Supplementary Document for Classification of Fall Out Boy Eras

Shifra L. Isaacs, Joseph Yudelson, Dr. Endre Boros

1 Table of Contents

2 Appendices

I Tools Used

- Code Blocks Extension
- Git / GitHub
- Jupyter Notebook / Jupyter Lab
- LyricsGenius
- MatPlotLib
- NumPy
- Pandas
- Pip (Python Install Package)
- Python 3.10+
- Python Regular Expressions
- PyTorch
- SciKit Learn
- Seaborn
- Spotipy
- Statsmodels

II Data Extraction

Notebook: notebooks/FOB_Extract.ipynb

In order to collect data for our regression model, we pulled FOB's musical data from Spotify and their lyrical data from LyricsGenius. I exported the dataset throughout these stages; although the export steps are omitted from this paper, they can be found on GitHub here.

We will use Spotipy, a lightweight Python library that wraps around Spotify's Web API, to access Spotify data. See Appendix A below for a description of the relevant Spotify audio features used in this project.

To install Spotipy and LyricsGenius, I used the following commands in a Jupyter notebook:

```
! pip install spotipy -q
! pip install lyricsgenius -q
```

Note: The -q flag here makes the command output less verbose.

Once Spotipy and LyricsGenius were installed, I moved onto library imports:

```
import re
import pandas as pd
import numpy as np
import matplotlib
from matplotlib import pyplot as
plt
%matplotlib inline
import seaborn as sns
from collections import Counter
from statsmodels.stats.weightstats
import ztest as ztest
```

I imported the necessary Spotipy and LyricsGenius dependencies separately to keep things organized:

```
import spotipy
from spotipy.oauth2 import Spoti-
fyClientCredentials
import spotipy.util as util
import lyricsgenius as lg
```

I generally use the following code to display the entire dataframe by setting all the display limits in Pandas to None:

```
# Display entire dataframe
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
```

Now that all our tools were imported, we set up the Spotipy client information. You will need to create your own Spotify Client ID, Client Secret, and User Name to run the entire notebook.

```
# Spotipy-specific variables
cid = 'Your Client ID here'
secret = 'Your Client secret here'
user = 'Your Client user name here'
scope = 'user-library-read playlist-modify-public playlist-read-private'
redirect_uri = 'http://google.com'
client_credentials_manager = SpotifyClientCredentials(client_id = cid, client_secret = secret)
sp = spotipy.Spotify(client_credentials_manager = client_credentials_manager)
```

We also needed to set up LyricsGenius with an access token. You will need to create your own access token to run the entire notebook.

```
# Genius-specific variables
genius_access_token = 'Your Genius access token here'
genius_obj = lg.Genius(genius_access_token)
```

Now, we were ready to leverage Spotipy and LyricsGenius in order to analyze FOB's Music. I created a playlist containing all 141 Fall Out Boy songs that were on Spotify as of June 2022. See Appendix B for information about the song choices.

I built a function to extract Spotify data from each song in the playlist, including its title, release date, and audio features:

```
def get_features(username, playlist_id):
    results = sp.user_playlist_tracks(username, playlist_id)
    songs = results['items']

    while results['next']:
        results = sp.next(results)
        songs.extend(results['items'])

    ids = []
    names = []
    dates = []

    for i in range(len(songs)):
        if songs[i]['track'] and songs[i]['track']['id'] != None:
            names.append(songs[i]['track']['name'])
            ids.append(songs[i]['track']['id'])
            dates.append(songs[i]['track']['album']['release_date'])
    ids = list(filter(None, ids))

    print(f'Found {len(ids)} songs.')

    features = []
    for id_idx, id_val in enumerate(ids):
        print(f'  Processing song
```

```
#{id_idx+1}: {names[id_idx]}
', end = '\r')
        audio_features = sp.au-
dio_features(id_val)

        for index, feature in enu-
merate(audio_features):
            features.append(fea-
ture)

    features = list(filter(None,
features))
    print(f'\r\nFinished pro-
cessing {len(ids)} songs.')
    df = pd.DataFrame(features)
    df['title'] = names
    df['release_date'] = dates
    return df
```

Note: The many spaces at the end of the 'Processing song' print statement are a cosmetic change that allows longer titles to be displayed as feedback.

The following line of code executed the function to pull music data:

```
fob_songs = get_fea-
tures('shiffytali',
'0ubKnC9ctDVxlGbF5jICvq')
```

Then, it was time to extract the Genius lyrics that correspond to each song in the FOB playlist. The goal behind this extraction was to count the number of total words and the number of unique words for each song. I had a theory that pre-hiatus songs contained higher unique word counts, as FOB's older music tends to be more lyrically complex than their newer material.

After spot-checking a sample of FOB lyric pages on Genius.com, I immediately noticed a glaring issue: the pages were full of sub-headings that indicated the beginnings of verses, choruses, pre-choruses, etc:

[Verse 1]
You are a brick tied to me that's dragging me down
Strike a match and I'll burn you to the ground
We are the jack-o-lanterns in July
Setting fire to the sky

"The Phoenix" Lyrics (SOURCE)

These sub-headings were enclosed in square brackets, and thus easily identifiable by a regular expression (RegEx). I knew that before I programmed the actual Genius extract, I needed to build a RegEx to eliminate these subheadings, which could potentially fill my dataset with useless information.

I noticed that there were several scenarios to account for once the lyrics were evaluated word-by-word: a lone left bracket, a lone right bracket, both brackets, and a few edge cases that emerged. My RegEx would replace all of these bracketed phrases with an empty string so that they wouldn't be counted when I analyzed the lyrics.

I built a regular expression to catch all of these cases, and incorporated the pattern into a Genius extract function to eliminate useless sub-heading information:

```
def get_lyrics(song_list, artist):
    total_words = []
    unique_words = []
    pattern = r'(\[?([\w\-
]+)\])|(\[([\w\-]+)\]?)|(\[([\w\-
]+)\])|.*\[.*]?|.*\[?.*]'

    for idx, song in enumer-
ate(song_list):
        lyrics = ge-
nius_obj.search_song(title = song,
artist = artist).lyrics
        # Substitute any useless
information with an empty string
        lyrics = re.sub(pattern,
'', lyrics)
```

```
        count = Counter(lyr-
ics.split())

        total = sum(count.val-
ues())
        unique = len(count)
        total_words.append(total)
        unique_words.ap-
pend(unique)

    print('Processed all songs')
    return total_words,
unique_words
```

This function used LyricsGenius to fetch all the song lyrics in one string. My RegEx then eliminated the sub-headings. Finally, I split the lyrics string into a list and stored it in a Counter object, which counted the frequency of each word. The sum of the words in this Counter object is the total word count in the song, and its length is the unique word count.

After running the function a few times, I noticed that some of the song titles were different on Spotify and Genius. These titles confused the Genius function and pulled incorrect word counts. To fix this problem, I modified the broken titles by index in the dataframe to ensure the correct word counts were extracted:

```
# Note titles that confuse the
get_lyrics function
manually = {78: "Beat It (Michael
Jackson Cover)", 115: "From Now On
We Are Enemies",
            116: "Yule Shoot Your
Eye Out", 132: "I Wan'na Be Like
You (The Monkey Song)",
            139: "Ghostbusters
(I'm Not Afraid)"}

# Replace with titles that will
work with the get_lyrics function
for idx, title in manu-
ally.items():
    fob_songs.at[idx, "title"] =
title
```

It was time to run the **get_lyrics** function and create two new columns from its output:

```
fob_songs['total_words'],
fob_songs['unique_words'] =
get_lyrics(fob_songs['title'],
'Fall Out Boy')
fob_songs.head()
```

III DATA CLEANING AND VALIDATION

This step combined data cleaning, data munging, data validation, and overall gaining a deeper understanding of the data.

First, I deleted the columns that wouldn't be helpful in my model.

```
# Remove irrelevant columns
fob_songs = fob_songs.drop(labels
= ['type','id', 'uri',
'track_href', 'analysis_url'],
axis = 'columns')
```

Then, I moved the title column to the left-most index.

```
first_column = fob_songs.pop('ti-
tle')
fob_songs.insert(0, 'title',
first_column)
```

I needed to convert the duration column of the Spotify features, which was given in milliseconds. I converted the column values from microseconds to minutes using division:

```
# Replace duration in ms with du-
ration in minutes
fob_songs['duration_min'] =
round((fob_songs['duration_ms'] /
```

```
60000), 2)
fob_songs = fob_songs.drop(['dura-
tion_ms'], axis = 1)
```

These values were not presented in a minutes:seconds format, but instead minutes:fractions of a minute. For example, the first song listed was "Honorable Mention," which is 3 minutes and 25 seconds long. My dataframe value for "Honorable Mention" in the *duration_min* column was 3.43 because 43/60 = ~25 seconds. This value was displayed as a fraction of a minute, *not* as a count of seconds. This was explicitly clarified in the notebook to ensure that the column is not misunderstood.

It was time to assign the target variable: the designation of pre-hiatus or post-hiatus class. I initialized a column containing all pre-hiatus values. I then reassigned the post-hiatus rows based on the release date of each song.

As a general rule, songs released after 2009 (the year the hiatus began) were classified as post-hiatus. However, one edge case emerged: FOB's post-hiatus cover of Elton John's "Saturday Night's Alright for Fighting". This song was incorrectly documented by Spotipy as a 1973 release in accordance with the release date of the original Elton song.

```
fob_songs['class'] = 'pre-hiatus'
# set default value as pre-hiatus

# Designate post-hiatus songs
for idx, date in enumer-
ate(fob_songs['release_date']):
    year = int(date[:4])
    if (year == 1973) or (year >
2009):
        # Accounts for Elton John
cover labeled as 1973 but recorded
in post-hiatus era
        fob_songs.at[idx, 'class']
= 'post-hiatus'
```

The final exported data frame can be found on GitHub HERE. The features in this dataset include: 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness', 'liveness', 'valence', 'tempo', 'duration_min', 'total_words', 'unique_words', 'key','mode', 'time_signature', and 'class'.

IV Exploratory Data Analysis and Visualization

Notebook: notebooks/FOB_EDA.ipynb

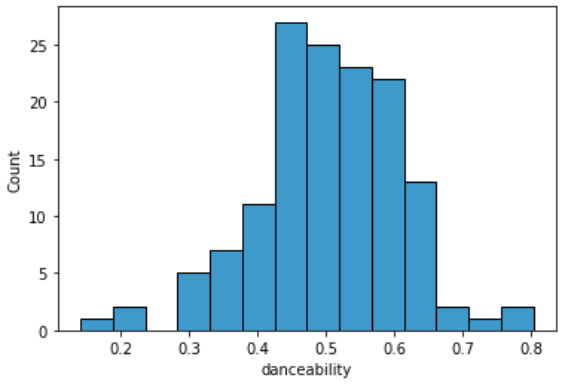
In order to visualize the data effectively, I created lists containing subsets of the feature columns:

```
# Numerical- instrumentalness is
left out because it doesn't graph
well
numerical = ['danceability', 'en-
ergy', 'loudness', 'speechiness',
'acousticness', 'liveness', 'va-
lence', 'tempo', 'duration_min',
'total_words', 'unique_words']
categorical = ['key','mode',
'time_signature']
target = fob_songs['class']
```

I then graphed histograms for all the numerical features to see their respective distributions:

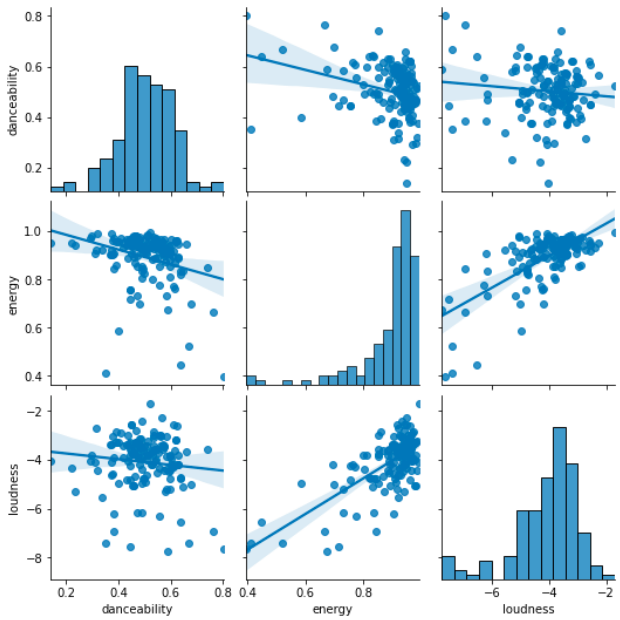
```
for i, feature in enumerate(numer-
ical):
    plt.figure(i)
    sns.histplot(data=fob_songs, x
= feature)
    plt.show()
```

The histogram for danceability looks like this:

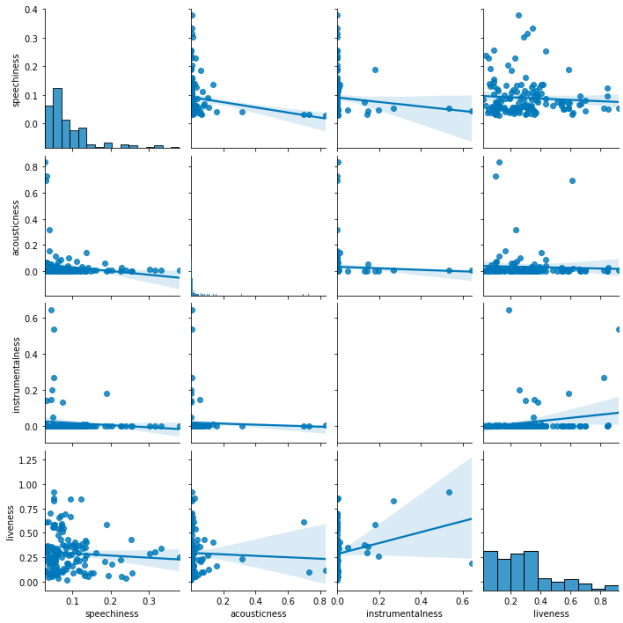


I then created scatterplot matrices to explore the distributions and relationships of certain variables which I thought might be correlated. The code and outputs for these scatterplot matrices can be found below:

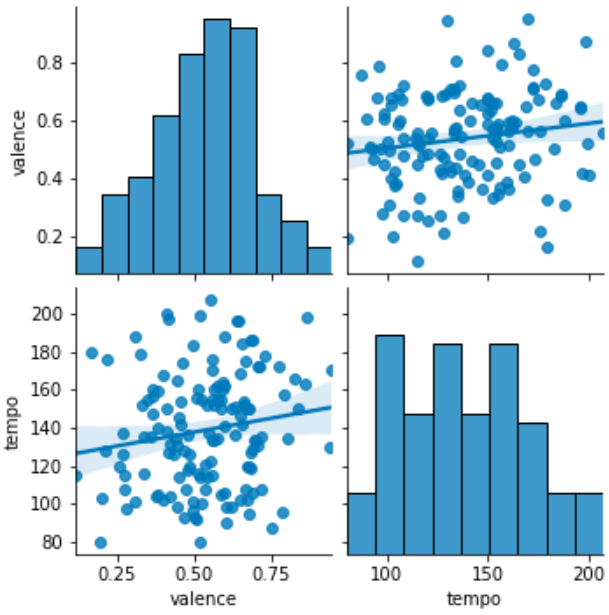
```
numerical_subset1 =
fob_songs[['danceability', 'ener-
gy', 'loudness']]
sns.pairplot(numerical_subset1,
kind = 'reg')
```




```
numerical_subset2 =
fob_songs[['speechiness', 'acous-
ticness', 'instrumentalness',
'liveness']]
sns.pairplot(numerical_subset2,
kind = 'reg')
```

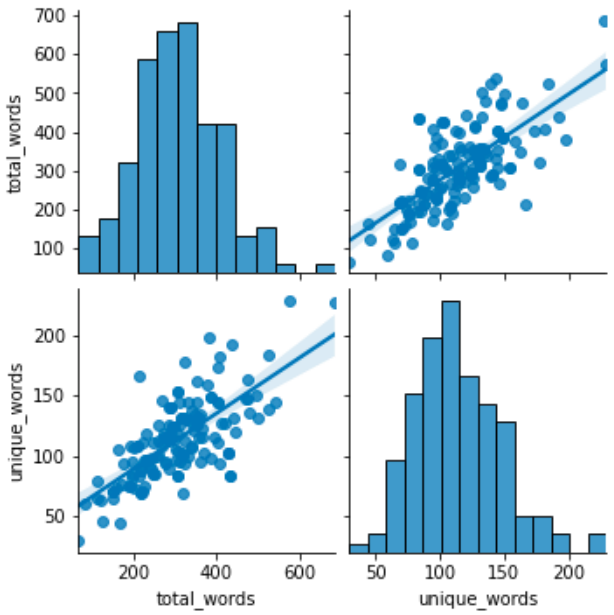



```
numerical_subset3 =
fob_songs[['valence', 'tempo']]
sns.pairplot(numerical_subset3,
kind = 'reg')
```



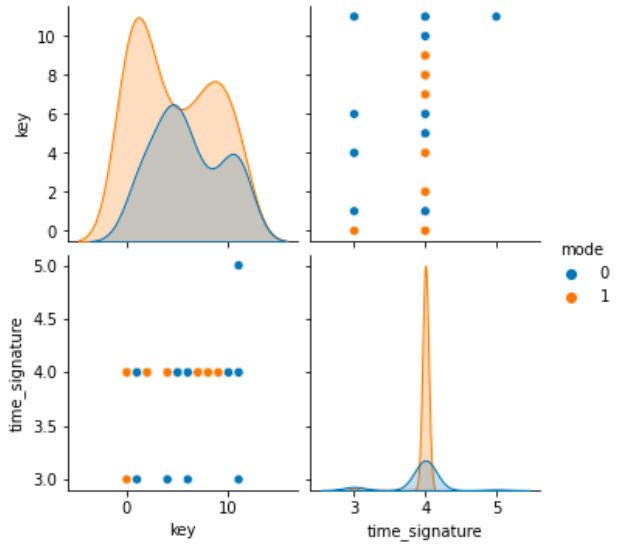

```
numerical_subset4 =
fob_songs[['total_words',
'unique_words']]
```

```
a = sns.pairplot(numerical_sub-
set4, kind = 'reg')
```



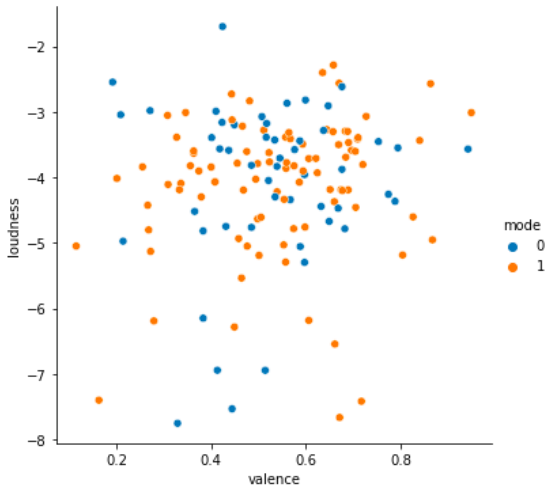
For the next plot, I wanted to visualize musical aspects of the data. Here, I explore the categorical distributions of song keys and time signatures. The data is separated by mode, where songs written in major (1) keys are shaded in orange and songs written in minor (0) keys are shaded in blue:

```
# Show major and minor in differ-
ent colors across musical varia-
bles
categorical_subset =
fob_songs[['key','mode',
'time_signature']]
sns.pairplot(categorical_subset1,
hue = 'mode')
```



I also wanted to visualize the valence and loudness across the major and minor keys. High valence values on a scale from 0 to 1 indicate a happier song, and vice versa, so I assumed that most of those higher-value songs would be written in major keys. I colored the data by key on a scatterplot of loudness and valence to investigate, and found that my assumption was largely incorrect:

```
# Check major and minor in rela-
tion to valence and loudness,
which describe a song's mood
sns.relplot(x="valence", y="loud-
ness", hue="mode",
data=fob_songs);
```



Note that valence is measured on a scale between 0 and 1, and loudness is measured in dB that typically range from -60 to 0. This scatterplot shows that high-valence and high-loudness values are distributed across both major and minor modes, and that the major mode does not necessarily indicate high valence or loudness.

For the final group of visualizations, I wanted to understand the distributions of certain features across the post-hiatus and pre-hiatus classes. This required some pre-processing because I wanted to compare percentages across the two class distributions.

```
# Count songs in each class
post_count = DataFrame(mode_df.groupby(by=['class']).size())[0][0]
pre_count = DataFrame(mode_df.groupby(by=['class']).size())[0][1]
print(f'There are {pre_count} pre-hiatus songs and {post_count} post-hiatus songs')
```

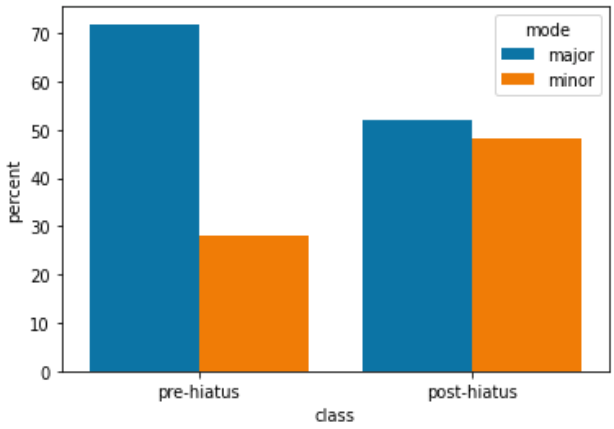
```
There are 89 pre-hiatus songs and 52 post-hiatus songs
```

```
# Balance of dataset
print(f'Pre-hiatus: {pre_count/len(fob_songs)*100:.2f}% of dataset')
print(f'Post-hiatus: {post_count/len(fob_songs)*100:.2f}% of dataset')
```

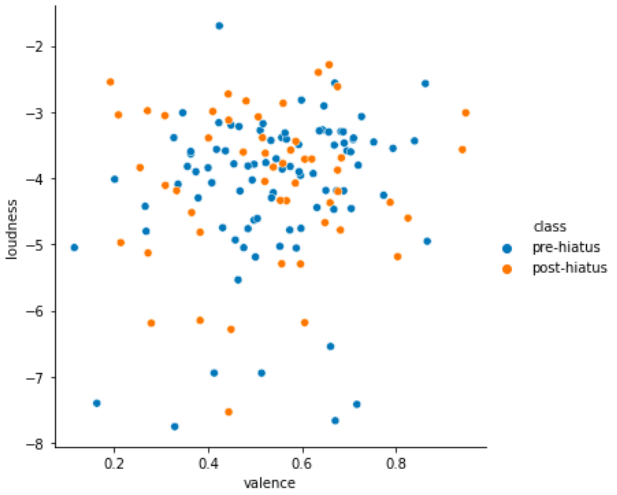
```
Pre-hiatus: 63.12% of dataset
Post-hiatus: 36.88% of dataset
```

```
mode_df = fob_songs[['class', 'mode']].copy()
mode_df['mode'] = mode_df['mode'].replace(1, 'major').replace(0, 'minor')
grouped_modes = DataFrame(mode_df.groupby(by=['class', 'mode']).size())
post_major, post_minor, pre_major, pre_minor = grouped_modes[0]
mode_dist = DataFrame({'class': ['pre-hiatus', 'pre-hiatus', 'post-hiatus', 'post-hiatus'],
                       'mode': ['major', 'minor', 'major', 'minor'],
                       'count': [pre_major, pre_minor, post_major, post_minor],
                       'total': [pre_count, pre_count, post_count, post_count]})
mode_dist['percent'] = (mode_dist['count'] / mode_dist['total'])*100
mode_dist
```

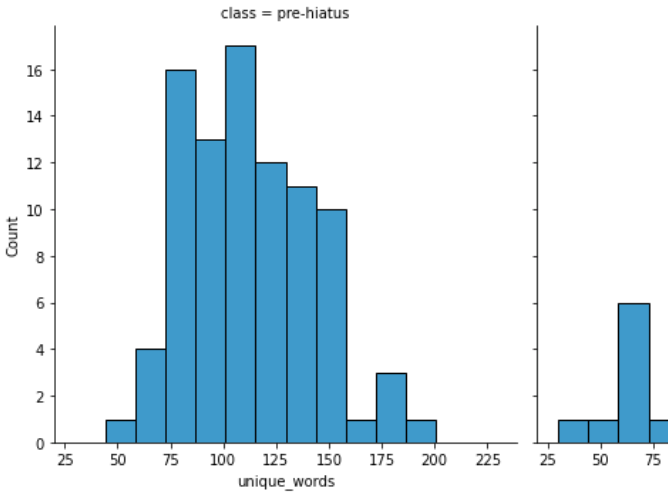
```
sns.barplot(x='class', y='percent', hue='mode', data=mode_dist)
```



```
sns.relplot(x="valence", y="loud-
ness", hue="class",
data=fob_songs)
```



I also wanted to test my earlier hypothesis, and determine whether pre-hiatus songs actually contain more unique words than post-hiatus songs. First, I plotted the distributions of unique words for each class.



Although "Rat A Tat" (post-hiatus track) boasts the highest unique word count of 229, the average pre-hiatus song appears to contain more words than the average post-hiatus song.

However, this histogram doesn't provide enough information to arrive at a meaningful conclusion. I decided to run a statistical test to determine the answer once and for all.

```
# Pull necessary columns for z-
test
pre_sample =
fob_songs[fob_songs['class'] ==
'pre-hiatus']['unique_words']
post_sample =
fob_songs[fob_songs['class'] ==
'post-hiatus']['unique_words']
```

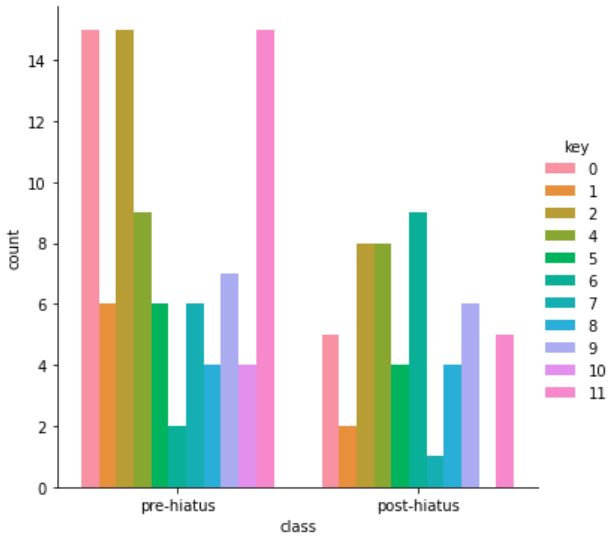
Once I established that the population variances were known, I implemented the correct method: a two-sample Z-test.

```
# Use z-test since the population
variances are known
ztest(pre_sample, post_sample,
value=0)
```

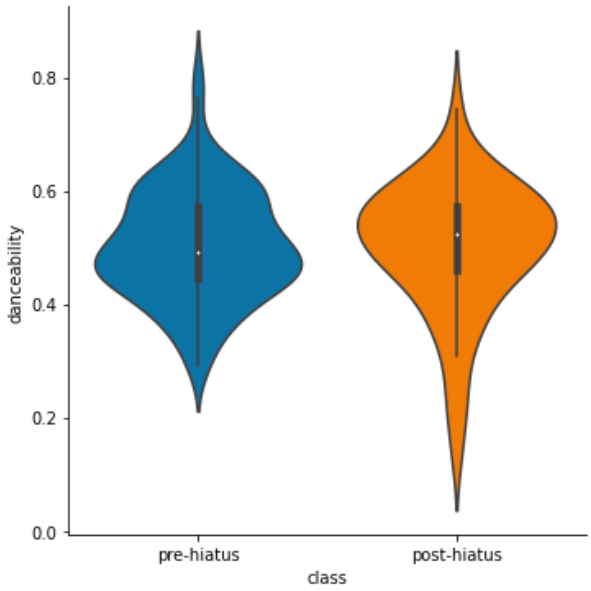
This test yields a high p-value of ~0.46, indicating no evidence of a difference in the mean unique lyrics between pre-hiatus songs and post-hiatus songs. Although this analysis ultimately disproved my hypothesis, it was still interesting to investigate the topic.

One plot I was quite excited about was the distribution of song keys, where 0 indicates the key of C, 1 indicates C#/Db, 2 indicates D, and so on. I think this plot was my favorite:

```
# Keys of all songs in both pre-
hiatus and post-hiatus periods
sns.catplot(x='class', hue =
'key', kind = 'count', data =
fob_songs)
```



To conclude the visualization section, I looped through the numerical features and created violin plots divided on the class variable. The use of the violin plot instead of a conventional box plot not only adds the kernel density estimation to each distribution graph, but also aligns with the musical theme of this analysis. Below is the violin plot for danceability:



V Summary Statistics and Correlational Analysis

First, I ran `fob_songs.isnull().sum()` to confirm that my dataset contained no null values.

Then, I built a loop to output a quick breakdown of the categorical variables:

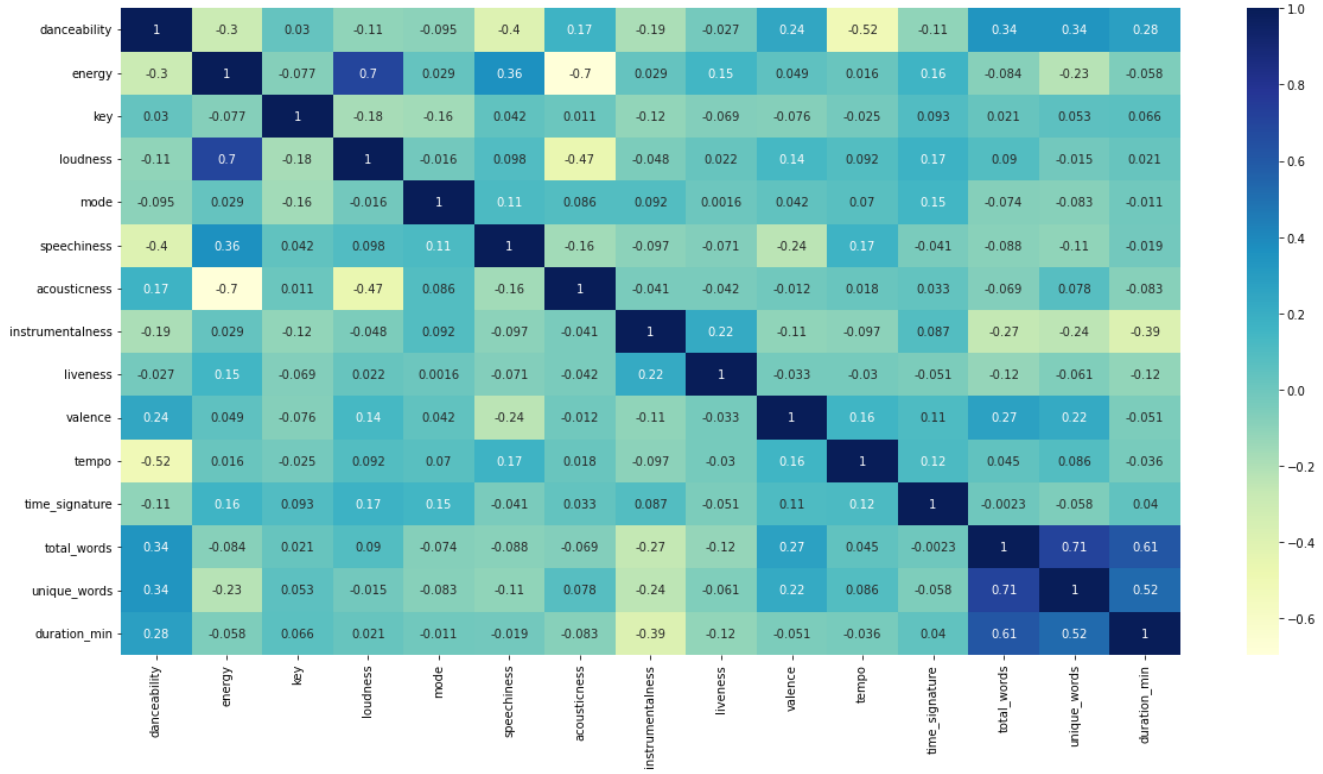
```
# Break down categorical data
for col in categorical:
    print(col)

print(fob_songs[col].value_counts(
), '\n')
```

```
key
2      23
0      20
11     20
4      17
9      13
6      11
5      10
1       8
8       8
7       7
10      4
Name: key, dtype: int64

mode
1    91
0    50
Name: mode, dtype: int64

time_signature
4    134
3      6
5      1
Name: time_signature, dtype:
int64
```

Next, I created a heatmap of the linear (Pearson) correlations in my dataset using the **.corr()** method:

```
plt.figure(figsize=(20,10))
sns.heatmap(fob_songs.corr(),
cmap="YlGnBu", annot=True)
plt.show()
```

Output:



I then output any correlations with an absolute value at or above 0.5 to analyze the more significant numbers:

```
correlations = fob_songs.corr()
correlations[abs(correlations) >=
0.5]
```

I documented those higher correlations here:

- The number of total words was positively correlated with the number of unique words at ~71.3%.
- The number of total words was positively correlated with song duration at ~61.4%.
- The number of unique words was positively correlated with song duration at ~52.3%.
- Loudness was positively correlated with energy at ~69.6%.
- Acousticness was negatively correlated with energy at ~-69.6%.

Strangely, tempo was negatively correlated with danceability at ~-51.6%. The other correlations made perfect sense, but this last one threw me off.

To finish the analysis, I ran `fob_songs.describe()` to pull all of the relevant descriptive statistics such as count, mean, standard deviation, and quartiles.

I then explored the few entries with non-standard time signatures:

```
# Which songs have non-standard time signatures (not in 4/4 time)?
fob_songs[fob_songs['time_signature'] != 4]
```

According to Spotipy, there are six FOB songs written in 3/4 time, and one song written in 5/4 time.

I exported the **fob_songs** dataframe at various stages HERE. This concludes the exploratory phase of the project.

VI Logistic Regression from Scratch

Script: models/LR.py

The goal of this customized class was to successfully implement a Logistic Regression model such that we can observe its progress. The class is also well-documented and fully type-hinted.

First, I imported the necessary libraries for mathematical operations and plotting:

```
# Setup
import math
import numpy as np
import pandas as pd
from pandas import DataFrame
from matplotlib import pyplot as plt
```

Then, I initialized the class along with some important properties.

```
class ScratchLogisticRegression:

    def __init__(self, learning_rate: float = 0.001, n_iters: int = 1000):
        """Model hyperparameters"""
        self.learning_rate = float(learning_rate)
        self.n_iters = n_iters

        """Properties related to cost function"""
        self.weights = None
        self.bias = None
        self.loss_history = []

        """Checks to see whether model has already been fit or    used to predict classes"""
        self.is_fit = None
        self.is_predicted = None
```

All of the methods shown below are defined within the ScratchLogisticRegression class with descriptive docstrings.

Next, I defined the **repr** and **str** methods:

```
def __repr__(self) -> str:
    return f'<ScratchLogisticRegression(learning_rate={self.learning_rate}, n_iters={self.n_iters})>'

def __str__(self) -> str:
    return \
    f"""ScratchLogisticRegression object
    self.learning_rate = {self.learning_rate}
    self.n_iters = {self.n_iters}
    self.weights = {self.weights}
    self.bias =  {self.bias}"""
```

Although both methods return a string representation of the object, they are designed for slightly different purposes. The **repr** method provides developers the exact syntax they need to recreate the object in question. Meanwhile, the **str** method simply provides information to the end user about the given object.

Next, I defined a pseudo-private stable sigmoid function. This function is a modified, stabilized version of sigmoid that's robust to Zero-Division Errors.

The Python implementation looks like this:

```
def _sigmoid(self, x: pd.DataFrame) -> np.ndarray:
      """Define stable sigmoid function which will be used in the gradient descent loop"""
            x = x.astype(float) # Convert types
            stable_sig = np.where(x < 0, \
                  # If x < 0
                  np.exp(x)/(1 + np.exp(x)), \
                  # If x >= 0
                  1/(1 + np.exp(-x)))

            return stable_sig
```

I then added a method that allows users to modify the hyperparameters without creating a new object:

```
def update_params(self, new_learning_rate: float = None, new_n_iters: int = None) -> str:
      """Account for missing arguments"""
      if new_learning_rate is None:
            new_learning_rate = self.learning_rate
      if new_n_iters is None:
            new_n_iters = self.n_iters

      self.learning_rate = new_learning_rate
      self.n_iters = new_n_iters
      return f'learning rate: {self.learning_rate}, number of iterations: {self.n_iters}'
```

Next was the most complicated method, **fit**. First, I modified the **is_fit** property to show that the model has been fit. Then, I stored the model feature information and initialized the model weights and bias.

```
def fit(self,  X_train: pd.DataFrame, y_train: pd.DataFrame,
      X_test: pd.DataFrame, y_test: pd.DataFrame) -> str:
      self.is_fit = True
      np.random.seed(0)

      """Store features of input data"""
      self.feature_names_in_ = list(X.columns)
      self.n_features_in_ = len(self.feature_names_in_)

      """Initialize weights and bias to random and zero, respectively"""
      n_rows, m_features = X.shape
      self.weights = np.random.random(m_features)
      self.bias = 0
```

I set up some lists to track error throughout the fitting process. I also created the **Xs, ys,** and **norms** lists to use in the **zip** function later.

```
"""Initialize lists to track both training and test error at each epoch"""
train_loss, test_loss, dw_history, db_history = [], [], [], []
Xs, ys = [X_train, X_test],
```

```
[y_train, y_test]
norms = [(1 / len(X_train)), (1 /
len(X_test))]
```

I then designed a loop structure that would allow me to calculate training and test error at each iteration:

```
for is_training, (X, y, norm) in
enumerate(zip(Xs, ys, norms)):
```

To unpack this briefly, the first nested loop within each iteration will correspond to the training error, where **is_training** is 0, **X** is the **X_train** dataframe, **y** is the **y_train** dataframe, and the **norm** is the inverse of the length of **X_train.** The second nested loop will correspond to the test error, where **is_training** is 1, **X** is the **X_test** dataframe, **y** is the **y_test** dataframe, and the **norm** is the inverse of the length of **X_test.**

Next, it was time to build the actual logistic regression model. Inside the above loop structure, I iterated through the specified number of epochs. At each iteration, I built a linear model by taking the dot product of the input data and weights (initialized to random numbers), then added the bias. These model scores were transformed by the sigmoid to arrive at the predicted y values.

```
for _ in range(self.n_iters):

"""Build fit method's version of
linear model as a dot product of
the X-vector and the model weights;
add the bias"""
     linear_model = np.dot(X,
self.weights) + self.bias
     y_predicted = self._sig-
moid(linear_model)
     y_predicted, y = y_pre-
dicted.astype(float),
y.astype(float) # Force float type
```

Continuing inside the loop, I computed the logarithmic loss to evaluate the model at each iteration. I separated the class 1 and class 0 components for readability, and tracked the final result in the **train_loss** and **test_loss** lists. Note that an epsilon term is added inside the class 1 component to prevent errors.

```
"Compute log_loss (binary cross-
entropy loss) function"
class1_error = y * np.log(y_pre-
dicted + np.finfo(float).eps) #
Add epsilon term to prevent errors
class0_error = (1-y) * np.log(1 -
y_predicted + np.finfo(float).eps)
log_loss = -norm *
np.sum(class1_error + class0_er-
ror)
```

Now that the loss was calculated, it was time to use our tracking system along with the new **match** statements available in Python 3.10. The **is_training** variable from the outer loop told us whether we were training or testing, and where to append the most recent loss calculation:

```
"""Case 0 corresponds to training
error tracking; case 1 corresponds
to test error"""
is_training = int(is_training)
match is_training):
     case 0:
          train_loss.ap-
pend(log_loss)
     case 1:
          test_loss.ap-
pend(log_loss)
```

Note: I tried evaluating **is_training** as a Boolean, but for some reason it decreased the model performance, so I just used 1 and 0 instead of True and False.

The final component of the loop was to optimize the model at each training iteration (where **is_training** is 0) using gradient descent. The derivatives of logarithmic loss were calculated with respect to the weights (**dw**) and bias (**db**), and stored in **dw_history** and **db_history**, respectively. After

multiplication by the learning rate parameter, the gradients were used to update the model weights and bias via the decrement operator.

```
"""Only update params during training"""
if is_training == 0:

      """Gradient Descent: Calculate partial derivatives of the cost function with respect to the weights and biases"""
      residuals = y_predicted - y
      dw = norm * np.dot(X.T, residuals)
      db = norm * np.sum(residuals)

      dw_history.append(dw)
      db_history.append(db)

      """Update weights and biases based on the learning rate and derivatives"""
      self.weights -= self.learning_rate * dw
      self.bias -= self.learning_rate * db
```

The control flow then exited the loop, the history lists were stored as class properties, and a feedback message was returned to assure the user that everything went smoothly.

```
self.loss_history, self.dw_history, self.db_history =
loss_history, dw_history, db_history
return 'Successfully fit Logistic Regression model to input data.'
```

Next up was the **predict_proba** method. This method first checks the **is_fit** property, and fits the model if there is no fit yet. Then, it builds a linear model using the object's fitted weights and bias, transforms the model scores using the sigmoid function, and outputs a vector of probabilities.

```
def predict_proba(self, X: pd.DataFrame) -> np.ndarray:
      """Build predict method's version of linear model as a dot product of the X-vector and updated model weights; add the updated bias
            Run sigmoid function to convert predictions to probabilities of Class 1 Membership"""
      if self.is_fit is None:
            self.fit(X)

      pred_linear_model = np.dot(X, self.weights) + self.bias
      return self._sigmoid(pred_linear_model)
```

The **predict** method first modifies the **is_predicted** property to show that classes have been predicted using the model. Then, it calls the **predict_proba** method above and uses a list comprehension to map the probabilities to output classes.

```
def predict(self, X: pd.DataFrame) -> list:
      """Output predicted classes for each data point"""
      self.is_predicted = True

      probabilities = self.predict_proba(X)
      self.predicted_classes = [1 if prob > 0.5 else 0 for prob in probabilities]
      return self.predicted_classes
```

Next was an extremely simple baseline model; all it does is return a random array of zeros

and ones to represent the class assignments. This array is also stored as a property.

```
def baseline(self, X: pd.DataFrame)
-> np.ndarray:
"""Build naïve baseline model that
outputs random ones and zeros as
class predictions"""
     rows = X.shape[0]
     self.baseline_predictions =
np.random.randint(low=0, high=2,
size=rows)
     return self.baseline_predic-
tions
```

The remaining methods focused on model evaluation and visualization. First in this group was the **metrics** method, which calculates accuracy, precision, recall, and the F1 score. See the formulas below for more information.

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

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

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

$$F1\text{-}score = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Common Classification Metrics (MACHINE LEARNING TUTORIAL)

After checking that the model predictions were stored, the first step was to count the amounts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) as predicted by the model.

To accomplish this, we used list comprehensions to compare the predicted class label (**p_label**) to the actual label (**a_label**). We summed all the cases of TP, FP, TN, and FN whenever their scenarios occurred:

```
def metrics(self, X: pd.DataFrame
| np.ndarray | list, actuals:
pd.DataFrame, predictions: pd.Da-
taFrame | np.ndarray | list =
None) -> pd.DataFrame:
if predictions is None:
          if self.is_predicted is
None:
               self.predict(X)
          predictions = self.pre-
dicted_classes

     """Count true positives, true
negatives, false positives, and
false negatives"""
     self.true_pos = sum([1 for
p,a in zip(predictions, actuals)
if int(p)==int(a)==1])
     self.false_pos = sum([1 for
p,a in zip(predictions, actuals)
if int(p)==1 and int(a)==0])
     self.true_neg = sum([1 for
p,a in zip(predictions, actuals)
if int(p)==int(a)==0])
     self.false_neg = sum([1 for
p,a in zip(predictions, actuals)
if int(p)==0 and int(a)==1])
```

Then, it was time to calculate the accuracy, precision, recall, and F1 Scores according to the formulas above. **Try-except** blocks were employed here to avoid Zero Division Errors.

```
"""Calculate accuracy, precision,
recall, and F1-Score; account for
ZeroDivisionErrors"""
self.accuracy = (predictions ==
actuals).sum() / len(actuals)

try:
     self.precision =
```

```
self.true_pos / (self.true_pos +
self.false_pos)
except ZeroDivisionError:
      self.precision = 0.0

try:
      self.recall = self.true_pos /
(self.true_pos + self.false_neg)
except ZeroDivisionError:
      self.recall = 0.0

try:
      self.f1 = (self.precision *
self.recall) / (self.precision +
self.recall)
except ZeroDivisionError:
      self.f1 = 0.0
```

Finally, the metrics were formatted as strings and stored in a list called **percentages,** then converted into a dataframe for convenient display.

```
metrics = [self.accuracy,
self.precision, self.recall,
self.f1]
percents = [str(round(metric*100,
2))+'%' for metric in metrics]
metrics_df = DataFrame({'Metric':
['Accuracy', 'Precision', 'Re-
call', 'F1 Score'], 'Value': per-
cents})

return metrics_df
```

Next up was the **confusion_matrix** method**,** which would display all the counts of TP, FP, TN, and FN in another DataFrame.

```
def confusion_matrix(self, actu-
als: pd.DataFrame) -> pd.Data-
Frame:
"""Display counts of the model's
true positives, true negatives,
false positives, and false nega-
tives"""
      self.metrics(actuals)
      matrix = DataFrame({'Index':
['Predicted', 'True', 'False'],
            '': ['True',
self.true_pos, self.false_neg],
            'Actual ': ['False',
self.false_pos, self.true_neg]})
            self.matrix = ma-
trix.set_index('Index')
            return self.matrix
```

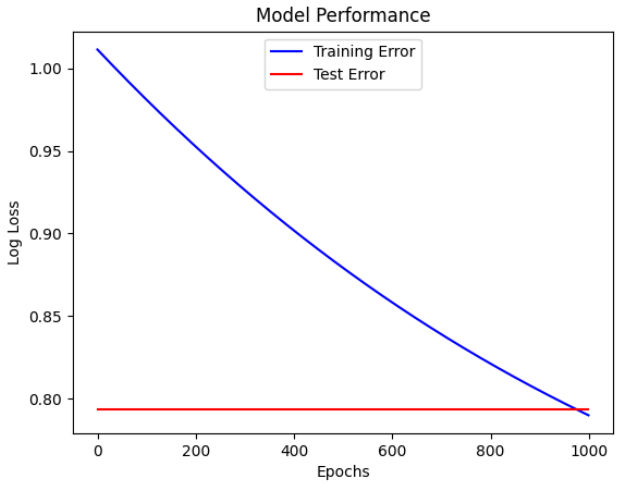
Finally, I built the **plot_loss** method, which allowed us to monitor and improve the performance of our cost function. This method takes a **graph** argument, which tells it to graph either training error only, test error only, or both. The function uses MatPlotLib to graph the number of epochs on the x-axis, and the log loss on the y-axis.

```
def plot_loss(self, graph: str =
'both') -> str:
      """Note final loss values and
max loss"""
      final_train =
self.train_loss[-1]
      final_test = self.test_loss[-
1]

      """Figure setup with sub-
plots"""
      fig, ax = plt.subplots()
      plt.title('Model Perfor-
mance')
      plt.xlabel('Epochs')
      plt.ylabel('Log Loss')
```

Three scenarios for each possible argument to the **graph** parameter are specified in the match statement below. Then, the graph is returned along with the final error metrics:

```
"""Choose whether to graph only
training error, only test error,
or both (all options show leg-
end)"""
match graph:
     case 'train':
          max_loss =
math.floor(max(self.train_loss) +
0.2

     ax.plot(self.train_loss,
color = 'blue',
                label = 'Training
Error')
          ax.legend(loc='upper
center')

     case 'test':
          max_loss =
math.floor(max(self.test_loss) +
0.2
          ax.plot(self.test_loss,
color = 'red',
                label = 'Test Er-
ror')
          ax.legend(loc='upper
center')

     case 'both':
          max_loss =
math.floor(max(max(self.train_loss
),
     max(self.test_loss))) + 0.2

     ax.plot(self.train_loss,
color = 'blue',
                label = 'Training
Error')
          ax.plot(self.test_loss,
color = 'red',
                label = 'Test Er-
ror')
          ax.legend(loc='upper
center')

          plt.show()
          return f'Final Training
Error: {final_train:.2f},
                Final Test Error:
{final_test:.2f}'
```

A plot with both training and test error looks like this:



That's it for our Scratch Logistic Regression class.

VII FEATURE ENGINEERING

NOTEBOOK: NOTEBOOKS/FEATURE_ENGINEERING.PY

Now that the infrastructure for our LR model was built, it was time for the most artistic aspect of data science: feature engineering.

I started with the setup:

```
# Install module for specific py-
thon version
! python3.10 -m pip install mlx-
tend -q

import pandas as pd
```

```
from pandas import DataFrame
import numpy as np
import matplotlib
from matplotlib import pyplot as
plt
%matplotlib inline
import seaborn as sns
from sklearn.preprocessing import
MinMaxScaler
from sklearn.preprocessing import
OneHotEncoder
from sklearn.model_selection im-
port train_test_split
from sklearn.linear_model import
LogisticRegression
from mlxtend.feature_selection im-
port SequentialFeatureSelector

# Display entire dataframe
pd.set_option('display.max_rows',
None)
pd.set_option('display.max_col-
umns', None)
pd.set_option('display.width',
None)
```

I then loaded and rearranged the data:

```
fob_songs = pd.read_csv('data/pro-
cessed/FOB_songs_processed.csv',
index_col=0)

# Rearrange columns with numbers
on the left and categoricals on
the right
fob_songs = fob_songs[['title',
'danceability', 'energy', 'loud-
ness',
       'speechiness', 'acoustic-
ness', 'instrumentalness',
'liveness',
       'valence', 'tempo', 'to-
tal_words', 'unique_words',
       'duration_min', 'key',
'mode', 'time_signature',
'class']]
fob_songs.head()
```

I also confirmed that no data was lost in transit by running `fob_songs.isnull().sum().sum()`, which returned 0.

Now, it was time to explore the feature distributions. I started by comparing the means and medians from each numerical column across the pre and post-hiatus classes to determine which features showed the most skewness:

```
pre = fob_songs[fob_songs['class']
== 'pre-hiatus']
post =
fob_songs[fob_songs['class'] ==
'post-hiatus']
numeric= fob_songs.columns[1:-
4].values
categorical = fob_songs.columns[-
4:].values
pre_means, pre_medians,
post_means, post_medians = [], [],
[], []

for col in numeric:
    pre_means.ap-
pend(pre[col].mean())
    post_means.ap-
pend(post[col].mean())
    pre_medians.ap-
pend(pre[col].median())
    post_medians.ap-
pend(post[col].median())
```

I compiled all this information into a data-frame.

```
# Set up dataframe to compare each mean and median with their deltas
num_features = DataFrame({'feature': numeric, 'pre-hiatus_mean': pre_means,
                          'post-hiatus_mean': post_means, 'pre-hiatus_median': pre_medians,
                          'post-hiatus_median': post_medians})
num_features['mean_delta'] = num_features['pre-hiatus_mean'] - num_features['post-hiatus_mean']
num_features['median_delta'] = num_features['pre-hiatus_median'] - num_features['post-hiatus_median']
```

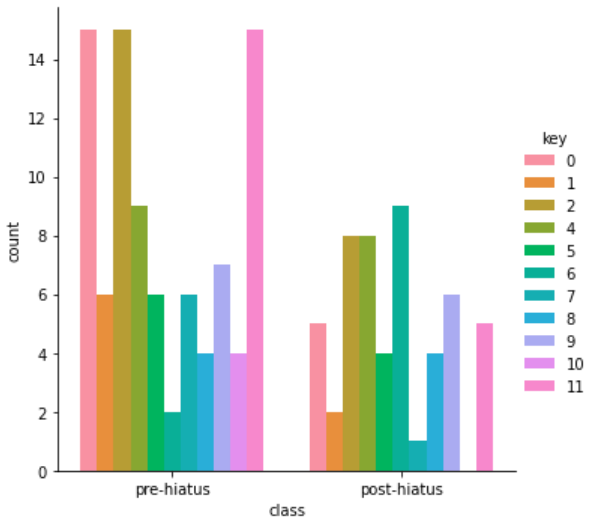
The first three rows of output looked like this:

feature	pre-hiatus_mean	post-hiatus_mean	pre-hiatus_median	post-hiatus_median	mean_delta	median_delta
danceability	0.50391	0.501096	0.493	0.5225	0.002814	-0.0295
energy	0.895124	0.884673	0.939	0.9065	0.010451	0.0325
loudness	-4.129865	-4.043577	-3.843	-3.8365	-0.086288	-0.0065

I also created some side-by-side visualizations of the categorical features using Seaborn:

```
for col in fob_songs[categorical[:-1]]:
    sns.catplot(x='class', hue = col,
    kind = 'count', data = fob_songs)
```

The plot for the feature **key** looked like this:



From this categorical exploration, I concluded that **mode** should be a feature because a minor song is a good predictor of the pre-hiatus class. I was unsure which **key**s (if any) should be included as features. I also noted that **time_signature** should be excluded from the final model, as most songs across both classes were in 4/4 time.

After the visualization stage, I explored feature scaling. The majority of the numerical features were already scaled between 0 and 1, so I simply used MinMax Scaler to place other variables on this scale as well:

```
unscaled_cols = ['tempo', 'total_words', 'unique_words', 'duration_min']
unscaled = features[unscaled_cols]
# MinMaxScaler yields features scaled from 0-1
scaled = MinMaxScaler().fit_transform(unscaled)
scaled = DataFrame(scaled, columns=unscaled_cols)
```

Once I prepared the numerical data, it was time to separate the feature components and prepare the class and categorical data for one-hot-encoding:

```
one = OneHotEncoder(sparse=False)
numbers = features[numeric].iloc[:,:-4].copy()
categories_df = features[['key', 'time_signature']].copy()
cols = categories_df.columns
# Rough by-hand label-encoding of class column
# Pre-hiatus maps to 0; Post-hiatus maps to 1
features = features.replace('pre-hiatus', 0).replace('post-hiatus', 1)
features['class'].head()
```

Then, it was time to one-hot-encode the categorical data. Because **mode** was already a binary variable (with 1 representing major and 0 representing minor), only **key** and **time_signature** needed to be encoded:

```
cats_encoded = one.fit_transform(categories_df)
feature_names = one.get_feature_names_out(cols)
# df of one-hot-encoded variables
cats_df = DataFrame(cats_encoded, columns=feature_names)
cats_df.head()
```

I also converted the **mode** column to the float type for uniformity. This dataset was exported HERE.

Now that all the data was prepared, it was time to select the features to be used in the final model. To determine this, I implemented a backward selection using the Sequential Feature Selector from the mlextend library that was scored using F1:

```
y = features.pop('class')
backward = SequentialFeatureSelector(LogisticRegression(max_iter=5000),
          k_features=(2, 8), forward=False, floating=True,
          verbose=2, scoring='f1', cv=3)
backward.fit(features, y)
```

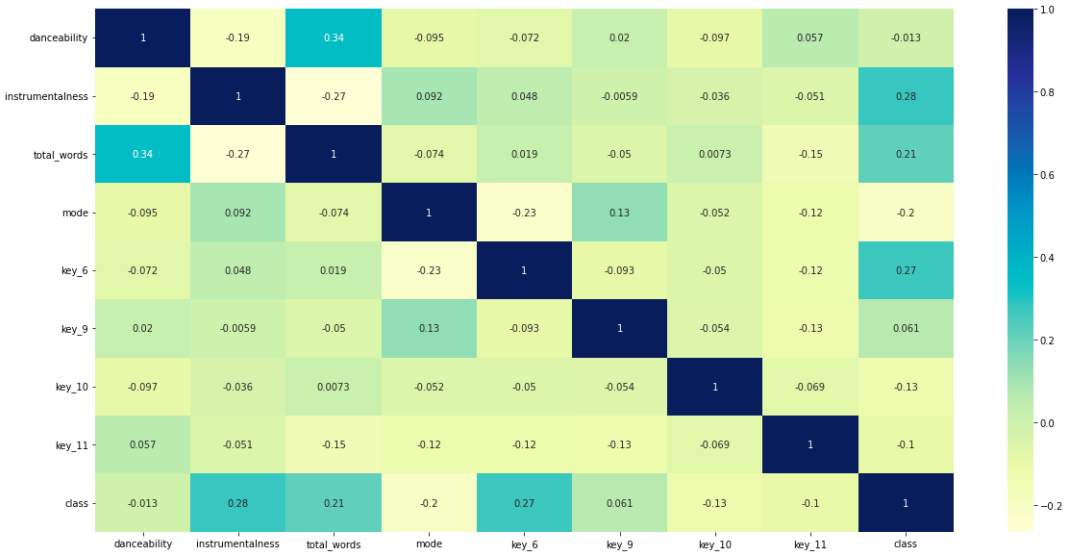
This backward selection yielded the following features: **danceability, instrumentalness, total_words, mode, key_6, key_9, key_10,** and **key_11**. Here, the binary **key** variables are 1 if a song uses that specific key and 0 otherwise.

The next step was to export the final features for modeling:

```
selected_features = features[final_features]
```

```
selected_features = pd.concat([se-
lected_features, y], axis=1)
```

I also built another heatmap to confirm that there were no significant correlations in this feature subset:



This feature set was exported HERE. I also experimented with manual feature selection; those features can be found HERE.

VIII Non-Technical Summary

I wanted to add a section that makes the contents of this paper accessible to everyone, regardless of their background.

First, let's explain how this project is suited to AI and machine learning. When the data we have is labeled (for example, as pre-hiatus or post-hiatus), we apply statistical methods that are called supervised techniques. These usually fall into one of two categories: regression or classification. Regression involves predicting a number, and classification involves predicting a category which tells us which type of data point we're looking at.

Our case, predicting whether or not a Fall Out Boy song belongs to the pre-hiatus or post-hiatus period, is a supervised classification task. It also happens to be the simplest kind: binary classification, where there are only two categories to choose from, and the classes are labeled as Class 0 and Class 1. These classes can be True/False, Good/Bad, Pass/Fail, etc. as long as there are only two categories.

Below, I have provided an example of linear regression:

Linear Regression (SOURCE)

In statistics, this method is known as Linear Regression, because you're predicting numbers (hence the "regression") using a line (hence the "linear"). Here's the trick: This same Linear Regression technique is used in our classification method.

You build the same linear model, and then convert its scores into probabilities. These probabilities represent the likelihood that our data point belongs to Class 1– or, in this case, the likelihood that our song belongs to the post-hiatus FOB era. Because probabilities always range from 0 to 1, the rule is as follows: If the probability is between 0 and 0.5, we assign the Class 0 label, and if it's between 0.5 and 1, we assign Class 1.

This method is called Logistic Regression (LR). LR is popular because it has a lot of practical applications and it's relatively easy to explain, as we just did. It's important for non-programmers to know that data scientists aren't coding this algorithm from scratch every time they need it. Especially under time constraints, that would be incredibly impractical.

Instead, they use a sort of cake-mix version of the algorithm. All they have to do is add their data and specifications to a tried-and-true recipe, and they achieve a perfect LR model with only a few steps; the ready-made mix does all the heavy lifting.

By creating the model from scratch, I was able to fine-tune many parameters to match the needs of my project. Once my algorithm was finished, I modeled my data and compared those results to some other cake-mix algorithms for binary classification. See the background on logistic regression and the comparative modeling section for more information.

The rest of this section will discuss the project results. For binary classification tasks, we measure the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). We then expand these counts into specific metrics which are used to evaluate model performance. The higher these metrics, the better the model:

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

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

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

$$F1\text{-}score = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Common Classification Metrics (SOURCE)

The results from my best-performing models are summarized here:

Model	Accuracy	F1 Score	Precision	Recall
Random Forest	79.07%	70.97%	64.71%	78.57%
LR (Scratch)	41.86%	28.81%	40.48%	100.00%
LR (Cake Mix)	76.74%	64.29%	52.94%	81.82%

If someone were to ask the question, "Can an AI differentiate between Fall Out Boy's pre-hiatus and post-hiatus music?", the answer would be "well enough."

A Relevant Spotify Audio Features

Numerical Features:

- *Acousticness:* Confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence that the track is acoustic.
- *Danceability:* Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
- *Duration_min:* Duration of the track in fractions of minutes
- Example: A track that is 3.2 minutes will be 3 minutes and 12 seconds (12 seconds is 20% of a minute)
- *Energy:* Measure from 0.0 to 1.0 representing intensity and activity. Typically, energetic tracks feel fast, loud, and noisy, based on the perceptual features of dynamic range, perceived loudness, timbre, onset rate, and general entropy.
- *Instrumentalness:* Measure from 0.0 to 1.0 representing the likelihood that the track is instrumental and contains no vocals.
- *Liveness:* Detects the presence of an audience in the recording. Higher liveness values (on a scale of 0.0 to 1.0) represent an increased probability that the track was performed live.
- *Loudness:* Overall loudness in decibels (dB) averaged across the entire track, useful for comparing relative loudness. Values typically range between -60 and 0 db.
- *Tempo:* Overall estimated tempo of a track in beats per minute (BPM)
- *Valence:* Measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

Categorical Features:

- *Key:* Estimated overall key of the track. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C#/Db, 2 = D, and so on. If no key was detected, the value is -1.
- *Mode:* mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor by 0.
    - It's unclear from this designation how Spotify handles other musical modes such as Dorian, Phrygian, Lydian, etc.
- *Time_Signature:* Estimated overall time signature, meter, of a track
    - The Time_Signature feature assumes that all songs are measured in quarter notes, meaning the bottom number in the meter is always assumed to be 4.
    - The Time_Signature value represents the top number of the meter, which describes the number of quarter notes per measure.
    - 4 indicates a time signature of 4/4
    - 3 indicates a time signature of 3/4
    - 5 indicates a time signature of 5/4

Target Variable Feature:

- ***Class:*** If the song was released before or during 2009, it will belong to Class 0 (pre-hiatus FOB). If the song was released during or after 2009, it will belong to Class 1 (post-hiatus FOB)

B Playlist Information

Playlist Link

Playlist includes:

- All FOB Songs on Spotify
- FOB Features, such as "I've Been Waiting" and "One and Only"
- Both studio versions of "Calm Before the Storm"
- Bonus tracks
- Covers

Playlist does not include:

- Remixes, including *American Beauty, American Psycho* remix album
- Acoustic versions of songs whose main studio versions are already included

- Live versions of songs whose studio versions are already included
- FOB songs which aren't on Spotify such as "Pavlove"
- Solo work during or after the hiatus

Classification of songs:

- Class 0 = pre-hiatus and Class 1 = post-hiatus
- Songs are classified by release date, not writing date
- For example, "Light Em Up" and "Lake Effect Kid" will be classified as post-hiatus even though they were written pre-hiatus

C Assumptions and Data Limitations

Assumptions **not** required by Logistic Regression

- Linear relationship between the dependent and independent variables
- Normally distributed residuals (error terms)
- Homoscedasticity

Assumptions of Logistic Regression

- Dependent variable (class: pre-hiatus or post-hiatus) is binary
- Observations are independent
- Little or no multicollinearity among the independent variables
- Independent variables are linearly related to the log odds
- Sufficiently large sample size

Addressing the Assumption of Sufficiently Large Sample Size

- Generally, the definition of a sufficient sample size for logistic regression is subjective based on the required conditions of the use case such as statistical power
- A dataset of 141 rows is not ideal for this analysis, but at the time is was the maximum sample size of unique Fall Out Boy songs available to analyze
- In the future, this project could be improved by including the latest Fall Out Boy songs in order to increase the sample size

Limitations of Spotipy Data

- At the time of this project, there was no song data available for collection aside from the data included in the playlist
- The **mode** feature is a binary variable limited to the major and minor modes. In reality, there are many more modes available in music
- The **time_signature** feature is likely limited in a similar way to the **mode** feature

Note: The final data frame contained no missing values, and did not require any imputing methods

D Glossary

See VII for a non-technical summary of this project

- Backward feature selection – Process of finalizing model features. Start with all available features and then remove unhelpful features one at a time until only salient features remain
- Binary classification– Process of classifying data points into one of two categories
- Binary cross-entropy loss / Logarithmic Loss – Special loss function for binary classification problems. See the definition of loss below.
- Confusion matrix – Diagram counting true positives, false positives, true negatives, and false negatives for the output of a binary classification model
- Epoch – Iteration of a machine learning model. If a model runs three times, it runs for three epochs
- F1 score– A score that combines multiple model metrics (specifically precision and recall) to provide a balanced picture of a model's performance with just one number

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

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

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

$$F1\text{-}score = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Common Classification Metrics (SOURCE)

- Gaussian Naive Bayes (GNB) – Machine learning model that uses conditional (Bayesian) probability with the normal (Gaussian) distribution. GNB returns the probability that a given data point will be classified into each group, and the group with the highest probability wins
- Gradient descent– Model optimization method that uses calculus to minimize loss throughout the model training process. See the definition of model training below
- Heatmap – Color-coded diagram showing the linear (Pearson) correlation between features. This diagram is helpful because high correlations between features, which are undesirable in modeling, can be seen at a glance
- K-Nearest Neighbors (KNN)– Compares new data to existing data based on a similarity measure. In this case, KNN is used for binary classification by finding the group most similar to each new data point in the test set
- Loss– Function that represents error in a machine learning model. The goal of modeling is to obtain the lowest possible loss for each task
- Model training– Process of fine-tuning a model by updating its coefficients at each epoch and then evaluating model performance on the training data to make sure loss is improving
- Model testing– Process of generalizing the trained model by running it on new test data. If the metrics for model training and model testing are both strong, the model is reliable
- One-hot encoding – Assigning a numerical presence-or-absence variable to a feature with multiple categories so that a machine can process that feature

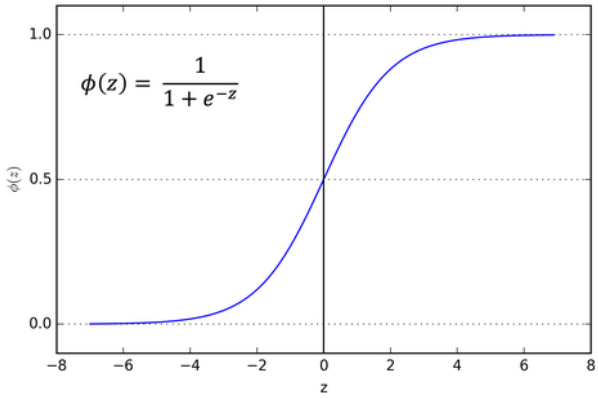
id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

(SOURCE)

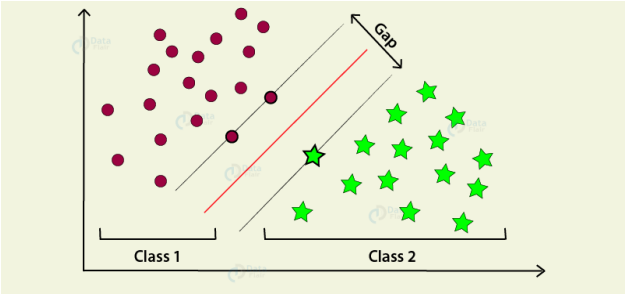
- Random Forest– Machine learning model that combines the output of multiple decision trees. For this classification task, each decision tree votes for the data point to be classified in a specific group; the group with the most votes wins
- Regular Expression (RegEx) – Sequence of characters that specifies a pattern to match in text. Used to search specific text patterns
- Sigmoid Function– Function that maps values between 0 to 1 into a graph shaped like the letter "S". In machine learning, this function is used to convert values to probabilities which fall between 0 and 1



(SOURCE)

- Supervised Machine Learning – Machine learning techniques that are used when the relevant data is labeled with a value or category

- Support Vector Machines– Machine learning model that creates a line or a hyperplane (for higher dimensions) which separates the data into classes



(Source)

- Two-sample z-test– Compares the means of two independent samples to determine if the difference between them is statistically significant