What Makes Trap Popular? Progress Report

Cristobal Sciutto - *csciutto* - 06099425

1 Model and Algorithm

The task being tackled is predicting the popularity of hip-hop songs based on their lyrics, using both critic reviews and Billboard charts. As described in the proposal, lyrical features of songs in the three major categories of semantics, structure, and vocabulary were extracted. The ideal end model that will be used is a recurrent neural-network (RNN), in order to consider all possible combinations of features ands to incorporate "temporal" features between successive lines of lyrics. Namely, following outputs and features were extracted for each song:

- Output
  - $y_b \propto$ best position $\times$ # weeks on charts.
    Due to the inverse proportionality of a peak position, I initially used $101 - peakPos$. This was then changed to the normalized version $\frac{(101-peakPos)^2}{10000}$, leading to drastically better results.
- Semantics
  - For each category in EmoLex, the proportion of words in each affect category.
- Structure
  - Length of song (# of words)
  - Number of stanzas
  - Average length of stanzas (# of words)
  - Number of lines of lyrics
  - Average length of line (# of words)
- Vocabulary
  - Indicators for the 10 most popular words in song

The infrastructure being used for this machine learning project is Keras with TensorFlow backend, running in a Python virtualenv.

For preliminary testing, the data extraction was restricted to only the Billboard charts. Taking into consideration critic scores would require additional web scraping, and limit training to the intersection of both datasets. Using the Billboard Charts API, I pulled songs from the R&B and Hip Hop charts from 2000 onwards, storing the song title, artist, peak position, and number of weeks on charts. Then, using the song's title and artist, I used the Genius API search to get the song's URL. With the URL, the Genius lyrics page was scraped. This data was stored in a Sqlite database, on which feature extraction was executed. Note: to simplify initial tests, features related to rhyming patterns were initially omitted.

Each individual feature dictionary created was transformed into an input vector using *scikit-learn*'s DictVectorizer, which creates a sparse matrix of all features for every song. From there, the *Keras* framework was used (following the tutorial in the documentation) to initially train Linear Regression (i.e. single layer neural-network). Despite an RNN being the end goal, more basic impediments have led to the postponing of its implementation. Nevertheless, due to the power of *Keras*, this will be rather trivial once the rest of the problems described below are solved.

1.1 *King Kunta, Kendrick Lamar* Exemplified

```
features = {
  # LYRIC SEMANTICS
  'anger':           0.1116751269035533,   # How predominant $anger$ words were
  'anticipation':    0.01015228426395939,
  'positive':        0.12690355329949238,
  'surprise':        0.005076142131979695,
  'trust':           0.015228426395939087,
  'sadness':         0.2233502538071066,
  'disgust':         0.1065989847715736,
  'fear':            0.1116751269035533,
  'joy':             0.015228426395939087,
  'negative':        0.27411167512690354,

  # GENERAL STRUCTURE
  'lines':           99,
  'avg_line':        7,
  'avg_stanzas':     63,
  'stanzas':         11,
  'word_count':      696,

  # SONG STRUCTURE
  '[Hook]':          0.36363636363636365,  # % of stanzas which were "hooks"
  '[Intro]':         0.09090909090909091,
  '[Outro]':         0.09090909090909091,
  '[Poem]':          0.09090909090909091,
  '[Produced]':      0.09090909090909091,  # Example of lyric parsing errors!
  '[Verse]':         0.2727272727272727,

  # VOCABULARY
  'pop_word a':      1,                    # Indicator for 'a' as popular word
  'pop_word funk':   1,
  'pop_word got':    1,
  'pop_word i':      1,
  'pop_word king':   1,
  'pop_word kunta':  1,
  'pop_word now':    1,
  'pop_word the':    1,
  'pop_word was':    1,
  'pop_word you':    1 }

output = (101 - peakPos) x weeks = (101 - 20) x 19 = 81 x 19 = 1539
output 2.0 = weeks x (101 - peakPos)**2 / 10000 = 19 x (101 - 20)^2 / 10000 = 19 x 0.6561 = 12.4659
```

2 Initial Results

Upon running linear regression for the first time, the size of the input vector was the same order of magnitude of the number of sizes (579 vs 1300). This led to terrible results on the neural-net, with essentially no changes to the loss. Consequently, some core changes were made.

- The peak position function was changed to a quadratic mapping from peak position to a score between 0 and 1. The final value of the output, y_b was also normalized.
- I decided to scrape more data, starting at 2000 instead of the originally planned 2009, doubling from

around 1300 samples to 2600.

- To reduce the dimensionality of the input vector, I opted for excluding the most popular words, as this was the only sparse feature set from the initial tests.

This led to the results displayed below:

```
Epoch 1/30
loss: 2029.7210 - mean_absolute_error: 7.4127
...
Epoch 30/30
loss: 5.1945 - mean_absolute_error: 0.4809
Final MSE, MAE: [3.0843481947260663, 0.44543924916379807]
```

These are still abysmal results, considering the outputs range is $[0, 1]$ and therefore a mean absolute error of 0.44 is essentially equivalent to guessing from a uniform distribution.

The baseline was also implemented. However, running it gave insight to a core problem with my data. Only 4 songs shared an artist with other entries. For the future, it is essential that I further clean up all my data. For this, I have begun to write a series of scripts.

3 Future Steps and Considerations

1. While the scraping of lyrics worked in general, I will be cleaning up the data better for future iterations. Some inconsistencies in the formatting used by the Genius community led to the failure of some processing assumptions. A unification of some values, such as removing secondary artists from the artist field, will allow for a less sparse input matrix. For the most popular words in songs, for example, I intend on removing common words such as "and" and "I".

2. The primary future objective is to incorporate a recurrent layer into the neural network being used in order to take into account features that repeat for each line in the lyrics. The goal with this would be to take into account complex "temporal" features, with ease.

3. Much more general neural-net testing must be conducted. Optimizing for number of epochs, batch sizes, number of hidden nodes, loss functions, activation functions, etc.

4. Start to incorporate rhyme features and develop some new ones!

4 Data Sources

The data required for this project will be primarily extracted via API of official sources such as Billboard and Genius. For data regarding critics reviews, web scraping will be necessary.

- Lyrics: Genius API docs.genius.com

- Semantics: NRC Word-Emotion Lexicon (EmoLex) will be used to associated the lyrics to emotions.

- Billboard: extra-official API github.com/guoguo12/billboard-charts

- Critics Reviews: webscraping AOTY.com for a single critics score, composed of all major critic magazines that reviewed the song

- Rhymes: previous CS 221 project which created a freestyle-AI will be used as a basis to extract rhymes from song lyrics (see literature review).

5 Relevant Literature

- Keras Docs: keras.io
- Hidden Layer Heuristics: https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw
- EmoLex: arxiv.org/pdf/1308.6297.pdf
- Rhyme and Style Features: http://www.ifs.tuwien.ac.at/ mayer/publications/pdf/may_ismir08.pdf
- Extracting Rhymes: https://worksheets.codalab.org/worksheets/0xa30cf00878ba4a85ab239b40fc9b5818/

6 Appendix I: Machine Learning Code

```
\\ learn.py, the actual machine learning

from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
import numpy as np
import features
# Consistent testing!
np.random.seed(7)

X, Y = features.getDataset("")
num, dim = X.shape
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=3)
train_num = X_train.shape[0]

model = Sequential()
# Linear Regression
model.add(Dense(1, input_dim=dim, kernel_initializer='normal'))

# Single-layer Neural Network
# Input Layer
# model.add(Dense(5, input_dim=dim, kernel_initializer='normal', activation='relu'))
# Output Layer
# model.add(Dense(1))

# Default compilation
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae'])
model.fit(X_train.todense(), Y_train, epochs=30, batch_size=500)

# Evaluation
print model.evaluate(X_test.todense(), Y_test, verbose=0)
# prediction = model.predict(X_test.todense())
# for i, predict in enumerate(prediction):
#     print predict, Y_test[i]


\\ baseline.py, my baseline implementation

import re
import sqlite3
```

```
import pprint

pp = pprint.PrettyPrinter()
db = sqlite3.connect("data/hiphop")
c = db.cursor()

def calculateScore(peak, weeks):
    normalized_peak = ((101 - peak * 1.0)**2)/10000
    score = normalized_peak * weeks
    return score

def baseline(title, artist):
    # Escape Error
    if artist.find("\"") != -1 or title.find("\"") != -1: return 0
    same_artist = c.execute(''' SELECT peak, weeks FROM songs
                                WHERE artist = "{}" and title != "{}" '''.format(title, artist))
    songs = same_artist.fetchall()
    if songs:
        scores = [calculateScore(peak, weeks) for peak, weeks in songs]
        return sum(scores) / len(scores)
    else:
        return 0

def evaluate():
    songs = c.execute(''' SELECT title, artist, peak, weeks FROM songs''').fetchall()
    results = []
    for title, artist, peak, weeks in songs:
        actual = calculateScore(peak, weeks)
        base = baseline(title, artist)
        if base:
            results.append((actual, base))

    pp.pprint(results)
    mae = sum([a - b for a, b in results]) / len(results)
    print mae

evaluate()

\\ features.py, extraction of features

from collections import defaultdict
import sqlite3
import re
import ast
import pprint
import util
import numpy as np
pp = pprint.PrettyPrinter()

def calculateScore(song):
    _, _, _, peak, weeks = song
    normalized_peak = ((101 - peak * 1.0)**2)/10000
    score = normalized_peak * weeks
    return score
```

```
emolex = util.parseEmoLex("./data/emolex.txt")

acceptable_verse_types = ["[hook]", "[chorus]", "[verse]",
        "[bridge]", "[intro]", "[outro]",
        "[prechorus]", "[postchorus]", "[prehook]",
        "[posthook]", "[interlude]", "[refrain]", "[drop]"]

def extractFeatures(song):
    title, artist, raw_lyrics, _, _ = song
    lyrics = ast.literal_eval(raw_lyrics)

    verse_types = defaultdict(int)
    affect_categories = defaultdict(int)
    word_count = defaultdict(int)
    line_count = 0
    number_stanzas = -3

    for line in lyrics:
        if not line:
            # Number of verses
            number_stanzas += 1
        elif line[0] == "[" and line.lower() in acceptable_verse_types:
            verse_types[line.lower()] +=1
        else:
            # Emo-lex
            line_count += 1
            for word in line.lower().split(' '):
                word = re.sub(r"[^A-Za-z]+", '', word)
                word_count[word] += 1
                for affect in emolex[word]:
                    affect_categories[affect] += 1

    # Singleton Features
    features = {
            "word_count": sum(word_count.values()),
            "stanzas": number_stanzas,
            "avg_stanzas": sum(word_count.values()) / number_stanzas,
            "lines": line_count,
            "avg_line": sum(word_count.values()) / line_count
            }

    popular_words = {"pop_word " + word: 1 \
            for word in sorted(word_count, key=word_count.get, reverse=True)[:3]}

    # Normalization
    util.normalize_vector(verse_types)
    util.normalize_vector(affect_categories)

    return util.merge_dicts(verse_types, affect_categories, features)

# TODO:
# Number of distinct rhyming phonemes
# Number of rhyming syllable pairs
```

```
# Feature Extraction
def getDataset(limit):
    db = sqlite3.connect("data/hiphop")
    c = db.cursor()
    songs = c.execute(''' SELECT title, artist, lyrics, peak, weeks
                          FROM songs WHERE lyrics is not NULL {}'''.format(limit)).fetchall()

    scores = []
    raw_features = []
    for s in songs:
        raw_features.append(extractFeatures(s))
        scores.append([calculateScore(s)])

    from sklearn.feature_extraction import DictVectorizer
    vec = DictVectorizer()
    features = vec.fit_transform(raw_features)
    np_scores = np.array(scores)
    normed_scores = np_scores / max(np_scores)

    return features, normed_scores
```

7 Appendix II: Utils and Parsing Code

```
\\ util.py, helpful functions

from collections import defaultdict

def parseEmoLex(path):
    words = defaultdict(list)
    with open(path, 'r') as emolex:
        for line in emolex:
            word, category, flag = line.strip().split("\t")
            if int(flag):
                words[word].append(category)
    return words

def merge_dicts(*dict_args):
    result = {}
    for dictionary in dict_args:
        result.update(dictionary)
    return result

def normalize(vector):
    total = sum(vector.values())
    for k, v in vector.items():
        vector[k] = v * 1.0 / total

\\ lyrics.py, extracts lyrics

import config
import re
```

```
import sqlite3
import requests
from bs4 import BeautifulSoup
import pprint
pp = pprint.PrettyPrinter()

base_url = 'https://api.genius.com'
headers = {'Authorization': 'Bearer ' + config.access_token}

def searchGenius(term):
    search_url = base_url + "/search"
    params = {'q': term}
    response = requests.get(search_url, params=params, headers=headers)
    return response.json()['response']['hits']

# Use search term to find songs
db = sqlite3.connect("data/hiphop")
c = db.cursor()
songs = c.execute(''' SELECT title, artist FROM songs WHERE api_path is NULL ''').fetchall()
# Re-run until no results for throttling
if not songs:
    print "Done with api_path!"
for i, (title, artist) in enumerate(songs):
    hits = searchGenius(title)
    if hits:
        c.execute(''' UPDATE songs SET api_path=:path WHERE title=:title ''',
                {"path": hits[0]['result']['api_path'], "title": title})
    else:
        print title, artist
    if i % 100 == 0:
        db.commit()
db.commit()

# Hard-coded where search fails
# c.execute(''' UPDATE songs SET api_path=:path WHERE title=:title ''',
#         {"path": "/songs/2867962", "title": "PPAP (Pen-Pineapple-Apple-Pen)"})
# c.execute(''' UPDATE songs SET api_path=:path WHERE title=:title ''',
#        {"path": "/songs/58343", "title": "HYFR (Hell Ya F*****g Right)"})
# c.execute(''' UPDATE songs SET api_path=:path WHERE title=:title ''',
#         {"path": "/artists/139436" , "title": "Cash Me Outside (#CashMeOutside)"})

def lyrics_from_api(song_api_path):
    response = requests.get(base_url + song_api_path, headers=headers).json()
    path = response["response"]["song"]["path"]
    page = requests.get("http://genius.com" + path)
    html = BeautifulSoup(page.text, "html.parser")
    [h.extract() for h in html('script')]
    lyrics = html.find("div", class_="lyrics").get_text()
    return lyrics

def process_lyrics(lyrics):
    final = []
    for line in lyrics.split("\n"):
        if line and line[0] == "[":
```

```
            comment = line.split(" ")[0]
            valids = re.sub(r"[^A-Za-z]+", '', line.split(" ")[0])
            final.append("[{}]".format(valids))
        else:
            final.append(line)
    return final
    #return [line for line in lyrics.split("\n") if (line[0] != "[" if line else True)]

songs = c.execute(''' SELECT title, api_path FROM songs WHERE lyrics is NULL ''').fetchall()
# Re-run until no results for throttling
if not songs:
    print "Done with lyrics!"

lyrics = []
for i, (title, api_path) in enumerate(songs):
    print title
    raw = lyrics_from_api(api_path)
    lyrics = str(process_lyrics(raw))
    c.execute(''' UPDATE songs SET lyrics=? WHERE title=? ''', (lyrics, title))
    if i % 100 == 0:
        db.commit()
db.commit()

\\ songs.py, extracts billboard charts information

import sqlite3
from dateutil import parser
import billboard

# NOTE: using only peakPosition rather than average of positions
db = sqlite3.connect("data/hiphop")
cursor = db.cursor()
# cursor.execute('''
#         CREATE TABLE songs(id INTEGER PRIMARY KEY, title TEXT unique,
                            artist TEXT, peak INTEGER, weeks INTEGER)
#         ''')
# db.commit()

# Get all songs
final_date = parser.parse("2009-01-01") # end date
chart = billboard.ChartData('r-b-hip-hop-songs', "2012-01-01") # start date
while chart.previousDate:
    print chart.previousDate

    for song in chart:
        cursor.execute('''
        INSERT OR IGNORE INTO songs(title, artist, peak, weeks) VALUES(?, ?, ?, ?)
        ''', (song.title, song.artist, song.peakPos, song.weeks))
        db.commit()

    prev = parser.parse(chart.previousDate)
    if prev < final_date: break
    chart = billboard.ChartData('r-b-hip-hop-songs', chart.previousDate)
```