. Great to hear after yesterday's presos on #uncertainty #iPad and/or #tablet #SXSW
555	I have my golden tickets f 4sq party Day after the real party #Redbullbpm with Felix da Housecat playing on iPad! #SXSW {link}
5501	RT @mention At #sxsw even the cabbies are tech savvy. That's his iPhone streaming twitter. @mention {link}
6676	RT @mention Soundtrckr featured by @mention @mention as a Must-have for #SXSW {link}
157	@mention #SXSW LonelyPlanet Austin guide for #iPhone is free for a limited time {link} #lp #travel
5019	Here he comes ladies! @mention @mention RT @mention I'll be at Austin Convention Center w/ @mention showing my iPhone game. #SXSW
8025	Someone asks Leo about an iPad 2 at #SXSW, he says 'Email me, I'll send you one free'. O.o

Fortunately there aren't very many of them, so not much hangs on my decision to go ahead and fill in the missing brands.

In [16]:

```
# Create regex for finding each brand
re_apple = r"ipad\s*\d?\s*app|ipad|iphone\s*\d?\s*app|iphone|apple"
re_google = r"android\s*app|android|google"

# Find brand/product name occurrences for each brand
findings = lang.locate_patterns(
    re_apple,
    re_google,
    strings=emotion_without_object["text"],
    exclusive=True,
    flags=re.I,
)

# Convert to lowercase
findings = findings.str.lower()

# View results
display(
    findings.value_counts(),
    findings.size,
)
```

```
ipad          125
google        122
apple          76
iphone         57
android        19
iphone app      8
ipad app        4
```

```
android app      1
Name: locate_patterns, dtype: int64
412
```

In [17]:

```
# Rename Apple apps to match categories defined previously
findings = findings.str.replace(
    r"ipad\s+app|iphone\s+app", "ios app", case=False, regex=True
)

# Fuzzy match with previously defined categories
findings = lang.fuzzy_match(findings, df["object_of_emotion"].cat.categories)

# View results
findings.sort_values("score")
```

Out[17]:

	original	match	score
46	ios app	iOS App	100
6220	iphone	iPhone	100
6202	iphone	iPhone	100
6180	apple	Apple	100
6180	ipad	iPad	100
...	...	...	...
3055	ipad	iPad	100
3055	ipad	iPad	100
3040	ipad	iPad	100
3269	android	Android	100
9054	ipad	iPad	100

412 rows × 3 columns

In [18]:

```
# Define sort order, i.e. fill priority
order = [
    "iOS App",
    "Android App",
    "iPhone",
    "iPad",
    "Android",
    "Apple",
    "Google",
]

# Sort values in reverse order
utils.explicit_sort(
    findings,
    order=order,
    by="match",
    ascending=False,
    inplace=True,
)

# Fill in reverse, overwriting lower priority values
for i, brand in findings.match.items():
    df.at[i, "object_of_emotion"] = brand
df.loc[findings.index].sample(10, random_state=rando)
```

Out[18]:

	text	object_of_emotion	emotion
8029	Yeah I wasn't doing it, but I got couldn't res...	iPad	Positive
2753	I love the waves!!!!!! {link} iPad Webber #jap...	iPad	Positive
8973	Google guy at #sxsw talk is explaining how he ...	Google	Negative
1089	���@mention So @mention just spilled the beans...	iPhone	Positive
4674	Apple opening up temporary store in downtown A...	iPad	Positive
4536	Whoa - line for ipad2 is 3blks long!!! #apple ...	iPad	Positive
6078	RT @mention I'm debuting my new iPhone & D...	iPhone	Positive
6710	RT @mention Temporary #apple store is def not ...	Apple	Positive
682	#technews iPad 2 Gets Temporary Apple Store fo...	iPad	Positive
5501	RT @mention At #sxsw even the cabbies are tech...	iPhone	Positive

In [19]:

```
# Get indices which were not filled
emotion_without_object.drop(findings.index, inplace=True)

# Drop unfilled observations
df.drop(emotion_without_object.index, inplace=True)

print(f"{emotion_without_object.shape[0]} observations dropped.")

del emotion_without_object
```

```
24 observations dropped.
```

Here are the tweets which are labeled 'Neutral' but have a brand label, implying that a non-neutral emotion is being expressed towards a brand. Most 'Neutral' tweets do not have a brand label, so these 91 tweets are an anomaly.

In [20]:

```
object_without_emotion = df.loc[
    (df.emotion == "Neutral") & df.object_of_emotion.notnull()
]
display(object_without_emotion.head(), object_without_emotion.shape)
```

	text	object_of_emotion	emotion
63	#Smile RT @mention I think Apple's "pop-u...	Apple	Neutral
265	The #SXSW Apple "pop-up" store was n...	Apple	Neutral
317	I arrived at #sxsw and my @mention issue hasn'...	iOS App	Neutral
558	haha. the google "Party like it's 1986&qu...	Google	Neutral
588	Diller on Google TV: "The first product w...	Other Google Product	Neutral

```
(91, 3)
```

Tweet 6517 seems clearly negative to me, and 7137 seems kind of sardonic. 2666 seems weakly positive. 8647, 5696, 7521, 668, and 265 don't seem to express an emotion toward a brand or product. Since most of them seem neutral to me, and that's consistent with their 'Neutral' label, I'm going to keep them that way.

In [21]:

```
lang.readable_sample(object_without_emotion["text"], random_state=rando)
```

	text
668	#sxsw guy in front of me at this panel has an ipad in an etch-a-sketch case...device of wondeR? #iusxsw
1628	@mention @mention Similarily, Tweetcaster for Android lets you zip tweets w annoying hash tags, like #sxsw
1253	Google vp to speak. The topic: 10 quick steps to owning everything in the world. #sxsw {link}
2849	Nice to see the speaker sneak in an irrelevant snarky comment about Apple. Class! #sxsw #authenticationdesign
7658	Score a free imo tshirt outside the SXSW Apple store today at 2:15 PM & check out imo's app for the iPad 2 {link} #sxsw #ipad2
4119	From #Apple to Naomi Campbell: pop-up stores are all the rage: {link} #sxsw
5912	RT @mention Google to launch new social network at SXSW? - CNET News {link} #sxsw
6082	RT @mention I'm not really at #sxsw. Just messing with you. I'm making money instead. // I bet someone left the iPad queue
6491	RT @mention RT @mention "IAVA wants to be the Google of nonprofits." / yes, we do b/c our #vets deserve nothing less! #sxsw #letshookup
8902	@mention Which is to say iPad is going to be ubiquitous a lot faster than anyone expected a year or even 6 mo. ago. #newsapps #sxsw

In [22]:

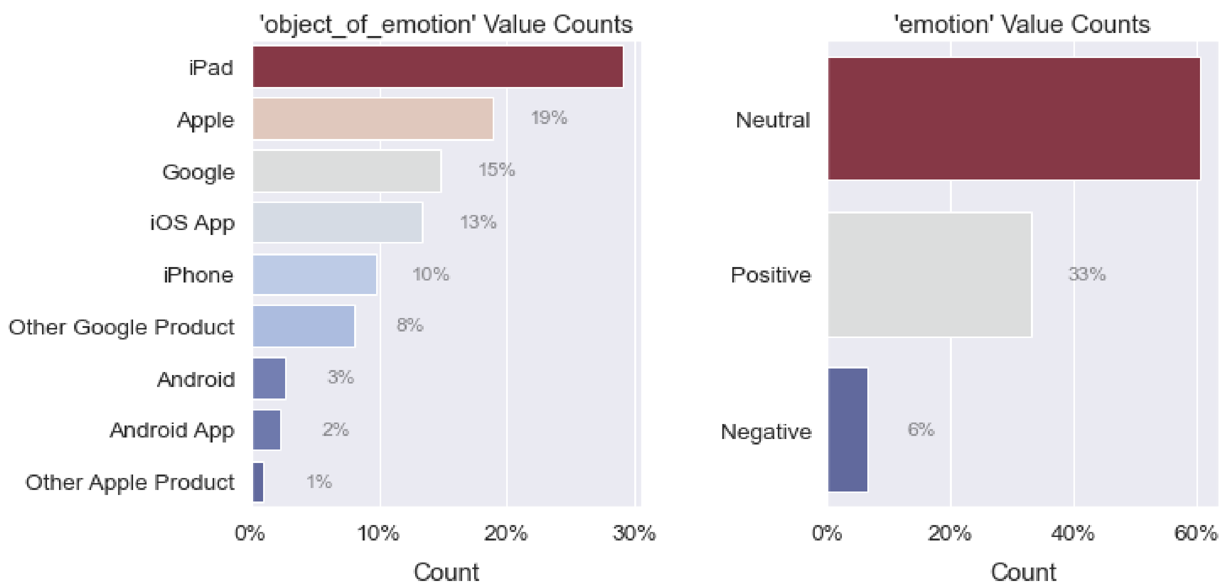
```
# Set object to null where emotion is neutral
df.loc[object_without_emotion.index, "object_of_emotion"] = np.nan

# Ensure that 'Neutral' rows line up with 'NaN' rows
(df["emotion"] == "Neutral").equals(df["object_of_emotion"].isnull())
```

Out[22]: True

Here's a look at the final distributions.

In [23]:

```
fig = plotting.countplot(df.select_dtypes("category"), normalize=1)
```



Duplicates

There are 22 duplicate rows, and even more when only the text is considered. I don't want to get rid of all retweets, but I do want to get rid of those which don't have novel content.

In [24]:

```
cleaning.dup_rows(df.text).sort_values()
```

Out[24]:

```
3962    #SXSW is just starting, #CTIA is around the co...
468        Before It Even Begins, Apple Wins #SXSW {link}
2559    Counting down the days to #sxsw plus strong Ca...
776     Google to Launch Major New Social Network Call...
8483    I just noticed DST is coming this weekend. How...
2232    Marissa Mayer: Google Will Connect the Digital...
8747    Need to buy an iPad2 while I'm in Austin at #s...
4897    Oh. My. God. The #SXSW app for iPad is pure, u...
5882    RT @mention Google to Launch Major New Social ...
5884    RT @mention Google to Launch Major New Social ...
5883    RT @mention Google to Launch Major New Social ...
5881    RT @mention Google to Launch Major New Social ...
5885    RT @mention Google to Launch Major New Social ...
6299    RT @mention Marissa Mayer: Google Will Connect...
6297    RT @mention Marissa Mayer: Google Will Connect...
6295    RT @mention Marissa Mayer: Google Will Connect...
6300    RT @mention Marissa Mayer: Google Will Connect...
6298    RT @mention Marissa Mayer: Google Will Connect...
6294    RT @mention Marissa Mayer: Google Will Connect...
6296    RT @mention Marissa Mayer: Google Will Connect...
6546    RT @mention RT @mention Google to Launch Major...
6576    RT @mention RT @mention It's not a rumor: Appl...
5338    RT @mention ��� GO BEYOND BORDERS! ��_ {link} ...
5341    RT @mention ��� Happy Woman's Day! Make love, ...
3950    Really enjoying the changes in Gowalla 3.0 for...
3814              Win free iPad 2 from webdoc.com #sxsw RT
3813              Win free ipad 2 from webdoc.com #sxsw RT
Name: text, dtype: object
```

I filter the text by removing occurrences of 'RT' and then check for duplicates. This should get rid of retweets which are just copies of original tweets in the dataset.

In [25]:

```
dups = df.text.str.replace(r"\s*RT\s*", "", regex=True).duplicated()
df = df.loc[~dups]
dups.sum()
```

Out[25]:

```
33
```

Define Stopwords

But before I proceed further, I want to define some stopwords for this particular project.

In [26]:

```
# SXSW and Twitter stopwords
MY_STOP = frozenset(
    {
        "america",
        "austin",
        "link",
        "mention",
        "rt",
        "southbysouthwest",
        "sxsw",
        "sxswi",
    }
)
```

```python
# Brand-related stopwords
BRAND_STOP = frozenset(
    {
        "app",
        "androidsxsw",
        "apple",
        "applesxsw",
        "android",
        "google",
        "iphone",
        "ipad",
        "andoid",
    }
)

MY_STOP, BRAND_STOP
```

Out[26]:

```
(frozenset({'america',
            'austin',
            'link',
            'mention',
            'rt',
            'southbysouthwest',
            'sxsw',
            'sxswi'}),
 frozenset({'andoid',
            'android',
            'androidsxsw',
            'app',
            'apple',
            'applesxsw',
            'google',
            'ipad',
            'iphone'}))
```

I save the stopword sets in JSON.

In [27]:

```python
# Create JSON-serializable dict
stopwords = {
    "MY_STOP": list(MY_STOP),
    "BRAND_STOP": list(BRAND_STOP),
}

# Save my stopwords
with open("data/stopwords.json", "w") as f:
    json.dump(stopwords, f)

del stopwords
```

Feature Engineering

Brand Terms

I extract brand terms based on the crowdsourced labels using regular expressions. I'm comfortable using these for training the model, since they were extracted algorithmically.

In [28]:

```python
# Combine my previous brand patterns
re_brand = fr"{re_apple}|{re_google}"
print(re_brand)

# Extract from raw text with document indices
```

```python
regex_brands = lang.locate_patterns(re_brand, strings=df.text, flags=re.I)

regex_brands.head(10)
```

```
ipad\s*\d?\s*app|ipad|iphone\s*\d?\s*app|iphone|apple|android\s*app|android|google
```

Out[28]:

```
0         iPhone
1           iPad
1     iPhone app
2           iPad
3     iPhone app
4         Google
5       iPad App
7         google
7        android
8       iPad app
Name: locate_patterns, dtype: object
```

Now I clean up the terms.

In [29]:

```python
regex_brands = (
    regex_brands
    # Make lowercase
    .str.lower()
    # Strip numerals
    .map(lang.strip_numeric)
    # Strip extra whitespace
    .map(lang.strip_multiwhite)
    # Deal with cases like 'iphoneapp'
    .str.replace(r"([a-z]+)app", lambda x: f"{x[1]} app", regex=True)
    # Replace space with underscore
    .str.replace(" ", "_")
)


regex_brands.unique()
```

Out[29]:

```
array(['iphone', 'ipad', 'iphone_app', 'google', 'ipad_app', 'android',
       'apple', 'android_app'], dtype=object)
```

Now the terms are ready to go into `df`. I retract them into nested lists with unique indices, add whatever indices are missing to make them match `df`, and create a 'None' category.

In [30]:

```python
# Retract into nested lists and index like `df`
regex_brands = utils.implode(regex_brands).reindex_like(df)

# Create 'None' category
regex_brands[regex_brands.isnull()] = ["none"]

# Put brand terms in new column
df["brand_terms"] = regex_brands

# Clear namespace
del regex_brands, re_apple, re_google, re_brand

# Show uniques
df["brand_terms"].explode().unique()
```

Out[30]:

```
array(['iphone', 'ipad', 'iphone_app', 'google', 'ipad_app', 'android',
       'apple', 'android_app', 'none'], dtype=object)
```

Simple Counts

I engineer character counts (minus spaces), word counts, and average word lengths for exploratory purposes.

Maybe an interesting pattern will show up.

In [31]:
```
# String length without whitespace
df["n_chars"] = df["text"].str.replace("\s+", "", regex=True).map(len)

# Number of words as parsed by TweetTokenizer
df["n_words"] = df["text"].map(nltk.casual_tokenize).map(len)

# Calculate average word length
df["avg_word_len"] = df["n_chars"] / df["n_words"]

# Show results
df[["n_chars", "n_words", "avg_word_len"]].head()
```

Out[31]:

	n_chars	n_words	avg_word_len
0	104	29	3.586207
1	118	26	4.538462
2	65	17	3.823529
3	68	16	4.250000
4	115	27	4.259259

I engineer exclamation point and question mark counts, which I've discovered have a surprisingly robust connection to sentiment.

In [32]:
```
df["ep_count"] = df["text"].str.count(r"\!")
df["qm_count"] = df["text"].str.count(r"\?")
df[["ep_count", "qm_count"]].head()
```

Out[32]:

	ep_count	qm_count
0	1	0
1	0	1
2	0	0
3	0	0
4	0	0

In [33]:
```
df.to_json(normpath("data/processed_tweets.json"))
```

Modeling

I develop my final model through an iterative process, starting with a basic, baseline version of the model.

Before I do anything, I turn my nested lists of brand terms into strings which can be vectorized. Vectorization is just a convenient way to one-hot-encode them.

In [34]:
```
df["brand_terms"] = df["brand_terms"].str.join(" ")
df["brand_terms"].head()
```

Out[34]:
```
0              iphone
1     ipad iphone_app
2                ipad
3          iphone_app
4              google
Name: brand_terms, dtype: object
```

Train-Test-Split

I perform the train-test split which I'll use throughout my modeling process. I let `X` and its derivatives be `DataFrame` objects because I plan to use a `ColumnTransformer` to process the two columns separately.

In [35]:
```
cols = [
    "text",
    "brand_terms",
]

# Define X and y
X = df.loc[:, cols].copy()
y = df.emotion.to_numpy()

# Perform the split
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    random_state=rando,
    stratify=y,
    shuffle=True,
)

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[35]:
```
((6659, 2), (6659,), (2220, 2), (2220,))
```

My `FreqVectorizer`

I first create a `ColumnTransformer` to process the data from columns of `X` and concatenate the results.

Inside `col_xform` are two of my `FreqVectorizer` objects. `FreqVectorizer` extends Scikit-Learn's `TfidfVectorizer` and adds a number of powerful preprocessing options. It's called `FreqVectorizer` to emphasize that, like its parent class, it offers several different word-frequency-based vectorization algorithms. Among these algorithms are term frequency (count) vectorization and TF*IDF (term frequency * inverse document frequency) vectorization. By default, `FreqVectorizer` is set to perform count vectorization.

In `col_xform`, the brand terms ('bra') are effectively one-hot-encoded using default tokenization and `binary=True`. This is how the feature was designed to be encoded. The raw text ('txt') is treated with simple count vectorization as a baseline option.

I'll discuss more details of my `FreqVectorizer` class as they become relevant.

In [36]:
```
col_xform = ColumnTransformer(
    [
        ("txt", FreqVectorizer(), "text"),
        (
            "bra",
            FreqVectorizer(binary=True),
            "brand_terms",
        ),
```

```
    ],
)
col_xform
```

Out[36]:
```
ColumnTransformer(transformers=[('txt', FreqVectorizer(), 'text'),
                                ('bra', FreqVectorizer(binary=True),
                                 'brand_terms')])
```

I check to make sure that all and only the correct brand terms are present.

In [37]:
```
col_xform.fit(X_train).named_transformers_["bra"].get_feature_names()
```

Out[37]:
```
['android',
 'android_app',
 'apple',
 'google',
 'ipad',
 'ipad_app',
 'iphone',
 'iphone_app']
```

For more information about `FreqVectorizer`, see the help page below.

In [38]:
```
help(FreqVectorizer)
```

```
Help on class FreqVectorizer in module tools.sklearn.vectorizers:

class FreqVectorizer(sklearn.feature_extraction.text.TfidfVectorizer, VectorizerMixin)
 |  FreqVectorizer(*, input='content', encoding='utf-8', decode_error='strict', strip_accents=None,
decode_html_entities=True, lowercase=True, strip_multiwhite=False, strip_numeric=False, split_alpha
num=False, alphanum_only=False, strip_punct=False, strip_twitter_handles=False, strip_html_tags=Fal
se, limit_repeats=False, filter_length=(None, None), stemmer=None, mark=None, preprocessor=None, to
kenizer=None, analyzer='word', stop_words=None, token_pattern='(?u)\\b\\w\\w+\\b', ngram_range=(1,
1), max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<class 'numpy.flo
at64'>, norm=None, use_idf=False, smooth_idf=True, sublinear_tf=False)
 |
 |  Convert a collection of raw documents to a matrix of word-frequency features.
 |
 |  Extends Scikit-Learn's `TfidfVectorizer` with advanced preprocessing options.
 |  These include numerous filters, stemming/lemmatization, and markers such as PoS tags.
 |  Some preprocessing options are applied before tokenization, and some, which require
 |  tokens, are applied during the tokenization step.
 |
 |  There are now a wider selection of built-in stopwords sets, and these include the NLTK
 |  sets for many different languages. Complex stopwords queries are now also supported.
 |
 |
 |  Parameters
 |  ----------
 |  input : {'filename', 'file', 'content'}, default='content'
 |      If 'filename', the sequence passed as an argument to fit is
 |      expected to be a list of filenames that need reading to fetch
 |      the raw content to analyze.
 |
 |      If 'file', the sequence items must have a 'read' method (file-like
 |      object) that is called to fetch the bytes in memory.
 |
 |      Otherwise the input is expected to be a sequence of items that
 |      can be of type string or byte.
 |
 |  encoding : str, default='utf-8'
 |      If bytes or files are given to analyze, this encoding is used to
 |      decode.
 |
 |  decode_error : {'strict', 'ignore', 'replace'}, default='strict'
 |      Instruction on what to do if a byte sequence is given to analyze that
 |      contains characters not of the given `encoding`. By default, it is
 |      'strict', meaning that a UnicodeDecodeError will be raised. Other
```

```
 |      values are 'ignore' and 'replace'.
 |
 |  strip_accents : {'ascii', 'unicode'}
 |      Remove accents and perform other character normalization
 |      during the preprocessing step.
 |      * 'ascii' is a fast method that only works on characters that have
 |          an direct ASCII mapping.
 |      * 'unicode' is a slightly slower method that works on any characters.
 |      *  None (default) does nothing.
 |
 |      Both 'ascii' and 'unicode' use NFKD normalization from
 |      :func:`unicodedata.normalize`.
 |
 |  decode_html_entities : bool, ** NEW **
 |      Decode HTML entities such as '&mdash;' or '&lt;' or '&gt;' into symbols,
 |      e.g. '—', '<', '>'. True by default.
 |
 |  lowercase : bool
 |      Convert all characters to lowercase before tokenizing. True by default.
 |
 |  strip_multiwhite: bool, ** NEW **
 |      Strip extra whitespaces (including tabs and newlines). False by default.
 |
 |  strip_numeric: bool, ** NEW **
 |      Strip numerals [0-9] from text. False by default.
 |
 |  split_alphanum: bool, ** NEW **
 |      Add space between alphabetic and numeric characters which appear together
 |      in a word-like sequence. For example, 'spiderman2' would become 'spiderman 2'.
 |      False by default.
 |
 |  alphanum_only: bool, ** NEW **
 |      Strip all non-alphanumeric characters (except underscore). False by default.
 |
 |  strip_punct: bool or str of punctuation symbols
 |      If True, strip all punctuation. If passed a string of punctuation symbols, strip
 |      only those symbols. False by default.
 |
 |  strip_twitter_handles: bool, ** NEW **
 |      Strip Twitter @mentions. False by default.
 |
 |  strip_html_tags: bool, ** NEW **
 |      Strip HTML tags such as '<p>' or '<div>'. False by default.
 |
 |  limit_repeats: bool, ** NEW **
 |      Limit strings of repeating characters, e.g. 'zzzzzzzzzzz', to length 3.
 |
 |  filter_length: tuple (int, int), ** NEW **
 |      Drop tokens which are outside the prescribed character length range.
 |      Range is inclusive. Defaults to (None, None).
 |
 |  stemmer: {'porter', 'wordnet'}, ** NEW **
 |      Stemming or lemmatization algorithm to use. Both implement caching in order to
 |      reuse previous computations. Valid options:
 |      * 'porter' - Porter stemming algorithm (faster).
 |      * 'wordnet' - Lemmatization using Wordnet (slower).
 |      * None - Do not stem tokens (default).
 |
 |  mark: str ** NEW **
 |      Mark negation or parts of speech. Valid options:
 |      * 'neg' - Mark words between a negating term and sentence punctuation with '_NEG'.
 |      * 'neg_split' - Mark negation but let the tags be independent tokens.
 |      * 'speech' - Mark parts of speech with e.g. '_NNS' using the recommended NLTK tagger.
 |      * 'speech_split' - Mark parts of speech but let the tags be independent tokens.
 |      * 'speech_replace' - Replace word tokens with their parts of speech.
 |      * None - Do not mark tokens (default).
 |
 |  preprocessor : callable, default=None
 |      Override the preprocessing (string transformation) stage while
 |      preserving the tokenizing and n-grams generation steps.
 |      Only applies if ``analyzer is not callable``.
 |
```

```
 |  tokenizer : callable, default=None
 |      Override the string tokenization step while preserving the
 |      preprocessing and n-grams generation steps.
 |      Only applies if ``analyzer == 'word'``.
 |
 |  analyzer : {'word', 'char', 'char_wb'} or callable, default='word'
 |      Whether the feature should be made of word or character n-grams.
 |      Option 'char_wb' creates character n-grams only from text inside
 |      word boundaries; n-grams at the edges of words are padded with space.
 |
 |      If a callable is passed it is used to extract the sequence of features
 |      out of the raw, unprocessed input.
 |
 |  stop_words : str, list, ** IMPROVED **
 |      If a string, it is passed to `tools.language.fetch_stopwords` and
 |      the appropriate stopword list is returned. Valid strings:
 |      * 'skl_english' - Scikit-Learn's English stopwords.
 |      * 'nltk_LANGUAGE' - Any NLTK stopwords set, where the fileid (language) follows the undersc
ore.
 |          For example: 'nltk_english', 'nltk_french', 'nltk_spanish'.
 |      * 'gensim_english' - Gensim's English stopwords set.
 |      * Supports complex queries using set operators, e.g. '(nltk_french & nltk_spanish) | skl_en
glish'.
 |
 |      If a list, that list is assumed to contain stop words, all of which
 |      will be removed from the resulting tokens.
 |      Only applies if ``analyzer == 'word'``.
 |
 |      If None, no stop words will be used. max_df can be set to a value
 |      in the range [0.7, 1.0) to automatically detect and filter stop
 |      words based on intra corpus document frequency of terms.
 |
 |  token_pattern : str, default=r"(?u)\b\w\w+\b"
 |      Regular expression denoting what constitutes a "token", only used
 |      if ``analyzer == 'word'``. The default regexp selects tokens of 2
 |      or more alphanumeric characters (punctuation is completely ignored
 |      and always treated as a token separator).
 |
 |      If there is a capturing group in token_pattern then the
 |      captured group content, not the entire match, becomes the token.
 |      At most one capturing group is permitted.
 |
 |  ngram_range : tuple (min_n, max_n)
 |      The lower and upper boundary of the range of n-values for different
 |      n-grams to be extracted. All values of n such that min_n <= n <= max_n
 |      will be used. For example an ``ngram_range`` of ``(1, 1)`` means only
 |      unigrams, ``(1, 2)`` means unigrams and bigrams, and ``(2, 2)`` means
 |      only bigrams. Defaults to (1, 1).
 |      Only applies if ``analyzer is not callable``.
 |
 |  max_df : float or int
 |      When building the vocabulary ignore terms that have a document
 |      frequency strictly higher than the given threshold (corpus-specific
 |      stop words). Defaults to 1.0.
 |      If float in range [0.0, 1.0], the parameter represents a proportion of
 |      documents, integer absolute counts.
 |      This parameter is ignored if vocabulary is not None.
 |
 |  min_df : float or int
 |      When building the vocabulary ignore terms that have a document
 |      frequency strictly lower than the given threshold. This value is also
 |      called cut-off in the literature. Defaults to 1.
 |      If float in range of [0.0, 1.0], the parameter represents a proportion
 |      of documents, integer absolute counts.
 |      This parameter is ignored if vocabulary is not None.
 |
 |  max_features : int
 |      If not None, build a vocabulary that only consider the top
 |      max_features ordered by term frequency across the corpus.
 |      None by default.
 |
 |      This parameter is ignored if vocabulary is not None.
```

```
 |
 |  vocabulary : Mapping or iterable
 |      Either a Mapping (e.g., a dict) where keys are terms and values are
 |      indices in the feature matrix, or an iterable over terms. If not
 |      given, a vocabulary is determined from the input documents. None by default.
 |
 |  binary : bool
 |      If True, all non-zero term counts are set to 1. This does not mean
 |      outputs will have only 0/1 values, only that the tf term in tf-idf
 |      is binary. (Set idf and normalization to False to get 0/1 outputs).
 |      False by default.
 |
 |  dtype : dtype
 |      Type of the matrix returned by fit_transform() or transform().
 |      'float64' by default.
 |
 |  norm : {'l2', 'l1', 'max'}
 |      Each output row will have unit norm, either:
 |      * 'l2': Sum of squares of vector elements is 1. The cosine
 |      similarity between two vectors is their dot product when l2 norm has
 |      been applied. None by default.
 |      * 'l1': Sum of absolute values of vector elements is 1.
 |      See :func:`preprocessing.normalize`.
 |
 |  use_idf : bool
 |      Enable inverse-document-frequency reweighting. False by default.
 |
 |  smooth_idf : bool
 |      Smooth idf weights by adding one to document frequencies, as if an
 |      extra document was seen containing every term in the collection
 |      exactly once. Prevents zero divisions. True by default.
 |
 |  sublinear_tf : bool
 |      Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).
 |      False by default.
 |
 |  Attributes
 |  ----------
 |  vocabulary_ : dict
 |      A mapping of terms to feature indices.
 |
 |  fixed_vocabulary_: bool
 |      True if a fixed vocabulary of term to indices mapping
 |      is provided by the user
 |
 |  idf_ : array of shape (n_features,)
 |      The inverse document frequency (IDF) vector; only defined
 |      if ``use_idf`` is True.
 |
 |  stop_words_ : set
 |      Terms that were ignored because they either:
 |
 |        - occurred in too many documents (`max_df`)
 |        - occurred in too few documents (`min_df`)
 |        - were cut off by feature selection (`max_features`).
 |
 |      This is only available if no vocabulary was given.
 |
 |  Method resolution order:
 |      FreqVectorizer
 |      sklearn.feature_extraction.text.TfidfVectorizer
 |      sklearn.feature_extraction.text.CountVectorizer
 |      VectorizerMixin
 |      sklearn.feature_extraction.text._VectorizerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, input='content', encoding='utf-8', decode_error='strict', strip_accents=None,
decode_html_entities=True, lowercase=True, strip_multiwhite=False, strip_numeric=False, split_alpha
num=False, alphanum_only=False, strip_punct=False, strip_twitter_handles=False, strip_html_tags=Fal
```

```
se, limit_repeats=False, filter_length=(None, None), stemmer=None, mark=None, preprocessor=None, to
kenizer=None, analyzer='word', stop_words=None, token_pattern='(?u)\\b\\w\\w+\\b', ngram_range=(1,
1), max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<class 'numpy.flo
at64'>, norm=None, use_idf=False, smooth_idf=True, sublinear_tf=False)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.feature_extraction.text.TfidfVectorizer:
 |
 |  fit(self, raw_documents, y=None)
 |      Learn vocabulary and idf from training set.
 |
 |      Parameters
 |      ----------
 |      raw_documents : iterable
 |          An iterable which yields either str, unicode or file objects.
 |      y : None
 |          This parameter is not needed to compute tfidf.
 |
 |      Returns
 |      -------
 |      self : object
 |          Fitted vectorizer.
 |
 |  fit_transform(self, raw_documents, y=None)
 |      Learn vocabulary and idf, return document-term matrix.
 |
 |      This is equivalent to fit followed by transform, but more efficiently
 |      implemented.
 |
 |      Parameters
 |      ----------
 |      raw_documents : iterable
 |          An iterable which yields either str, unicode or file objects.
 |      y : None
 |          This parameter is ignored.
 |
 |      Returns
 |      -------
 |      X : sparse matrix of (n_samples, n_features)
 |          Tf-idf-weighted document-term matrix.
 |
 |  transform(self, raw_documents)
 |      Transform documents to document-term matrix.
 |
 |      Uses the vocabulary and document frequencies (df) learned by fit (or
 |      fit_transform).
 |
 |      Parameters
 |      ----------
 |      raw_documents : iterable
 |          An iterable which yields either str, unicode or file objects.
 |
 |      Returns
 |      -------
 |      X : sparse matrix of (n_samples, n_features)
 |          Tf-idf-weighted document-term matrix.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.feature_extraction.text.TfidfVectorizer:
 |
 |  idf_
 |
 |  norm
 |
 |  smooth_idf
 |
 |  sublinear_tf
 |
 |  use_idf
 |
 |  ----------------------------------------------------------------------
```

```
 |  Methods inherited from sklearn.feature_extraction.text.CountVectorizer:
 |
 |  get_feature_names(self)
 |      Array mapping from feature integer indices to feature name.
 |
 |      Returns
 |      -------
 |      feature_names : list
 |          A list of feature names.
 |
 |  inverse_transform(self, X)
 |      Return terms per document with nonzero entries in X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          Document-term matrix.
 |
 |      Returns
 |      -------
 |      X_inv : list of arrays of shape (n_samples,)
 |          List of arrays of terms.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from VectorizerMixin:
 |
 |  build_preprocessor(self)
 |      Return a function to preprocess the text before tokenization.
 |
 |      Returns
 |      -------
 |      preprocessor: callable
 |            A function to preprocess the text before tokenization.
 |
 |  build_tokenizer(self)
 |      Return a function that splits a string into a sequence of tokens.
 |
 |      Returns
 |      -------
 |      tokenizer: callable
 |            A function to split a string into a sequence of tokens.
 |
 |  get_stop_words(self)
 |      Build or fetch the effective stop words set.
 |
 |      Returns
 |      -------
 |      stop_words: frozenset or None
 |              A set of stop words.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.feature_extraction.text._VectorizerMixin:
 |
 |  build_analyzer(self)
 |      Return a callable that handles preprocessing, tokenization
 |      and n-grams generation.
 |
 |      Returns
 |      -------
 |      analyzer: callable
 |          A function to handle preprocessing, tokenization
 |          and n-grams generation.
 |
 |  decode(self, doc)
 |      Decode the input into a string of unicode symbols.
 |
 |      The decoding strategy depends on the vectorizer parameters.
 |
 |      Parameters
 |      ----------
 |      doc : str
 |          The string to decode.
```

```
 |
 |      Returns
 |      -------
 |      doc: str
 |          A string of unicode symbols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.feature_extraction.text._VectorizerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
```

Baseline Model: Logistic Regression

I create a `LogisticRegression` classifier. Logistic Regression is my go-to option for classification, and it performs well on this dataset. Since the `y` classes are wildly imbalanced, I set `class_weight='balanced'`. I also hike up `max_iter` because otherwise the model fails to converge.

In [39]:

```
logit = LogisticRegression(
    class_weight="balanced",
    max_iter=1000,
    random_state=rando,
```

```
)

logit
```

Out[39]:
```
LogisticRegression(class_weight='balanced', max_iter=1000,
                   random_state=RandomState(MT19937) at 0x2D788219640)
```

I create my main `Pipeline`, consisting simply of `col_xform` and `logit`.

In [40]:
```
main_pipe = Pipeline(
    [
        ("col", col_xform),
        ("cls", logit),
    ]
)
main_pipe
```

Out[40]:
```
Pipeline(steps=[('col',
                 ColumnTransformer(transformers=[('txt', FreqVectorizer(),
                                                  'text'),
                                                 ('bra',
                                                  FreqVectorizer(binary=True),
                                                  'brand_terms')])),
                ('cls',
                 LogisticRegression(class_weight='balanced', max_iter=1000,
                                    random_state=RandomState(MT19937) at 0x2D788219640))])
```

In [41]:
```
# Make copy of baseline for future reference
baseline = clone(main_pipe)
```

Looks like `col_xform` is outputting ~8,500 features and ~6,500 samples (i.e. vectors, observations, tweets) with the current settings. The features are words (found in the text) and preset brand terms.

In [42]:
```
vecs = col_xform.fit_transform(X_train)
vecs
```

Out[42]:
```
<6659x8447 sparse matrix of type '<class 'numpy.float64'>'
        with 114260 stored elements in Compressed Sparse Row format>
```

Here are the results of the default preprocessing and tokenizing. The minimalist default settings actually look pretty good.

In [43]:
```
analyzer = col_xform.named_transformers_["txt"].build_analyzer()
df.text.head(10).map(analyzer)
```

Out[43]:
```
0     [wesley83, have, 3g, iphone, after, hrs, tweet...
1     [jessedee, know, about, fludapp, awesome, ipad...
2     [swonderlin, can, not, wait, for, ipad, also, ...
3     [sxsw, hope, this, year, festival, isn, as, cr...
4     [sxtxstate, great, stuff, on, fri, sxsw, maris...
5     [teachntech00, new, ipad, apps, for, speechthe...
7     [sxsw, is, just, starting, ctia, is, around, t...
8     [beautifully, smart, and, simple, idea, rt, ma...
9     [counting, down, the, days, to, sxsw, plus, st...
10    [excited, to, meet, the, samsungmobileus, at, ...
Name: text, dtype: object
```

Fitting the Model

I create a dictionary to hold `X_train`, `X_test`, `y_train`, and `y_test` for easy access.

In [44]:

```
split_data = dict(
    X_train=X_train,
    X_test=X_test,
    y_train=y_train,
    y_test=y_test,
)

split_data.keys()
```

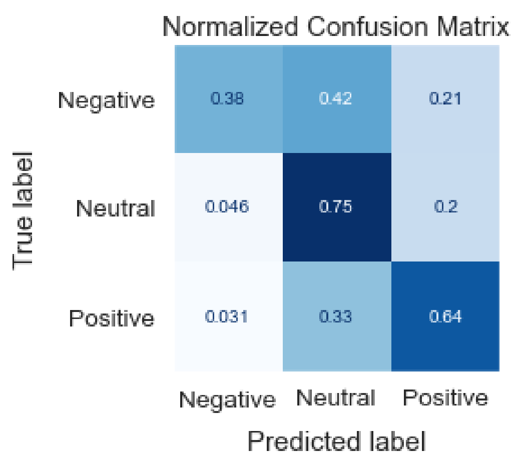
Out[44]:

```
dict_keys(['X_train', 'X_test', 'y_train', 'y_test'])
```

In [45]:

```
diag.test_fit(main_pipe, **split_data)
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.384	0.770	0.609	0.588	0.692	0.690	0.589
recall	0.376	0.751	0.640	0.589	0.690		
f1-score	0.380	0.760	0.624	0.588	0.691		
support	0.064	0.605	0.332				



Not the best model I've ever seen, but it's a baseline. The text is being count-vectorized, which is a pretty crude strategy. That'll be the first thing to change.

Compare with Dummy

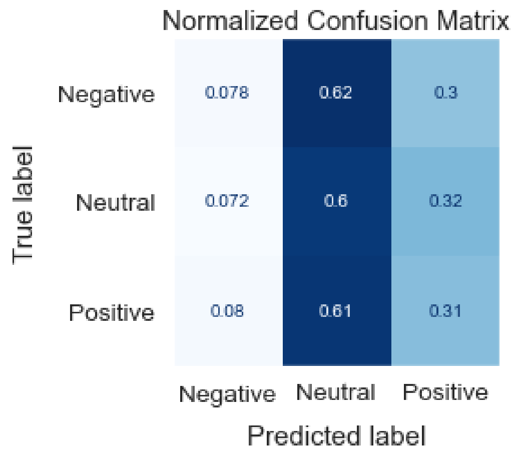
The baseline, while crude, is much better than the dummy model. This `DummyClassifier` algorithm randomly selects classes with probability weighted according to the class balance. With this dataset, it almost never selects Negative, and it's a 60-30 split between Neutral and Positive.

In [46]:

```
dummy = DummyClassifier(strategy="stratified", random_state=15)
dummy_pipe = clone(main_pipe).set_params(cls=dummy)
diag.test_fit(dummy_pipe, **split_data)
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.066	0.602	0.324	0.331	0.476	0.473	0.331
recall	0.078	0.605	0.310	0.331	0.473		
f1-score	0.071	0.603	0.317	0.330	0.474		

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
support	0.064	0.605	0.332				



In [47]:

```
sweep_params = dict(
    X=X_train,
    y=y_train,
    scoring="balanced_accuracy",
    n_jobs=-1,
    cv=5,
)

sweep_params.keys()
```

Out[47]: `dict_keys(['X', 'y', 'scoring', 'n_jobs', 'cv'])`

Configure Logistic Regression

I begin by running a search over the `LogisticRegression` hyperparameters. This includes penalty type ('l1', 'l2', 'elasticnet'), regularization strength ('C'), multi-class strategy, whether to fit an intercept, and the solver. I also include the most important `FreqVectorizer` parameters.

My `sweep` Function

My `selection.sweep` function is a generic function for searching parameter spaces using Scikit-Learn. If you pass `kind='grid'`, it fits a `GridSearchCV`, running an exhaustive search over every combination of parameters. This is the default (and most thorough) option. You can also pass `kind='rand'` to fit a `RandomizedSearchCV`, which searches a random sample of the parameter space. If you want to speed things up, you can pass `kind='hgrid'` or `kind='hrand'` to fit Scikit-Learn's experimental `HalvingGridSearchCV` or its randomized counterpart. The "halving" searches try to weed out the weak candidates using only a small amount of computational resources (e.g. a small sample of the data).

Rather than returning a `GridSearchCV` object or equivalent, `selection.sweep` immediately serializes the search estimator and saves it via `joblib`. This is done to prevent loss of the search results. It's very easy to load a serialized search estimator, and I have a function `selection.load_results` which trims down the `cv_results_` and returns a `DataFrame`.

First, I run a quick search to see which solvers are efficient for this dataset.

In [48]:

```
solver_grid = {
```

```
    "solver": ["liblinear", "lbfgs", "newton-cg", "sag", "saga"],
    # High max iterations to gauge speed
    "max_iter": [1e4],
}
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        solver_grid,
        add_prefix="cls__",
        dst="sweeps/logit_solvers",
        **sweep_params,
    )
```

Looks like 'newton-cg', 'lbfgs', and 'liblinear' are efficient on this dataset. 'newton-cg' looks the most promising overall. The fastest is definitely 'liblinear', but it also has the lowest mean score. The slowest by far are 'sag' and 'saga'. They're so slow that I'm not going to include them in the next sweep.

In [49]:

```
selection.load_results("sweeps/logit_solvers").style.bar("mean_fit_time")
```

Out[49]:

	max_iter	solver	mean_fit_time	mean_score	rank_score
0	10000.000000	newton-cg	1.217998	0.574055	1
1	10000.000000	lbfgs	1.973602	0.574055	1
2	10000.000000	saga	37.956995	0.572681	3
3	10000.000000	sag	43.558397	0.565402	4
4	10000.000000	liblinear	0.655007	0.563722	5

Constructing document vectors with raw **term frequencies** is a very crude approach. Words like 'the', if not filtered out, will have a high frequency in many tweets. But 'the' contains no information about the tweet's overall content. The TF*IDF algorithm addresses this problem by normalizing term frequencies according to **inverse document frequency**. A term's inverse document frequency is the (logarithmically scaled) number of documents in the corpus divided by the number of documents containing the term. It represents the rarity of a term.

I lay out the TF*IDF parameters which determine whether `FreqVectorizer` produces binary occurrence vectors, count vectors, normalized occurrence vectors, or normalized TF*IDF vectors. The 'norm' often strongly affects model quality, so I've included that too.

In [50]:

```
tfidf_grid = {
    "binary": [True, False],
    "norm": ["l2", "l1", None],
    "use_idf": [True, False],
}

tfidf_grid = pd.Series(tfidf_grid).add_prefix("col__txt__")
tfidf_grid
```

Out[50]:

```
col__txt__binary      [True, False]
col__txt__norm       [l2, l1, None]
col__txt__use_idf     [True, False]
dtype: object
```

In [51]:

```
logit_grid = [
    # lbfgs & newton-cg: L2
    {
        "cls__C": np.geomspace(1e-3, 1e3, 7),
```

```python
            "cls__fit_intercept": [True, False],
            "cls__multi_class": ["multinomial", "ovr"],
            "cls__solver": ["lbfgs", "newton-cg"],
            "cls__penalty": ["l2"],
            **tfidf_grid,
        },
        # lbfgs & newton-cg: no penalty
        {
            "cls__fit_intercept": [True, False],
            "cls__multi_class": ["multinomial", "ovr"],
            "cls__solver": ["lbfgs", "newton-cg"],
            "cls__penalty": ["none"],
            **tfidf_grid,
        },
        # liblinear: L1
        {
            "cls__C": np.geomspace(1e-3, 1e3, 7),
            "cls__fit_intercept": [True, False],
            "cls__multi_class": ["ovr"],
            "cls__solver": ["liblinear"],
            "cls__penalty": ["l1"],
            **tfidf_grid,
        },
        # liblinear: L2 & dual
        {
            "cls__C": np.geomspace(1e-3, 1e3, 7),
            "cls__dual": [True, False],
            "cls__fit_intercept": [True, False],
            "cls__multi_class": ["ovr"],
            "cls__solver": ["liblinear"],
            "cls__penalty": ["l2"],
            **tfidf_grid,
        },
    ]

    # Show size of param space
    selection.space_size(logit_grid)
```

Out[51]:

```
n_params       9
n_combos    1272
n_folds        5
n_fits      6360
dtype: int64
```

In [52]:

```python
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        logit_grid,
        dst="sweeps/logit",
        **sweep_params,
    )
```

In [53]:

```python
results = selection.load_results("sweeps/logit", drop_dicts=False)

# Hide param dicts for display
results.drop(columns="params").head(10).style.bar("mean_score")
```

Out[53]:

	C	dual	fit_intercept	multi_class	penalty	solver	binary	norm	use_idf	mean_fit_time	mean_score	r
0	10.000000	nan	True	multinomial	l2	lbfgs	False	l1	True	3.677400	0.614568	
1	10.000000	nan	True	multinomial	l2	newton-cg	False	l1	True	1.463201	0.614485	

	C	dual	fit_intercept	multi_class	penalty	solver	binary	norm	use_idf	mean_fit_time	mean_score	r
2	10.000000	nan	False	multinomial	l2	lbfgs	False	l1	True	2.412797	0.613091	
3	10.000000	nan	False	multinomial	l2	newton-cg	False	l1	True	1.177401	0.612073	
4	10.000000	nan	False	multinomial	l2	newton-cg	True	l1	True	1.145798	0.611669	
5	10.000000	nan	False	multinomial	l2	lbfgs	True	l1	True	2.412201	0.611586	
6	10.000000	nan	True	multinomial	l2	newton-cg	True	l1	True	1.509003	0.610617	
7	10.000000	nan	True	multinomial	l2	lbfgs	True	l1	True	3.557401	0.610535	
8	1.000000	nan	True	multinomial	l2	newton-cg	False	l2	True	0.992597	0.607395	
9	1.000000	nan	True	multinomial	l2	lbfgs	False	l2	True	1.607802	0.607395	

Fitting the Model

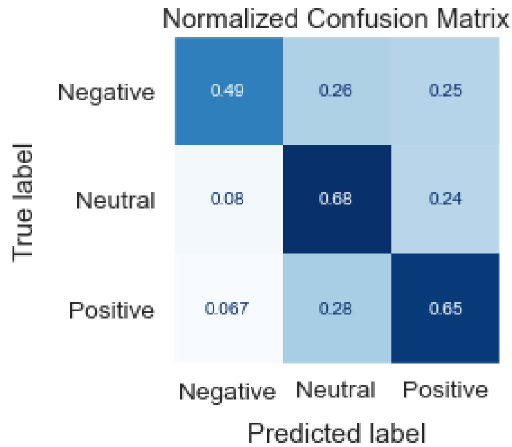
The solvers 'newton-cg' and 'lbfgs' seem to be roughly the same, except that 'newton-cg' is consistently faster. I go with 'newton-cg', for speed. After trying some of the top-ranking combinations, I've decided to go with the following: no intercept, multinomial strategy, L2 regularization, `C=10`, and TF*IDF vectors with L1 norm.

In [54]:

```
choice = 3

# Get params from search results and set them
main_pipe.set_params(**results.iloc[choice].params)

# Show what I'm setting
display(results.iloc[choice].params)

diag.test_fit(main_pipe, **split_data)
```

```
{'cls__C': 10.0,
 'cls__fit_intercept': False,
 'cls__multi_class': 'multinomial',
 'cls__penalty': 'l2',
 'cls__solver': 'newton-cg',
 'col__txt__binary': False,
 'col__txt__norm': 'l1',
 'col__txt__use_idf': True}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.307	0.790	0.571	0.556	0.687	0.658	0.607
recall	0.489	0.677	0.655	0.607	0.658		
f1-score	0.377	0.729	0.610	0.572	0.667		
support	0.064	0.605	0.332				



It's a significant improvement over the baseline, which had a balanced accuracy of ~0.59. Probably much of the improvement is due to using TF*IDF vectors instead of raw token-frequency vectors.

My next step will be to compare the optimized `LogisticRegression` with some alternative models. It's important to try some other options, since there are many other classifiers on the market.

Compare with Naive Bayes

Scikit-Learn recommends using Naive Bayes for text datasets like this one. For the Naive-Bayes classifier, I go with `ComplementNB`, which is supposed to perform better on imbalanced data than `MultinomialNB`. There are not many hyperparameters, but I tune the smoothing parameter 'alpha' thoroughly.

In [55]:
```
nb_grid = {
    "cls__alpha": np.arange(0.0, 1.1, 0.1),
    "cls__fit_prior": [True, False],
    "cls__norm": [True, False],
    "cls": [ComplementNB()],
    **tfidf_grid,
}

# Show size of param space
selection.space_size(nb_grid)
```

Out[55]:
```
n_params       7
n_combos     528
n_folds        5
n_fits      2640
dtype: int64
```

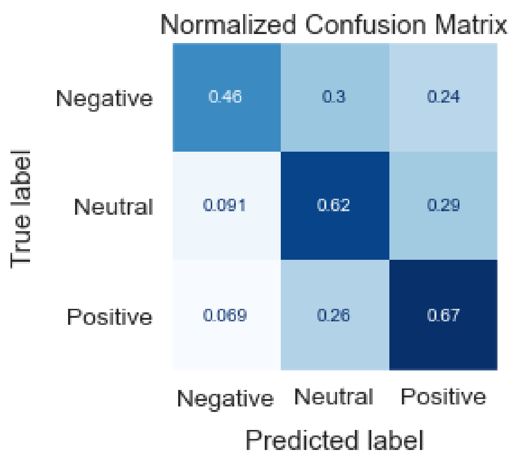
In [56]:
```
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        nb_grid,
        refit=True,
        dst="sweeps/naive_bayes",
        **sweep_params,
    )
```

In [57]:
```
nb_search = selection.load("sweeps/naive_bayes")

# Get best pipeline
nb_pipe = nb_search.best_estimator_
```

```python
# Show best params
display(nb_search.best_params_)

diag.test_fit(nb_pipe, **split_data)
```

```
{'cls': ComplementNB(alpha=0.30000000000000004, norm=True),
 'cls__alpha': 0.30000000000000004,
 'cls__fit_prior': True,
 'cls__norm': True,
 'col__txt__binary': False,
 'col__txt__norm': None,
 'col__txt__use_idf': False}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.273	0.782	0.541	0.532	0.670	0.628	0.585
recall	0.461	0.622	0.671	0.585	0.628		
f1-score	0.343	0.693	0.599	0.545	0.640		
support	0.064	0.605	0.332				



The top-ranking Naive-Bayes pipeline performs worse on balanced accuracy than my logistic regression, but looks decent overall. The Naive-Bayes classifier preferred raw token-frequency vectors to TF*IDF, which is expected. It's also relatively fast.

Compare with SVM

I also train a support vector machine (`LinearSVC`), which Scikit-Learn recommends for text datasets like this one. `LinearSVC` is faster than most SVMs but is still much slower than `ComplementNB` and `LogisticRegression` .

In [58]:

```python
svc_grid = {
    "cls__C": np.geomspace(1e-2, 1e2, 5),
    "cls__fit_intercept": [True, False],
    "cls__loss": ["hinge", "squared_hinge"],
    "cls__penalty": ["l2"],
    "cls": [LinearSVC(class_weight="balanced", random_state=rando, max_iter=1e5)],
    **tfidf_grid,
}

# Show size of param space
selection.space_size(svc_grid)
```

Out[58]:

```
n_params      8
n_combos    240
n_folds       5
```

```
n_fits      1200
dtype: int64
```

In [59]:

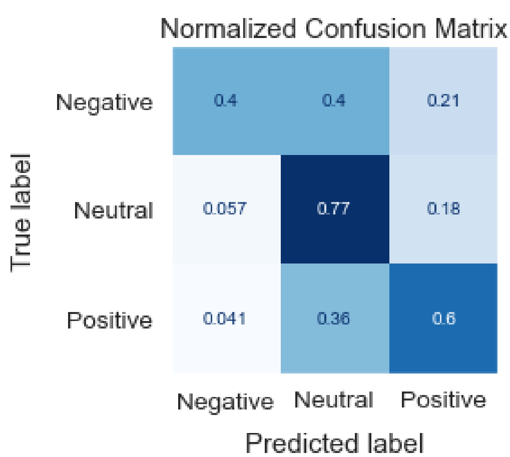
```
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        svc_grid,
        refit=True,
        dst="sweeps/linear_svc",
        **sweep_params,
    )
```

In [60]:

```
svc_search = selection.load("sweeps/linear_svc")

# Get best pipeline
svc_pipe = svc_search.best_estimator_

# Show best params
display(svc_search.best_params_)

diag.test_fit(svc_pipe, **split_data)
```

```
{'cls': LinearSVC(C=10.0, class_weight='balanced', loss='hinge', max_iter=100000.0,
           random_state=RandomState(MT19937) at 0x2D7897E4540),
 'cls__C': 10.0,
 'cls__fit_intercept': True,
 'cls__loss': 'hinge',
 'cls__penalty': 'l2',
 'col__txt__binary': True,
 'col__txt__norm': 'l1',
 'col__txt__use_idf': True}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.346	0.762	0.623	0.577	0.690	0.687	0.588
recall	0.397	0.766	0.599	0.588	0.687		
f1-score	0.370	0.764	0.611	0.582	0.688		
support	0.064	0.605	0.332				



The `LinearSVC` results aren't too impressive to me, although it performs better on (non-balanced) accuracy than the other model types. It's very accurate for Neutral, but it seems to falsely predict Neutral pretty often. Neutral recall is the least interesting and the least important. Both the model's Negative and Positive recall are lower than I'd like.

Select Tokenizer

Now use my `sweep` function to run a grid search over a variety of tokenizers.

The callable tokenizers which I plan to test are in the cell below. The `WhiteSpaceTokenizer` is there as a baseline. NLTK's recommended word tokenizer is `nltk.word_tokenize`, but `nltk.TweetTokenizer` is another obvious one to try.

I've also thrown in some other tokenizers that are on the market. The difference between `nltk.word_tokenize` and `nltk.NLTKWordTokenizer` is that the former uses `nltk.sent_tokenize` before tokenizing words.

In [61]:

```
tokenizers = [
    nltk.word_tokenize,
    nltk.wordpunct_tokenize,
    nltk.TweetTokenizer().tokenize,
    nltk.NLTKWordTokenizer().tokenize,
    nltk.TreebankWordTokenizer().tokenize,
    nltk.ToktokTokenizer().tokenize,
    nltk.WhitespaceTokenizer().tokenize,
    MosesTokenizer().tokenize,
]
tokenizers
```

Out[61]:

```
[<function nltk.tokenize.word_tokenize(text, language='english', preserve_line=False)>,
 <bound method RegexpTokenizer.tokenize of WordPunctTokenizer(pattern='\\w+|[^\\w\\s]+', gaps=Fals
e, discard_empty=True, flags=re.UNICODE|re.MULTILINE|re.DOTALL)>,
 <bound method TweetTokenizer.tokenize of <nltk.tokenize.casual.TweetTokenizer object at 0x000002D7
898559D0>>,
 <bound method NLTKWordTokenizer.tokenize of <nltk.tokenize.destructive.NLTKWordTokenizer object at
0x000002D789855A30>>,
 <bound method TreebankWordTokenizer.tokenize of <nltk.tokenize.treebank.TreebankWordTokenizer obje
ct at 0x000002D789855A90>>,
 <bound method ToktokTokenizer.tokenize of <nltk.tokenize.toktok.ToktokTokenizer object at 0x000002
D789855790>>,
 <bound method RegexpTokenizer.tokenize of WhitespaceTokenizer(pattern='\\s+', gaps=True, discard_e
mpty=True, flags=re.UNICODE|re.MULTILINE|re.DOTALL)>,
 <bound method MosesTokenizer.tokenize of <sacremoses.tokenize.MosesTokenizer object at 0x000002D78
98557F0>>]
```

Here is the parameter grid which will be passed to `GridSearchCV` (after the proper prefix is added by my `sweep` function). In addition to the tokenizer, I will also be testing different token-stemmers and token-markers, since both of these functionalities depend on the tokenizer. The two stemming options built into my `FreqVectorizer` are Porter stemming and Wordnet lemmatization. The former is much faster, because Wordnet lemmatization depends on tagging parts of speech.

In [62]:

```
tokenizer_grid = [
    # Callable tokenizers
    {
        "tokenizer": tokenizers,
        "token_pattern": [None],
        "stemmer": ["porter", "wordnet", None],
        "mark": ["neg", "speech", "neg_split", "speech_split", None],
    },
    # Default regex
    {
        "tokenizer": [None],
        "token_pattern": [r"(?u)\b\w\w+\b"],
        "stemmer": ["porter", "wordnet", None],
        "mark": ["neg", "speech", "neg_split", "speech_split", None],
    },
]
```

```
# Show size of param space
selection.space_size(tokenizer_grid)
```

Out[62]:

```
n_params      4
n_combos    135
n_folds       5
n_fits      675
dtype: int64
```

Word Markers

In addition to its new stemming options, `FreqVectorizer` has word-marking capabilities. It can mark words between a negating term and a punctuation mark with a suffix (`mark='neg'`). For example:

> "I don't like_NEG dolphins_NEG."

It can likewise tag words with their part of speech (`mark='speech'`). For example:

> "Some_DT customers_NNS complained_VBD about_IN the_DT service*NN* .."

If `mark='neg_split'` or `mark='speech_split'` is set, the markers are broken off into separate tokens. The result looks as follows:

> "Some DT customers NNS complained VBD about IN the DT service NN . ."

Since `FreqVectorizer` treats each document as a bag-of-words (i.e. ignores word order), turning the markers into independent tokens can have an interesting effect. Because `nltk.pos_tag` typically marks punctuation with a duplicate punctuation mark, the 'speech' and 'speech_split' settings may produce odd-looking or duplicated punctuation. This looks ugly, but only the computer has to read it.

In [63]:

```
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        tokenizer_grid,
        add_prefix="col__txt__",
        dst="sweeps/tokenizer",
        **sweep_params,
    )
```

And the winners are the regex pattern from `nltk.wordpunct_tokenize` and `MY_STOP` . This regex pattern is also considerably faster than the second best option of `nltk.word_tokenize` .

In [64]:

```
results = selection.load_results("sweeps/tokenizer", drop_dicts=False)

# Remove punctuation for readability
results.tokenizer = results.tokenizer.map(str).map(lang.strip_punct)

# Hide param dicts for display
results.drop(columns="params").head(10).style.bar("mean_score")
```

Out[64]:

	mark	stemmer	token_pattern	tokenizer	mean_fit_time	mean_score	rank_score
0	None	porter	None	function word tokenize at 0x000002D782F270D0	2.625003	0.628654	1

	mark	stemmer	token_pattern	tokenizer	mean_fit_time	mean_score	rank_score
1	neg_split	porter	None	bound method NLTKWordTokenizer tokenize of nltk tokenize destructive NLTKWordTokenizer object at 0x000002D789B99310	2.420400	0.628064	2
2	None	wordnet	None	function word tokenize at 0x000002D782F270D0	4.286799	0.627781	3
3	None	porter	None	bound method NLTKWordTokenizer tokenize of nltk tokenize destructive NLTKWordTokenizer object at 0x000002D789B99310	1.996799	0.626433	4
4	neg_split	wordnet	None	bound method TweetTokenizer tokenize of nltk tokenize casual TweetTokenizer object at 0x000002D789B992B0	2.145999	0.626283	5
5	None	porter	None	bound method TreebankWordTokenizer tokenize of nltk tokenize treebank TreebankWordTokenizer object at 0x000002D789B99340	2.033803	0.625872	6
6	neg_split	None	None	bound method NLTKWordTokenizer tokenize of nltk tokenize destructive NLTKWordTokenizer object at 0x000002D789B99310	2.396201	0.625504	7
7	neg_split	porter	None	bound method TreebankWordTokenizer tokenize of nltk tokenize treebank TreebankWordTokenizer object at 0x000002D789B99340	2.277600	0.625181	8
8	neg_split	porter	None	function word tokenize at 0x000002D782F270D0	3.369200	0.624907	9
9	neg_split	porter	None	bound method TweetTokenizer tokenize of nltk tokenize casual TweetTokenizer object at 0x000002D789B992B0	2.303999	0.624649	10

Fitting the Model

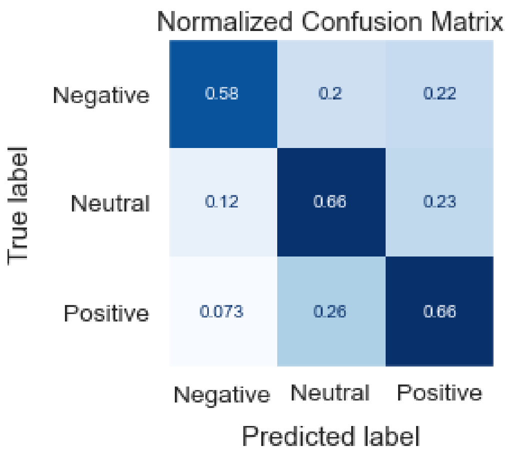
I try some of the top scoring combinations.

In [65]:

```
choice = 1

# Get params from search results and set them
main_pipe.set_params(**results.iloc[choice].params)

# Show what I'm setting
display(results.iloc[choice].params)

diag.test_fit(main_pipe, **split_data)
```

```
{'col__txt__mark': 'neg_split',
 'col__txt__stemmer': 'porter',
 'col__txt__token_pattern': None,
 'col__txt__tokenizer': <bound method NLTKWordTokenizer.tokenize of <nltk.tokenize.destructive.NLTK
WordTokenizer object at 0x000002D789B99310>>}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.282	0.798	0.590	0.556	0.696	0.653	0.633
recall	0.582	0.655	0.662	0.633	0.653		
f1-score	0.380	0.720	0.624	0.574	0.666		
support	0.064	0.605	0.332				



I go with `NLTKWordTokenizer`, Porter stemming, and `mark='neg_split'`. As I suspected, marking negation makes a big difference for Negative recall, and therefore substantially increases balanced accuracy. Again, balanced accuracy is equivalent to macro average recall, or the average of the diagonal on the (normalized) confusion matrix.

I'm glad Porter stemming performed well, because it's much faster than lemmatizing with Wordnet.

Select Text Filters

Next I load up my pre-made parameter grid for text preprocessors. My `FreqVectorizer`, which extends `TfidfVectorizer`, has several additional preprocessing options. It can:

- Decode HTML Entities
  - e.g. `—` becomes '—'
  - `&` becomes '&'
- Strip Punctuation
- Strip Numerals
- Split Up Alphanumeric Sequences
- Force Only Alphanumeric
- Strip Twitter Handles
- Limit Repeating Characters
- Strip Multiple Whitespaces

Like its parent class, it can also strip Unicode accents, or aggressively transliterate Unicode to ASCII.

In [66]:
```
filter_grid = {
    "decode_html_entities": [True, False],
```

```python
    "strip_punct": [True, False],
    "alphanum_only": [True, False],
    "strip_numeric": [True, False],
    "split_alphanum": [True, False],
    "strip_twitter_handles": [True, False],
    "limit_repeats": [True, False],
    "strip_accents": ["unicode", "ascii", None],
    "strip_multiwhite": [True, False],
}

# Show size of param space
selection.space_size(filter_grid)
```

Out[66]:

```
n_params       9
n_combos     768
n_folds        5
n_fits      3840
dtype: int64
```

In [67]:

```python
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        filter_grid,
        add_prefix="col__txt__",
        dst="sweeps/text_filters",
        **sweep_params,
    )
```

In [68]:

```python
# Load with param dicts
results = selection.load_results("sweeps/text_filters", drop_dicts=False)

# Hide dicts for display
results.drop(columns="params").head(10).style.bar("mean_score")
```

Out[68]:

	alphanum_only	decode_html_entities	limit_repeats	split_alphanum	strip_accents	strip_multiwhite	strip_numeric	s
0	False	True	False	False	None	False	False	
1	False	True	True	False	None	False	False	
2	False	True	False	False	None	True	False	
3	False	True	True	False	None	True	False	
4	False	True	False	False	unicode	False	False	
5	False	True	False	False	unicode	True	False	
6	False	True	True	False	unicode	False	False	
7	False	True	True	False	unicode	True	False	
8	False	True	False	False	ascii	False	True	
9	False	True	False	True	ascii	False	True	

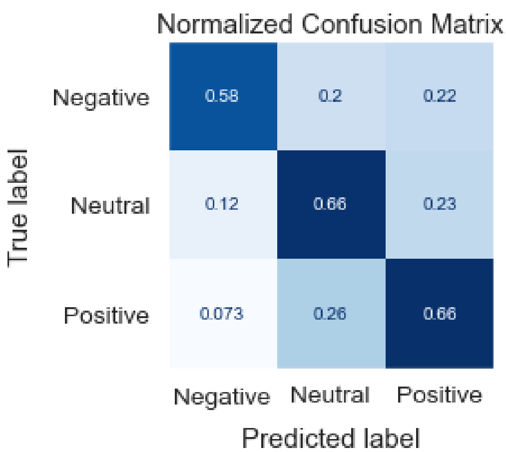
Fitting the Model

I try all the models tied for first and find no difference, so I choose the fastest (row 0). This happens to be the default settings.

In [69]:
```
choice = 0

# Set with best params in `results`
main_pipe.set_params(**results.iloc[choice].params)

# Show the params to confirm
display(results.iloc[choice].params)

diag.test_fit(main_pipe, **split_data)
```

```
{'col__txt__alphanum_only': False,
 'col__txt__decode_html_entities': True,
 'col__txt__limit_repeats': False,
 'col__txt__split_alphanum': False,
 'col__txt__strip_accents': None,
 'col__txt__strip_multiwhite': False,
 'col__txt__strip_numeric': False,
 'col__txt__strip_punct': False,
 'col__txt__strip_twitter_handles': False}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.282	0.798	0.590	0.556	0.696	0.653	0.633
recall	0.582	0.655	0.662	0.633	0.653		
f1-score	0.380	0.720	0.624	0.574	0.666		
support	0.064	0.605	0.332				



I've landed on the default settings, so there's no significant improvement. Still, it was worthwhile to run a search on the text filters.

I have 'decode_html_entities' set True by default on `FreqVectorizer`, because I can't imagine why anyone would want raw HTML entities like '—' getting mangled by the tokenizer. At least that decision was vindicated by this sweep.

Next, I'll try out sets of stopwords and tune other stopword-related parameters.

Select Stopwords

The following are the stopwords sets I plan to test. It includes `MY_STOP`, a short list which I defined earlier. My `FreqVectorizer` is capable of fetching the a number of different stopwords sets given the relevant string argument. That's because it makes use of my `lang.fetch_stopwords` function, which I use below in order to preprocess the stop words before testing them.

In [70]:
```
stop_sets = [
```

```python
    lang.fetch_stopwords("skl_english"),
    lang.fetch_stopwords("nltk_english"),
    lang.fetch_stopwords("gensim_english"),
    lang.fetch_stopwords("skl_english | nltk_english | gensim_english"),
    lang.fetch_stopwords("skl_english & nltk_english & gensim_english"),
    MY_STOP,
    MY_STOP | lang.fetch_stopwords("nltk_english"),
    BRAND_STOP,
    BRAND_STOP | lang.fetch_stopwords("nltk_english"),
    MY_STOP | BRAND_STOP,
    MY_STOP | BRAND_STOP | lang.fetch_stopwords("nltk_english"),
    None,
]

# Display sizes for brevity
[len(x) if x else x for x in stop_sets]
```

Out[70]:
```
[318, 179, 337, 390, 119, 8, 187, 9, 188, 17, 196, None]
```

Preprocessing

I stem or lemmatize the stopwords (if necessary) to match the text preprocessing.

In [71]:
```python
# Get 'stemmer' setting
stemmer = main_pipe["col"].named_transformers_["txt"].get_params()["stemmer"]

# Stem stopwords
if stemmer is not None:
    # Convert frozensets to tuple for token processors, preserving None
    stop_sets = [tuple(x) if x else x for x in stop_sets]

    # Apply Porter stemming
    if stemmer == "porter":
        stop_sets = [lang.porter_stem(x) if x else x for x in stop_sets]
        print("Applied Porter stemming.\n")

    # Apply Wordnet lemmatization
    elif stemmer == "wordnet":
        stop_sets = [lang.wordnet_lemmatize(x) if x else x for x in stop_sets]
        print("Applied Wordnet lemmatization.\n")

    # Raise error if unrecognized stemmer
    else:
        raise ValueError(f"Expected 'porter' or 'wordnet', got {stemmer}.")

pprint(stop_sets[0], compact=True)
```

```
Applied Porter stemming.

('their', 'abov', 'still', 'afterward', 'after', 'first', 'hi', 'alreadi',
 'same', 'those', 'from', 'fill', 'keep', 'nowher', 'nine', 'nobodi', 'amount',
 'therefor', 'due', 'you', 'describ', 'third', 'for', 'sometim', 'further',
 'everyth', 'through', 'thick', 'sincer', 'detail', 'alon', 'each', 'de', 'six',
 'twenti', 'where', 'at', 'side', 'interest', 'amoungst', 'are', 'name',
 'becom', 'thi', 'again', 'ie', 'within', 'somewher', 'sometim', 'yourself',
 'to', 'becom', 'if', 'wherea', 'can', 'me', 'itself', 'whatev', 'anyway', 'be',
 'call', 'mani', 'there', 'toward', 'ourselv', 'over', 'also', 'these', 'her',
 'what', 'veri', 'anyhow', 'nevertheless', 'behind', 'ten', 'much', 'though',
 'besid', 'etc', 'made', 'down', 'three', 'mill', 're', 'thenc', 'they', 'none',
 'whether', 'ever', 'neither', 'thin', 'it', 'other', 'and', 'put', 'see',
 'four', 'own', 'about', 'not', 'otherwis', 'except', 'seem', 'ha', 'himself',
 'herself', 'must', 'take', 'a', 'seem', 'among', 'anywher', 'so', 'almost',
 'cant', 'sever', 'seriou', 'her', 'co', 'els', 'becom', 'therebi', 'toward',
 'seem', 'while', 'would', 'anoth', 'around', 'whenev', 'nor', 'be', 'part',
 'anyth', 'am', 'get', 'throughout', 'it', 'someon', 'thereupon', 'whither',
```

```
'becaus', 'someth', 'wherein', 'few', 'amongst', 'onli', 'never', 'go', 'he',
'back', 'henc', 'therein', 'below', 'fire', 'either', 'even', 'pleas',
'couldnt', 'front', 'we', 'thu', 'on', 'our', 'until', 'yourselv', 'hundr',
'latterli', 'empti', 'under', 'eleven', 'five', 'con', 'alway', 'everywher',
'ltd', 'both', 'then', 'next', 'by', 'eg', 'un', 'cannot', 'of', 'noon',
'hasnt', 'mine', 'wa', 'enough', 'everyon', 'thereaft', 'rather', 'less',
'former', 'whereaft', 'now', 'bill', 'might', 'show', 'your', 'find', 'that',
'well', 'him', 'thru', 'onto', 'how', 'whoever', 'howev', 'had', 'i', 'upon',
'beyond', 'wherebi', 'off', 'such', 'done', 'yet', 'which', 'than', 'dure',
'into', 'no', 'could', 'eight', 'should', 'full', 'our', 'my', 'per', 'who',
'is', 'myself', 'least', 'fifteen', 'herein', 'sinc', 'formerli', 'but',
'whereupon', 'out', 'befor', 'us', 'have', 'some', 'latter', 'whole', 'along',
'onc', 'mostli', 'hereaft', 'besid', 'two', 'herebi', 'as', 'without', 'fifti',
'wherev', 'been', 'seem', 'often', 'last', 'noth', 'inc', 'them', 'too',
'whenc', 'all', 'cri', 'across', 'beforehand', 'between', 'or', 'may',
'perhap', 'meanwhil', 'an', 'inde', 'system', 'whom', 'move', 'found',
'bottom', 'everi', 'in', 'do', 'give', 'whose', 'twelv', 'becam', 'when',
'togeth', 'sixti', 'themselv', 'up', 'although', 'most', 'the', 'other',
'here', 'were', 'will', 'name', 'top', 'elsewher', 'more', 'forti', 'your',
'ani', 'anyon', 'against', 'somehow', 'via', 'moreov', 'she', 'hereupon',
'one', 'with', 'whi')
```

The Sweep

I create the parameter grid and add 'min_df' and 'max_df'. Increasing 'min_df' (minimum document frequency) filters out extremely rare words, while reducing 'max_df' (maximum document frequency) filters out extremely common words.

In [72]:
```
stop_grid = {
    "stop_words": stop_sets,
    "min_df": [1, 5, 10],
    "max_df": np.arange(0.25, 1.25, 0.25),
}
selection.space_size(stop_grid)
```

```
C:\Users\ndgig\anaconda3\envs\learn-env\lib\site-packages\numpy\core\_asarray.py:136: VisibleDeprec
ationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-o
r-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you
must specify 'dtype=object' when creating the ndarray
  return array(a, dtype, copy=False, order=order, subok=True)
```

Out[72]:
```
n_params      3
n_combos    144
n_folds       5
n_fits      720
dtype: int64
```

In [73]:
```
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        stop_grid,
        add_prefix="col__txt__",
        dst="sweeps/stopwords",
        **sweep_params,
    )
```

Now I load up the results.

In [74]:
```
# Load with param dicts
results = selection.load_results("sweeps/stopwords", drop_dicts=False)

# Hide dicts for display
results.drop(columns="params").head(10).style.bar("mean_score")
```

Out[74]:

	max_df	min_df	stop_words	mean_fit_time	mean_score	rank_score
0	1.000000	5	('rt', 'sxsw', 'mention', 'link', 'sxswi', 'austin', 'southbysouthwest', 'america')	1.828000	0.628529	1
1	1.000000	1	None	2.134799	0.628064	2
2	0.750000	1	None	2.045800	0.628031	3
3	1.000000	1	('rt', 'sxsw', 'mention', 'link', 'sxswi', 'austin', 'southbysouthwest', 'america')	2.074999	0.627308	4
4	0.500000	1	None	2.016200	0.626527	5
5	0.750000	5	('ipad', 'googl', 'applesxsw', 'andoid', 'appl', 'app', 'androidsxsw', 'android', 'iphon')	2.040199	0.626516	6
6	0.750000	1	('ipad', 'googl', 'applesxsw', 'andoid', 'appl', 'app', 'androidsxsw', 'android', 'iphon')	2.141198	0.626398	7
7	0.750000	1	('rt', 'sxsw', 'ipad', 'googl', 'applesxsw', 'andoid', 'mention', 'appl', 'link', 'sxswi', 'app', 'androidsxsw', 'android', 'austin', 'iphon', 'southbysouthwest', 'america')	2.006399	0.626090	8
8	1.000000	1	('rt', 'sxsw', 'ipad', 'googl', 'applesxsw', 'andoid', 'mention', 'appl', 'link', 'sxswi', 'app', 'androidsxsw', 'android', 'austin', 'iphon', 'southbysouthwest', 'america')	2.149798	0.625948	9
9	0.750000	1	('rt', 'sxsw', 'mention', 'link', 'sxswi', 'austin', 'southbysouthwest', 'america')	2.036400	0.625331	10

Fitting the Model

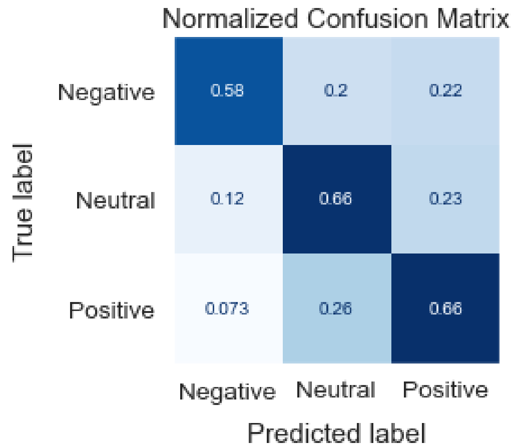
Strangely, the top-ranked settings perform much worse on the actual test set. Thus, I go with the default settings (row 1).

In [75]:

```
choice = 1

# Set with best params in `results`
main_pipe.set_params(**results.iloc[choice].params)

# Show the params to confirm
display(results.iloc[choice].params)

diag.test_fit(main_pipe, **split_data)
```

```
{'col__txt__max_df': 1.0, 'col__txt__min_df': 1, 'col__txt__stop_words': None}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.282	0.798	0.590	0.556	0.696	0.653	0.633
recall	0.582	0.655	0.662	0.633	0.653		
f1-score	0.380	0.720	0.624	0.574	0.666		
support	0.064	0.605	0.332				



Once again, there's no improvement in the model because I landed on the defaults (no stopwords, `max_df=1.0`, `min_df=1`). I've now exhausted all of my text preprocessing options.

In [76]:
```
# Build final analyzer used within `FreqVectorizer`
analyzer = col_xform.named_transformers_["txt"].build_analyzer()

# Analyze `X_train`
X_train.text.head(10).map(analyzer)
```

Out[76]:
```
7610    [#, foursquar, and, @, mention, now, see, #, g...
4742    [ten, percent, of, the, crowd, at, ``, design,...
7077    [appl, to, open, pop-up, shop, at, sxsw, [, re...
8700    [we, interrupt, your, regularly-schedul, #, sx...
7604    [appl, store, #, sxsw, line, is, move, at, the...
1535    [@, mention, {, link, }, you, get, the, #, sxs...
5356    [rt, @, mention, 40, %, of, googl, map, use, i...
354     [tri, to, balanc, the, power, of, power, need,...
8019    [chri, messina, from, googl, drop, knowledg, a...
2099    [such, a, smart, idea, rt, @, mention, w00t, !...
Name: text, dtype: object
```

The lists of tokens above are the final tokens that `FreqVectorizer` actually turns into vectors. As you can see, there are a lot of isolated punctuation marks. Apparently these, along with potential stopwords, actually result in better vectors.

Since I've exhausted my text preprocessing options, I'll run another, more refined, search over the TF*IDF parameters.

Add `VaderVectorizer`

I add a new vectorizer to the mix which performs VADER sentiment analysis on each tweet and returns the polarity scores as vectors.

In [77]:
```
main_pipe["col"].transformers.append(("vad", VaderVectorizer(), "text"))
main_pipe["col"]
```

Out[77]:
```
ColumnTransformer(transformers=[('txt',
                                 FreqVectorizer(mark='neg_split', norm='l1',
                                                stemmer='porter',
                                                token_pattern=None,
                                                tokenizer=<bound method NLTKWordTokenizer.tokenize of <nltk.tokenize.destructive.NLTKWordTokenizer object at 0x000002D789B99310>>,
                                                use_idf=True),
```

```
                                                 'text'),
                                                ('bra', FreqVectorizer(binary=True),
                                                 'brand_terms'),
                                                ('vad', VaderVectorizer(), 'text')])
```

In [78]:

```python
vader_grid = {
    "trinarize": [True, False],
    "compound": [True, False],
    "category": [True, False],
    "norm": ["l1", "l2", "max", None],
}

vader_grid
```

Out[78]:

```
{'trinarize': [True, False],
 'compound': [True, False],
 'category': [True, False],
 'norm': ['l1', 'l2', 'max', None]}
```

In [79]:

```python
if RUN_SWEEPS:
    selection.sweep(
        main_pipe,
        vader_grid,
        add_prefix="col__vad__",
        dst="sweeps/vader_switches",
        **sweep_params,
    )
```

In [80]:

```python
results = selection.load_results("sweeps/vader_switches", drop_dicts=False)
results.drop(columns="params").head(10).style.bar("mean_score")
```

Out[80]:

	category	compound	norm	trinarize	mean_fit_time	mean_score	rank_score
0	True	True	l1	True	10.304001	0.631659	1
1	True	True	l1	False	7.152203	0.630763	2
2	True	False	l1	False	2.940600	0.629380	3
3	True	False	None	False	2.941401	0.629380	3
4	True	False	l1	True	2.798202	0.628581	5
5	True	False	l2	True	2.915001	0.628444	6
6	True	True	l2	True	2.877001	0.627986	7
7	True	False	l2	False	2.966801	0.627402	8
8	True	True	l2	False	2.837803	0.626161	9
9	True	True	None	False	2.768601	0.625747	10

Fitting the Model

Looks like the best settings are to use all scores ('pos', 'neg', 'neu', 'comp'), trinarize them (-1.0, 0.0, 1.0 sign indicators), and apply L1 normalization.

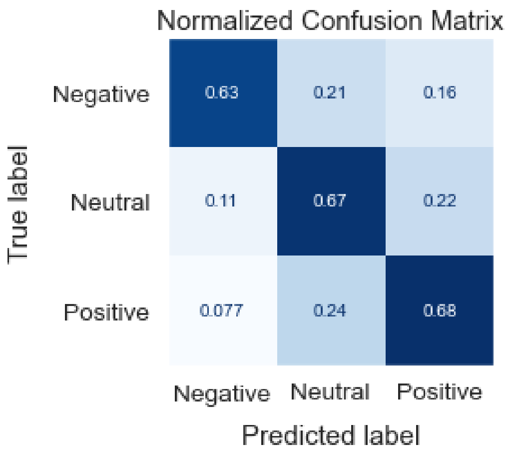
In [81]:

```python
choice = 0

# Get params from search results and set them
```

```python
main_pipe.set_params(**results.iloc[choice].params)

# Show what I'm setting
display(results.iloc[choice].params)

diag.test_fit(main_pipe, **split_data)
```

```
{'col__vad__category': True,
 'col__vad__compound': True,
 'col__vad__norm': 'l1',
 'col__vad__trinarize': True}
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.308	0.811	0.608	0.576	0.712	0.670	0.660
recall	0.631	0.669	0.679	0.660	0.670		
f1-score	0.414	0.734	0.641	0.596	0.683		
support	0.064	0.605	0.332				



It's a dramatic improvement of almost 0.3 in balanced accuracy. Recall is up all around and so is precision. Average precision increased by about 0.2.

Since I have three vectorizers running, my next step is to try feature selection.

Add Feature Selection

In the cell below, I run the `X_train` through `col_xform` (which contains the vectorizers) to remind myself how many features it's putting out.

In [82]:

```python
main_pipe["col"].fit_transform(X_train)
```

Out[82]:

```
<6659x8487 sparse matrix of type '<class 'numpy.float64'>'
	with 162070 stored elements in Compressed Sparse Row format>
```

Currently the model is being trained on ~8,500 features, which is a pretty high number. It's certainly workable, especially since the large vectors are sparse and contain mostly zeros. Nevertheless, perhaps reducing the number of features in a principled manner would improve model performance.

In [83]:

```python
main_pipe["col"].set_params(bra="drop")
main_pipe.steps.insert(-1, ("sel", SelectPercentile(percentile=30)))
main_pipe
```

Out[83]:

```
Pipeline(steps=[('col',
```

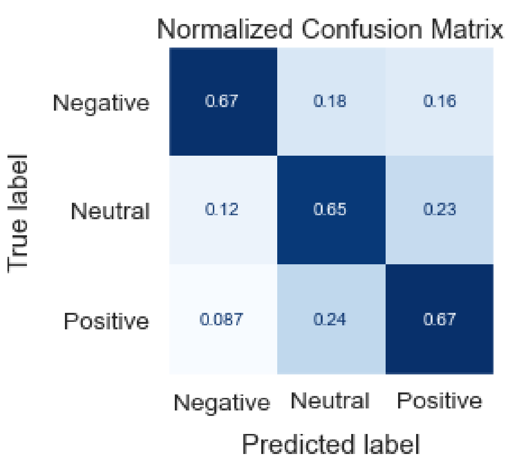
```
                  ColumnTransformer(transformers=[('txt',
                                                   FreqVectorizer(mark='neg_split',
                                                                  norm='l1',
                                                                  stemmer='porter',
                                                                  token_pattern=None,
                                                                  tokenizer=<bound method NLTKWordTo
kenizer.tokenize of <nltk.tokenize.destructive.NLTKWordTokenizer object at 0x000002D789B99310>>,
                                                                  use_idf=True),
                                                   'text'),
                                                  ('bra', 'drop', 'brand_terms'),
                                                  ('vad',
                                                   VaderVectorizer(norm='l1',
                                                                   trinarize=True),
                                                   'text')])),
                 ('sel', SelectPercentile(percentile=30)),
                 ('cls',
                  LogisticRegression(C=10.0, class_weight='balanced',
                                     fit_intercept=False, max_iter=1000,
                                     multi_class='multinomial',
                                     random_state=RandomState(MT19937) at 0x2D788219640,
                                     solver='newton-cg'))])
```

After some playing around, I find that dropping the 'brand_terms' vectorizer entirely and then keeping only features in the top 30th percentile of ANOVA F-values improves the model. The brand terms probably don't contribute much novel information, since brand-related terms are already features of the TF*IDF vectors.

In [84]:

```
diag.test_fit(main_pipe, **split_data)
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.295	0.810	0.597	0.567	0.707	0.657	0.662
recall	0.667	0.649	0.670	0.662	0.657		
f1-score	0.409	0.720	0.631	0.587	0.671		
support	0.064	0.605	0.332				



Negative recall is now a high 0.67, which is about equal to Positive recall. Balanced accuracy has gone up slightly as a result.

In [85]:

```
main_pipe[:-1].fit_transform(X_train, y_train)
```

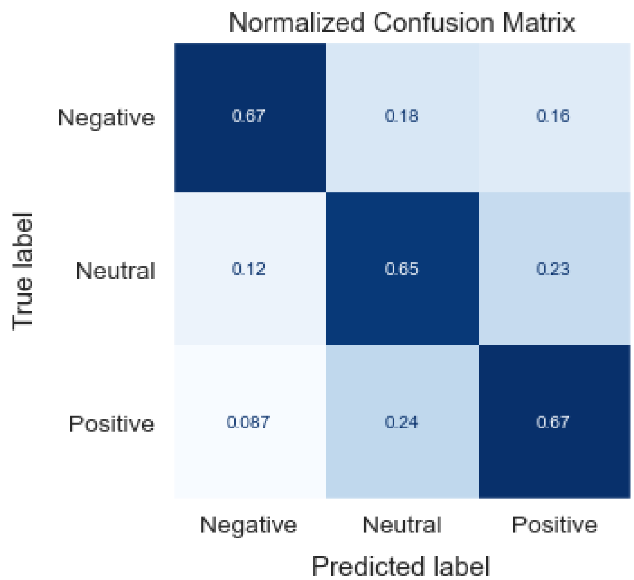
Out[85]:

```
<6659x2543 sparse matrix of type '<class 'numpy.float64'>'
	with 127923 stored elements in Compressed Sparse Row format>
```

Now the model is being trained on a mere ~2,500 features. Too bad there aren't any more parameter sweeps to run, because they would be noticeably faster.

In [86]:

```
fig = diag.classification_plots(
    main_pipe.fit(X_train, y_train), X_test, y_test, pos_label="Positive"
)
fig.savefig(normpath("images/final_confusion_matrix.svg"))
```



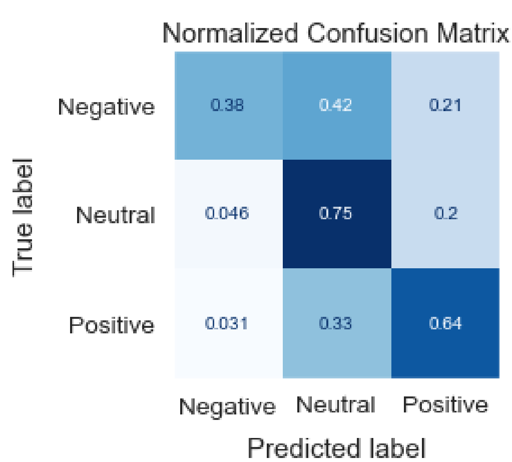
Final Model

After considerable experimentation and engineering, the final model's balanced accuracy of 0.66 vastly surpasses that of the baseline (0.59). I've reproduced the baseline below, for reference.

In [87]:

```
diag.test_fit(baseline, **split_data)
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.384	0.770	0.609	0.588	0.692	0.690	0.589
recall	0.376	0.751	0.640	0.589	0.690		
f1-score	0.380	0.760	0.624	0.588	0.691		
support	0.064	0.605	0.332				



The `FreqVectorizer` assigned to the text is one of the two most important components in the pipeline (along with the classifier). I put considerable effort into trying to optimize its preprocessing parameters. The most important decisions were the choice of tokenizer, stemmer, and word-markers. I chose the high-scoring combination of `nltk.NLTKWordTokenizer`, Porter stemmer, and negation markers. Marking negation means that words which fall between a negating word and sentence punctuation get marked 'NEG'. Ordinarily, 'NEG' markers are joined with underscore to the words they mark. However, I took the unorthodox approach of placing the 'NEG' markers directly in the bags-of-words as independent tokens.

With regard to filtering, the text is simply lowercased and html entities are decoded into symbols. No stopwords were chosen. As it turns out, many stopwords and punctuation symbols are associated with the Neutral class.

In [88]:
```
main_pipe["col"]
```

Out[88]:
```
ColumnTransformer(transformers=[('txt',
                                 FreqVectorizer(mark='neg_split', norm='l1',
                                                stemmer='porter',
                                                token_pattern=None,
                                                tokenizer=<bound method NLTKWordTokenizer.tokenize
of <nltk.tokenize.destructive.NLTKWordTokenizer object at 0x000002D789B99310>>,
                                                use_idf=True),
                                 'text'),
                                ('bra', 'drop', 'brand_terms'),
                                ('vad',
                                 VaderVectorizer(norm='l1', trinarize=True),
                                 'text')])
```

Originally I had binary occurrence vectors for regex-extracted brand terms. At the end, I discovered that these brand terms were no longer helping the model. Roughly the same information is contained in the text TF*IDF vectors.

The `VaderVectorizer` was a late addition to the pipeline, and it brought a substantial increase in balanced accuracy. I opted to trinarize the VADER scores, i.e. reduce them to ternary sign indicators (-1.0, 0.0, 1.0).

In [89]:
```
main_pipe[1:]
```

Out[89]:
```
Pipeline(steps=[('sel', SelectPercentile(percentile=30)),
                ('cls',
                 LogisticRegression(C=10.0, class_weight='balanced',
                                    fit_intercept=False, max_iter=1000,
                                    multi_class='multinomial',
                                    random_state=RandomState(MT19937) at 0x2D788219640,
                                    solver='newton-cg'))])
```

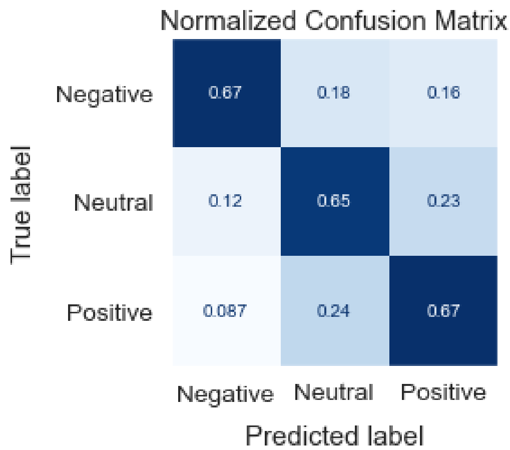
The addition of the `SelectPercentile` estimator was an important development because it reduced the ultimate vector size from ~8,500 to ~2,500. This resulted in a cleaner, more accurate, and more easily interpretable model.

And of course, the `LogisticRegression` itself is an essential component. It significantly outperformed the Naive Bayes and Linear SVM models I created. Plus, unlike `ComplementNB`, `LogisticRegression` is able to handle negative values. This allows for the addition of VADER vectors to the mix.

In [90]:
```
diag.test_fit(main_pipe, **split_data)
```

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
precision	0.295	0.810	0.597	0.567	0.707	0.657	0.662
recall	0.667	0.649	0.670	0.662	0.657		
f1-score	0.409	0.720	0.631	0.587	0.671		

	Negative	Neutral	Positive	macro avg	weighted avg	accuracy	bal accuracy
support	0.064	0.605	0.332				



The final model has both high positive recall (0.67) and high negative recall (0.67). The high Negative recall is particularly impressive given the extremely low support (~6%) for the Negative class. The lackluster Neutral recall is not too worrisome, because the Neutral class is the least important. The Positive and Negative classes offer the most interesting material for Apple's market research.

Refit with Final Parameters

In [91]:

```
main_pipe.fit(X, y)
```

Out[91]:

```
Pipeline(steps=[('col',
                 ColumnTransformer(transformers=[('txt',
                                                  FreqVectorizer(mark='neg_split',
                                                                 norm='l1',
                                                                 stemmer='porter',
                                                                 token_pattern=None,
                                                                 tokenizer=<bound method NLTKWordTo
kenizer.tokenize of <nltk.tokenize.destructive.NLTKWordTokenizer object at 0x000002D789B99310>>,
                                                                 use_idf=True),
                                                  'text'),
                                                 ('bra', 'drop', 'brand_terms'),
                                                 ('vad',
                                                  VaderVectorizer(norm='l1',
                                                                  trinarize=True),
                                                  'text')])),
                ('sel', SelectPercentile(percentile=30)),
                ('cls',
                 LogisticRegression(C=10.0, class_weight='balanced',
                                    fit_intercept=False, max_iter=1000,
                                    multi_class='multinomial',
                                    random_state=RandomState(MT19937) at 0x2D788219640,
                                    solver='newton-cg'))])
```

Interpretation

The first order of business is to label the coefficients.

In [92]:

```
feat_names = main_pipe["col"].get_feature_names()
feat_names = np.array(feat_names)

# Slice names with boolean mask from 'sel'
feat_names = feat_names[main_pipe["sel"].get_support()]
```

```
classes = main_pipe["cls"].classes_
coef = pd.DataFrame(main_pipe["cls"].coef_, columns=feat_names, index=classes).T

# coef.rename({"bra__": "bra__none"}, inplace=True)
coef.sort_values("Negative", ascending=False)
```

Out[92]:

	Negative	Neutral	Positive
txt__headach	10.893108	-5.545003	-5.348105
txt__fail	9.962107	-5.850980	-4.111127
txt__fade	9.024025	-5.084594	-3.939431
txt__long	8.728936	-3.996980	-4.731956
txt__crash	7.603669	-3.203684	-4.399985
...	...	...	...
txt__{	-6.090938	4.919628	1.171311
txt__}	-6.098581	4.995304	1.103277
txt__link	-6.279301	4.662615	1.616686
txt__at	-6.377992	4.972304	1.405688
txt__free	-6.439906	4.948811	1.491095

2906 rows × 3 columns

Top 25 Overall

Then I examine the 25 coefficients with the largest magnitude.

In [93]:

```
top25 = coef.abs().max(axis=1).sort_values().tail(25).index
top25
```

Out[93]:

```
Index(['txt__batteri', 'txt__link', 'txt__too', 'txt__at', 'txt__free',
       'txt__it', 'txt__love', 'txt__whi', 'txt__great', 'txt__becaus',
       'txt__iphon', 'txt__ridic.', 'txt__&', 'txt__delet', 'txt__again',
       'txt__button', 'txt__ipad', 'txt__crash', 'txt__long', 'txt__app',
       'txt__fade', 'txt__!', 'txt__cool', 'txt__fail', 'txt__headach'],
      dtype='object')
```

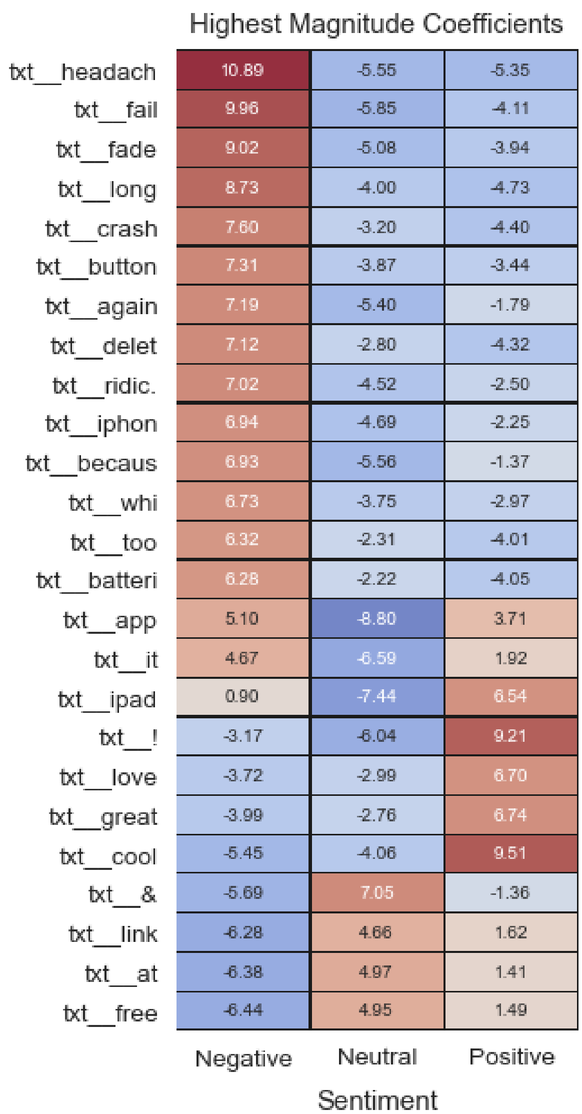
Most of the top 25 coefficients are from the TF*IDF word vectors, unsurprisingly. As predicted, '!' shows up as a top positive coefficient. Another notable top positive term is 'ipad'. Punctuation and very common stopword-like words are related to 'Neutral'.

In [94]:

```
fig, ax = plt.subplots(figsize=(4, 10))
hm_style = dict(plotting.HEATMAP_STYLE)
del hm_style["square"]

sns.heatmap(
    coef.loc[top25].sort_values("Negative", ascending=False),
    ax=ax,
    square=False,
    **hm_style,
)

ax.set(xlabel="Sentiment")
```

```
ax.set_title("Highest Magnitude Coefficients", pad=10)
plotting.save(fig, "images/top25_coef.svg")
```

Out[94]: 'images\\top25_coef.svg'

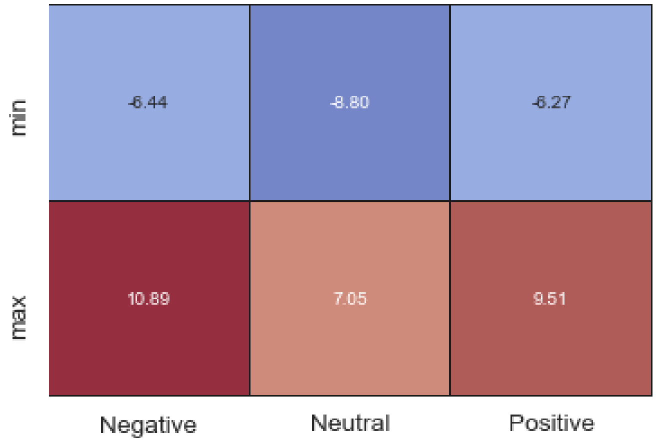


Interesting that the largest overall coefficient is for 'Negative'. Also the maxima are greater in magnitude than the minima.

In [95]:

```
sns.heatmap(
    coef.agg(["min", "max"]),
    square=False,
    **hm_style,
)
```

Out[95]: <AxesSubplot:>



I create a function for grabbing and formatting subsets of the coefficients.

In [96]:

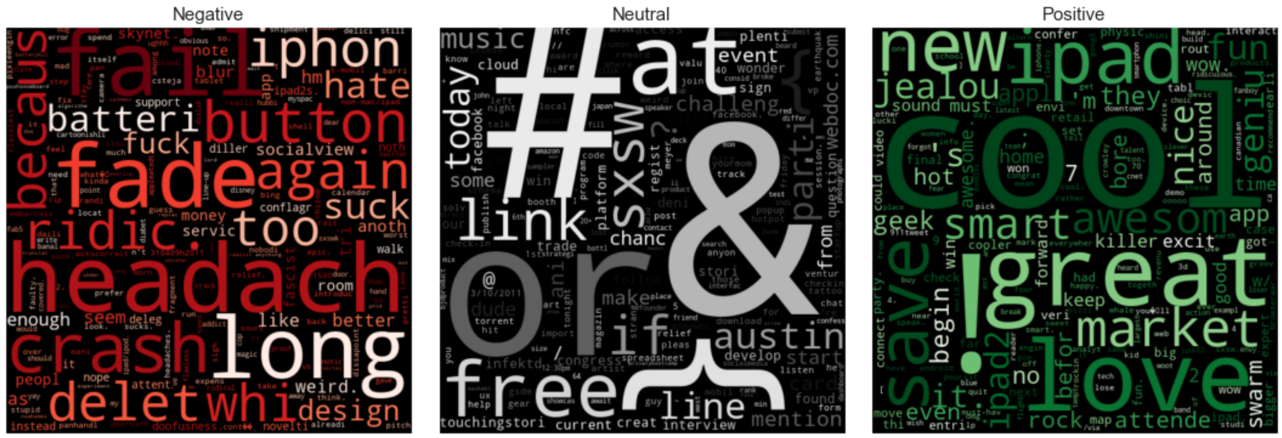
```
def get_coefs(
    prefix,
    index_name,
    coef=coef,
    titlecase=True,
    icase=False,
):
    data = coef.filter(regex=fr"^{prefix}__", axis=0)

    # Remove prefix
    data.index = data.index.str.replace("\w+__", "", regex=True)

    # Make snake_case titlecase
    data.index.name = index_name
    if titlecase:
        data = utils.title_mode(data)
        if icase:
            data.index = data.index.str.replace("Ip", "iP")
    return data.sort_values("Positive")
```

TF*IDF Words

The TF*IDF features were the most influential overall. While the unigram term coefficients from the model are not *completely* useless for brand-related research, they are too coarse-grained and simplistic. See my EDA notebook (exploratory.ipynb) for a deeper brand-related examination of TF*IDF keywords.

In [97]:

```
text_coef = get_coefs("txt", "Text", titlecase=False)

fig = plotting.wordcloud(
    text_coef,
    cmap=dict(Negative="Reds", Neutral="Greys", Positive="Greens"),
    size=(5, 5),
    random_state=rando,
)
fig.savefig(normpath("images/txt_coef_wordclouds.svg"), bbox_inches="tight")
```



The terms in both the 'Positive' and 'Negative' wordclouds make good sense, and many of them such as 'iphon' and 'batteri' show up in the EDA wordclouds. The Neutral category is associated with punctuation and very common (stopword-like) words. That explains why no stopwords were selected.

I create a color palette for the three classes and a function for making positive vs. negative coefficient plots.

In [98]:

```
emo_pal = dict(Negative="r", Neutral="gray", Positive="g")
emo_pal
```

Out[98]:

```
{'Negative': 'r', 'Neutral': 'gray', 'Positive': 'g'}
```

In [99]:

```
def pos_neg_catplot(
    coefs,
    name=None,
    drop_neutral=True,
    palette=emo_pal,
    col_wrap=4,
    sup_y=1.05,
    annot_dist=0.15,
    annot_pad=0.025,
    height=3,
):
    if drop_neutral:
        coefs = coefs.drop("Neutral", axis=1)

    # Plot bars on FacetGrid
    g = sns.catplot(
        data=coefs.reset_index(),
        col=coefs.index.name or "index",
        col_wrap=col_wrap,
        kind="bar",
        palette=palette,
        height=height,
    )

    # Annotate
    plotting.annot_bars(g.axes, orient="v", dist=annot_dist, pad=annot_pad)

    # Add horizontal y=0 line
    for ax in g.axes:
        ax.axhline(0, color="k", lw=1, alpha=0.7)

    # Set Axes titles and ylabels
    g.set_titles("{col_name}")
    g.set_ylabels("Importance")

    # Create overall title
```

```
    if name is None:
        title = "Feature Importances"
    else:
        title = f"Importance of {name}"
    g.fig.suptitle(title, y=sup_y, fontsize=16)
    return g
```

In [100…

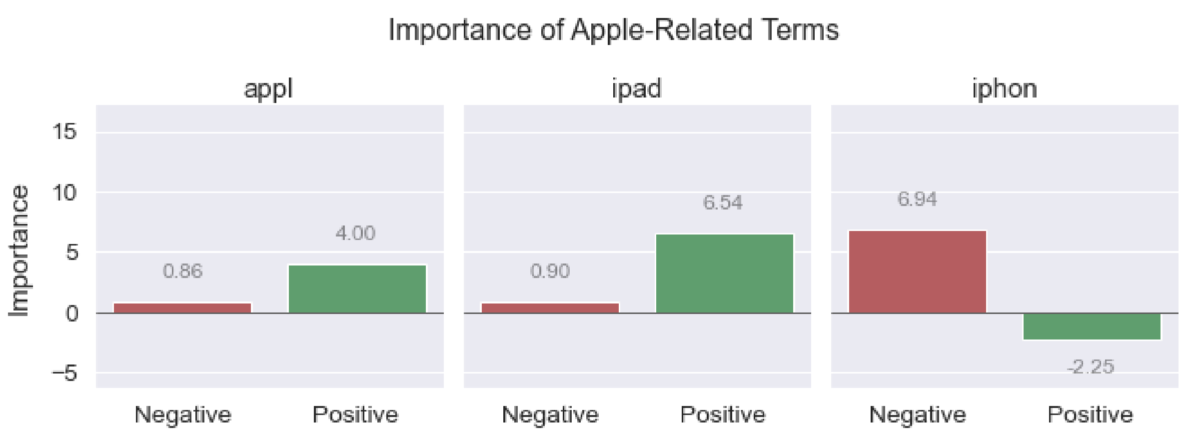
```
apple_coef = text_coef.loc[text_coef.index.isin({"appl", "ipad", "iphon"})]
apple_coef
```

Out[100…

Text	Negative	Neutral	Positive
iphon	6.938021	-4.687938	-2.250084
appl	0.861356	-4.859489	3.998133
ipad	0.899048	-7.441400	6.542352

In [101…

```
g = pos_neg_catplot(
    apple_coef.sort_index(),
    name="Apple-Related Terms",
    col_wrap=3,
    annot_dist=3,
    annot_pad=0.15,
    sup_y=1.1,
)

g.savefig(normpath("images/apple_tfidf_terms.svg"))
```



Interestingly, 'iphon' has a very strong association with Negative, whereas 'ipad' has a very strong association with Positive. There were a lot of complaints about the iPhone's battery life and AT&T's unreliable service.

VADER Valence

Here are the VADER coefficients. They are relatively large, as expected. Adding VADER vectors to the mix proved to be a good idea.

In [102…

```
vad_coef = coef.filter(like="vad__", axis=0)
vad_coef.index = vad_coef.index.str.replace("vad__", "")
vad_coef = utils.title_mode(vad_coef)
vad_coef
```

Out[102...

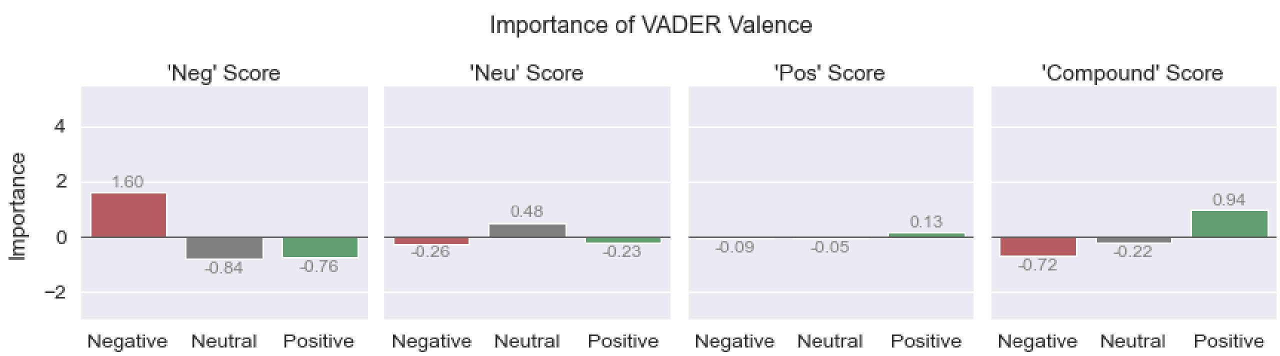
	Negative	Neutral	Positive
Neg	1.602566	-0.840775	-0.761790
Neu	-0.255241	0.484427	-0.229186
Pos	-0.087879	-0.047055	0.134934
Compound	-0.719995	-0.216964	0.936959

In [103...

```
g = pos_neg_catplot(
    vad_coef,
    name="VADER Valence",
    col_wrap=4,
    annot_dist=0.45,
    annot_pad=0.1,
    sup_y=1.1,
    drop_neutral=False,
)

g.set_titles("'{col_name}' Score")
```

Out[103...

```
<seaborn.axisgrid.FacetGrid at 0x2d78cf9b670>
```



The two most important VADER features were 'Neg' and 'Compound'. Unsurprisingly, 'Neg' had a strong association with Negative and a strong inverse association with Positive. 'Compound' is a summary of the other three scores which is enhanced with additional rules. It's not surprising that it had such a robust association with Positive and an inverse association with Negative. 'Neu' and 'Pos' had lackluster importance.

Recommendations

See exploratory.ipynb for the investigation which led to my recommendations.

1. Try to shake your authoritarian image by ostensibly allowing end-users more freedom.

People like that Apple products just work out of the box, but they find your paternalistic approach to managing your products off-putting. **Send the message** that when you buy an Apple product, you are free to do what you want with it. Keep control over the most important things, but relinquish control over the less important things. Make people feel like they have the freedom to customize your products as they see fit. Make some concessions to placate the majority, while allowing the elite techno-snobs to continue complaining on the fringe.

2. Do something to improve the iPhone's battery life and turn it into a marketing campaign.

There were a lot complaints about the iPhone's battery life. One user suggested that their Blackberry was doing much better. There were also complaints about #batterykiller apps which use geolocation in the background. If you made a big publicized effort to increase the iPhone's battery life, that would get people excited.

3. Open another temporary popup store for your next big product launch.

The iPad 2 popup store was a roaring success, and people couldn't stop talking about it. Terms like 'shiny new', 'jealous', and 'cool technology' were closely associated with the iPad 2 and popup store.

Future Work

Stacking Classifiers

After experimenting a little with Scikit-Learn's `StackingClassifier`, it's become clear that I could use it to develop a more accurate final model. The `StackingClassifier` trains several classifiers on the data and then trains a final classifier on the concatenated output of those classifiers. It also allows you to pass the training data to your final estimator, so the final estimator is trained both on prior data and the predictions of the classifier ensemble.

Sophisticated Vectorization

I experimented some with Doc2Vec, a sophisticated unsupervised document vectorization algorithm, but didn't find it to offer any advantage over `FreqVectorizer` when trained on this small dataset. It proved to be slower, much more complicated, and much less interpretable. However, if trained on a large corpus of tweets, and then used to predict vectors for the present dataset, it could prove to be better than TF*IDF vectorization. Even if the Doc2Vec vectors didn't turn out to be better than the TF*IDF vectors, they could potentially augment them. A Doc2Vec model trained on a large corpus would probably contribute **novel information**.

Conclusion

I created an accurate model, at around 0.66 balanced accuracy. The dataset is small, noisy, and not particularly well labeled. Nevertheless, I'm confident that I can increase the accuracy by stacking classifiers. I'd also like to try alternative methods of vectorization, but I'm not as confident that it will improve the model.

Through interpreting my model and conducting a brief exploratory analysis in exploratory.ipynb, I arrived at three recommendations. First, you should publicly relinquish a small amount of control over your products to send the message that you care about individual freedom (and aren't a "fascist company"). Second, you should improve the iPhone's battery life and turn that into a rallying point for a marketing campaign. People are really concerned about the iPhone's battery life. Third, you should repeat the temporary popup store for your next big product launch. There was an overwhelming amount of chatter about the iPad 2 popup store.

In [ ]: