Predicting Song Popularity with Deep Learning

Ian Leefmans
New York University

Abstract

This paper investigates whether or not a song's popularity can be predicted via deep learning methods. This objective is attempted by applying a Convolutional Neural Network to spectrograms representing a song's frequencies and amplitudes over time. The network predicts the popularity of out of sample songs with roughly 67% accuracy.

1 Introduction

1.1 Motivation

Music is widely considered an art. It enthralls, inspires, entertains, and so much more. So often a song that becomes popular does so because it innovates and challenges the norm. It gives you a sound that you have never heard before. This, at least, was the assumption until it came to light that many successful artists did extensive research into the types of patterns, melodies and lyrics that created a hit song, and were very successful in doing so. Bruno Mars, with his many chart topping hits, is a prime example of this.

So, is there a science to what makes a song popular? Can one predict whether or not a song will be popular based on its patterns and composition? There certainly is significant monetary incentive within the music industry to be able to make such predictions. This would not only prevent artists from wasting time producing potential "duds", but also allow labels uncover talent much more easily. This paper investigates how deep learning techniques might be used to predict the popularity of a song.

1.2 Overview

In order to attempt to answer the very broad question stated above, it is necessary to narrow our focus and outline how to achieve this objective more specifically. First, a criterion must be specified for determining whether or not a song is popular. I determined that reaching the end of the year Billboard Top 100 chart is a worthy metric to use in order to gauge whether or not a song achieves lasting popularity. In order to capture the various melodies, notes, and patterns that that make a song unique, frequencies and their amplitudes are employed. These two measures quantitatively capture the sound one hears when listening to a song. Patterns in these two measurements over time can be visualized in a 3-dimensional representation which is often flattened into 2-dimensions with the use of a progressive color pallet. These 2 dimensional images, known as spectrograms, contain all of the auditory information that a song possesses. These spectrograms are pixelated and be used as input in a deep learning method. In this particular case

a Convolutional Neural Network (CNN) is employed as the learning method. A CNN is used for three reasons:

(1) Deep Neural Networks in general are suited for modeling the complex patterns that make up the intricate melodies in songs
(2) CNNs are particularly adept at dealing with input in the form of images
(3) The "pooling" feature of CNNs allows frequencies that are close together in time to be generalized, effectively reducing the number of parameters that need to be estimated.

2 Data and Preprocessing

2.1 Data Description

The input data used in this study consists of 803 hip-hop songs from the past 5 years. Songs range from extremely popular, chart topping songs, to obscure songs from artists' early mixtapes. Each song is assigned a value of 1 if it appears on the end of the year Billboard Top 100 list, and a value of 0 if it does not. Of the 803 songs, 176 were considered popular by this criterion.

2.2 Data Preprocessing

To ensure consistency of the way in which the sound of each song is stored, all songs were converted to the common mp3 format (some songs existed in the recently popular m4a format). Songs were then converted to the audio storing format ".wav", as this format is convenient for audio decomposition. Audio signal was then extracted from each song and demeaned. Two problems became evident at this stage:

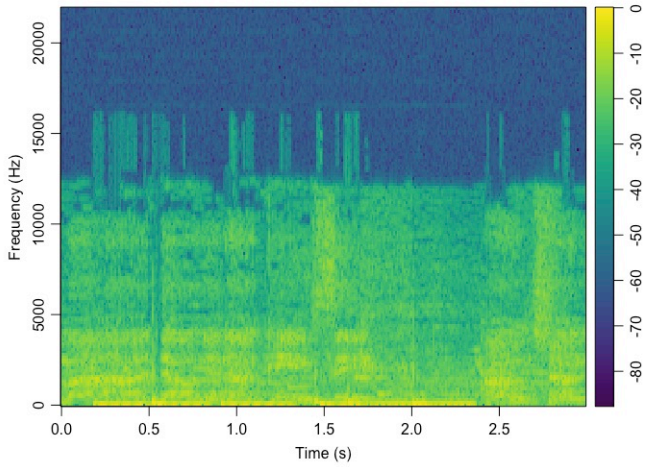
(1) The number of samples that produce audio signals was vastly too large to use as potential features in a the CNN. The number of parameters that would need to be estimated would be far too large to handle with my current computing power.
(2) Songs are both repetitive and constantly evolving. While the underlying beat and melody of the song exhibits an extremely repetitive pattern, both of these aspects are often coupled with very different instruments and vocals over the course of the song (i.e. the intro, chorus etc.). This results in both extreme overlap in the features used for prediction, and excess noise that makes it difficult to for the CNN to learn the overarching pattern of the song

Luckily, there is one solution that addresses both of these issues. By randomly sampling a short interval from the beginning, middle and end of each song the number of audio signals is vastly reduced, the general repetition of the beat and melody are captured, and the progression of the song is taken into account.[1] Each interval then represented its own individual song and was assigned the popularity value of the overall song. This not only nearly tripled the number of observations (now 2408) that would be used in the CNN, but also forced the CNN to discard the noise added as the song progressed and focus on the overarching pattern.

After random sampling each interval was passed through a function that created a spectrogram by dividing the interval into overlapping slices and then passing each slice through a Discrete Short-

[1] *Three intervals were sampled from every song with the exception of one song from which only two intervals were sampled due to the abnormally short duration of the song.*

Time Fourier Transform in order to extract the frequencies and their amplitudes at discrete points in time throughout the interval. A spectrogram like the one shown below was then produced from these values.



In each spectrogram Frequency is displayed on the vertical axis while time is displayed on the horizontal axis. The amplitude of frequencies at a particular level is displayed by the color pallet where higher amplitudes are represented with brighter color. These spectrograms were then converted to black and white images, normalized into squares, reduced in resolution, and pixelated.[2] The result of this was a [160 x 160] black and white image. Each pixel contained a numerical value representing where it is on the spectrum from black to white. These values were normalized to be between 0 and 1. Finally, this [160 x 160] matrix of values was "flattened" into a [25600 x 1] array. This array of values corresponding to the brightness of the pixels of each spectrogram were used as the input in the CNN. The 2,408 observations (with their array of input values and binary labels) were then divided into a training set of 1,800 observations and a test set of 608 observations. The popularity labels for each observation were also converted to a 2 x 1 array. Here the first value of the array was 1 if the song is considered "popular" (while the other value was zero) and the second value of the array was 1 if the song is considered "not popular" (while the other value was zero). The final structure of the data after preprocessing is summarized in the table below.[3]

	TRAINING OBSERVATIONS (1,800)	*TEST OBSERVATIONS (608)*
INPUT FEATURES	25600x1 array	25600x1 array
LABELS	2x1 array	2x1 array

[2] *All labels, axes, and color pallets were removed for this process.*

[3] *The majority of the code for the data preprocessing can be found in the Appendix under "R Code". The final steps of this procedure during which the data is pixelated, normalized etc. can be found in the Appendix under "Python Code".*

3 Empirical Estimation and Results

3.1 Model Architecture

As was previously stated, prediction was done using a CNN. The network consisted of an input layer with 25,600 neurons which fed into two hidden convolutional layer, followed two hidden dense fully-connected layers, and finally and output layer of 2 neurons. In each convolutional layer the filter size was [10 x 10] and the maximum value within each filter was selected. The stride size in this part of the layer was [1, 1] and multiple channels (to be trained separately) were passed to the next stage.[4] Padding was added in this stage in order to ensure each channel produced was the same size as was inputted. In the second stage of each convolutional layer pooling was performed where the pool size applied to each channel fed to it was [4 x 4] and the stride size was [4, 4] greatly reducing the number of feature passed out of the convolutional layer.

The dense, fully connected layers received an input of 64 [10 x 10] channels which are flattened into the 6400 neurons that made up the first dense layer. The second dense, fully connected layer was 500 neurons.

3.2 Optimization and Training

The 1,800 training observations were divided by random sampling into 45 mini-batches of 40 observations per mini-batch. Training was done to minimize the cross entropy loss function:

$$Cost = -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ji} log\left(\hat{y}_{ji}\right) + \left(1 - y_{ji}\right) log\left(1 - \hat{y}_{ji}\right) \qquad (1)$$

where:

$$M = number\ of\ mini - batches$$
$$N = number\ of\ terminal\ neurons$$

The optimization method used to minimize the cost function over each mini-batch is Adaptive Momentum Estimation (Adam):

$$\vec{\boldsymbol{w}}^{(\tau+1)} = \vec{\boldsymbol{w}}^{(\tau)} + \Delta\vec{\boldsymbol{w}}^{(\tau)} \qquad (2)$$

where:

$$\Delta\vec{\boldsymbol{w}}^{(\tau)} = -\eta\nabla E|_{\vec{\boldsymbol{w}}^{(\tau)}} + \mu\Delta\vec{\boldsymbol{w}}^{(\tau-1)}$$

$$E = Cost\ Function,\ \vec{\boldsymbol{w}}^{(\tau)} = vector\ of\ weights\ at\ iteration\ \tau$$

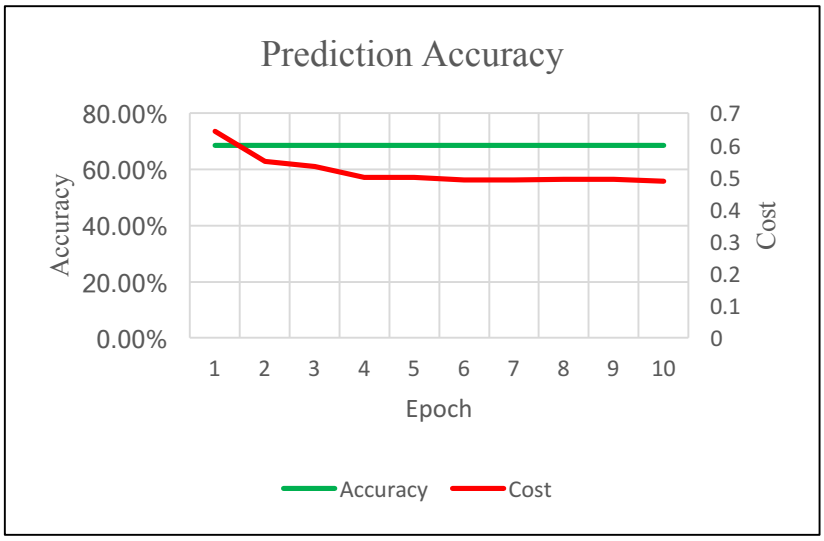
Adam includes the inertia term $\mu\Delta\vec{\boldsymbol{w}}^{(\tau-1)}$ used to add Momentum which increases the steps the algorithm takes in oscillatory areas of the cost function space, speeding up the optimization process. In addition, the learning rate η adapts as each iteration of the algorithm is calculated. Dropout is also employed at every fully connected layer of the network to help prevent overfitting

[4] *32 channels were created in the first convolutional layer. 64 channels were created in the second convolutional layer from the 32 channels passed to it from the first layer.*

during training and to help the network learn the most essential features of the data.[5] After the optimization process iterated through all of the mini-batches, quality of the model was assessed by feeding the test observations into the CNN and calculating the percentage for which the network accurately predicted the popularity label. This process was repeated over 10 epochs where the new mine-batches were re-randomly sampled every epoch, and the weights of the CNN were continually adjusted.[6]

3.3 Results

The predictive results from the CNN are summarized in Table A which can be located in the Appendix. The chart below visualizes the cost and accuracy of the model over the 10 epochs.



While the cost modeled by the cross entropy function displays an immediate decrease during the first few epochs there is no significant decrease in cost after the 4th epoch. The decrease in cost over the 10 epochs at no point impacts the accuracy of the model. This is a result of the output neurons becoming purer and purer over the 10 epochs while the classification of the test observations never changes. The overall out of sample accuracy of this model is fairly good as it correctly determines whether a song is popular or not roughly 67% of the time. This is far better than than a random flip of a coin, and should be viewed as an overall success given the limitations of the model I will discuss next.[7]

[5] *The probability a neuron is kept was 50%*

[6] *The general structure of the CNN and the optimization process follows the tensorflow tutorial outlined by the website Adventures in Machine learning but alters much of the code to fit this paper's purpose. The tutorial can be located at: https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-tensorflow/*

[7] *It should be noted that a confusion matrix would be very instructive in assessing where and how the model failed in predictability. Due to technical issues I was not able to include it in this paper but do acknowledge the gap it leaves in my model assessment.*

4 Limitations

Despite the overall success of the model in predicting the popularity of a song, there are a number of limitations from which this model and method suffers:

(1) **Small Sample Size**: Even if the 803 songs were sliced and sampled to yield 2,408 observations, this is still a relatively small number of samples for training and testing a CNN. The small test set of 608 observations in particular suggests viewing these successful results with caution.

(2) **Computing power**: Initially a CNN with more convolutional layers and dense layers was planned to be used to account for the immense complexity of each spectrogram. This was computationally unfeasible given the computing power of the device that was used. In addition, the issue of small sample size is compounded by the fact that a possible solution (or at least alleviation) to the problem was rendered unfeasible by this lack of computing power. This possible solution being taking more than 3 random slices from each song, vastly increasing the sample size.

(3) **The Nature of Music**: While these results are very encouraging, they must be viewed with skepticism due to the nature of what makes music popular. So often a song becomes popular because it introduces something never heard before. This is how new musical styles are created. While many songs imitate these new styles and become popular as well, this innovation renders this model useless in identifying new trends in music.

5 Conclusion

In spite of the limitations listed above this model marks a good starting point for popularity prediction in music. Possible jumping-off points for improving this model begin with addressing the limitations above. In addition, restricting the sample to music from a single year might further improve the accuracy of prediction. This would hopefully eliminate any structural shifts in the style of popular music. Overall, however, a prediction accuracy significantly above 50% is a success when attempting to predict something that relies entirely on an entire population's preferences.

Appendix

Table A

Epoch	*Cost*	*Accuracy*
1	0.650	0.686%
2	0.554	0.686%
3	0.514	0.686%
4	0.503	0.686%
5	0.501	0.686%
6	0.495	0.686%
7	0.487	0.686%
8	0.494	0.686%
9	0.486	0.686%
10	0.490	0.686%

R Code:

```
install.packages('tuneR')
install.packages('oce')

#############################################################################################
setwd("/Users/ianleefmans/Desktop/Project_Input_mp3")
counter<-0
specimport<-function(x) {
  snd<-x@left
  ind1<-seq(from=1, to=length(snd)/3, by=1)
  ind2<-seq(from=length(snd)/3, to=length(snd)*(2/3), by=1)
  ind3<-seq(from=length(snd)*(2/3)+30000, to=length(snd)*(2/3)+80000, by=1)
  set.seed(1)
  start1<-sample(ind1,1)
  set.seed(1)
  start2<-sample(ind2,1)
  set.seed(1)
  start3<-sample(ind3,1)
  end1<-start1+132299
  end2<-start2+132299
  end3<-start3+132299
  snd1<-snd[start1:end1]
  snd2<-snd[start2:end2]
  snd3<-snd[start3:end3]

  mat1<-cbind(snd1,snd2,snd3)

  fs<-x@samp.rate
  for (i in seq(1:ncol(mat1))){
    ssnd1<-mat1[,i]-mean(mat1[,i])
    spec<-signal::specgram(x=ssnd1,n=300,Fs=fs,window=300,overlap=-150)

    P<-abs(spec$S)
    P<-P/max(P)
    P<-10*log10(P)
    filename<-paste(i+counter,".jpg",sep="")
    jpeg(filename)
    oce::imagep(x=spec$t, y=spec$f, z=t(P), col=oce::oce.colorsViridis, axes=FALSE, drawPalette = F, decimate = F, mar=c(0,0,0,0))
    dev.off()
  }
}
```

```
################################################################################################
num1<-list.files(pattern='*.mp3')
for (i in 1:length(num1)) {
  setwd("/Users/ianleefmans/Desktop/Project_Input_mp3")
  data<-assign(num1[i],tuneR::readMP3(num1[i]))
  setwd("/Users/ianleefmans/Desktop/Project_Input_jpg")
  tryCatch(specimport(data),error=function(e) {
    print(paste(e,num1[i], sep=""))
  }, warning=function(w){

    }
  )
  print(num1[i])
  counter<-counter+3
  rm(list=ls()[! ls() %in% c("counter","specimport","num1")])
}
################################################################################################
setwd("/Users/ianleefmans/Desktop")
label<-read.csv('machine learning project output.csv',header=TRUE)

expand_by_3<- function(x){
  nvector<-c()
  for (i in seq(1:length(x))){
    if (i == 751){
      nvector<- append(nvector, c(x[i], x[i]))
    }else{
      nvector<- append(nvector, c(x[i], x[i], x[i]))
    }
  }
  nvector
}

label2<- expand_by_3(label[,3])
label1<- expand_by_3(label[,2])

flabel<-cbind(label1,label2)

write.csv(flabel, file = "Project_Labels.csv")
```

Python Code:

```
import numpy as np
import pandas as pd
from PIL import Image
import tensorflow as tf

pic_labels = pd.read_csv('Project_Labels.csv')
year_labels = []
for i in range(len(pic_labels['Year'])):
    year_labels.append(pic_labels['Year'][i])
pic_labels = None

input_list = []
for i in range(2409):
    try:
        img = str(i+1)+".jpg"
        im = Image.open(img, 'r')
        im = im.convert(mode="L")
        im = im.resize((160, 160), Image.ANTIALIAS)
        pix_val = list(im.getdata())
        pix_val = np.array(pix_val)
        input_list.append(pix_val)
```

```
        print("done", i)
    except FileNotFoundError:
        print("file", i, "not found")
        pass
pix_val = None

input_list = np.array(input_list)
max_of_lists = []
for i in input_list:
    max_of_lists.append(max(i))
input_list = input_list/219

train_input_and_year_labels = [input_list[0:1800], year_labels[0:1800]]
test_input_and_year_labels = [input_list[1800:2408], year_labels[1800:2408]]

year_labels = None
input_list = None


def mini_batch_sampling(ipt, b_size=40):
    new_list1 = []
    new_list2 = []
    idx = []
    for q in range(len(ipt[0])):
        idx.append(q)
    for i in range(int(len(ipt[0])/b_size)):
        batchx = []
        batchy = []
        samp = np.random.choice(idx, size=b_size, replace=False)
        for j in range(b_size):
            batchx.append(ipt[0][samp[j]])
            batchy.append(ipt[1][samp[j]])
            idx.remove(samp[j])
        new_list1.append(batchx)
        new_list2.append(batchy)
    return new_list1, new_list2


batchx = None
batchy = None


def pick_batch(mini_x, mini_y, batch_index):
    return mini_x[batch_index], mini_y[batch_index]


def add_dimension(onedim):
    new_y = []
    for i in onedim:
        if i == 1:
            add = [i, 0]
            new_y.append(np.array(add))
        else:
            add1 = [i, 1]
            new_y.append(np.array(add1))
    new_y = np.array(new_y)
    return new_y


add1 = None
```

```
learning_rate = 0.0001
epochs = 10
batch_size = 40

x = tf.placeholder(tf.float32, [None, 25600])
x_shaped = tf.reshape(x, [-1, 160, 160, 1])
y = tf.placeholder(tf.float32, [None, 2])
keep_prob = tf.placeholder(tf.float32)


def create_new_conv_layer(input_data, num_input_channels, num_filters, filter_shape,
pool_shape, name):
    conv_filt_shape = [filter_shape[0], filter_shape[1], num_input_channels,
num_filters]
    weights = tf.Variable(tf.truncated_normal(conv_filt_shape, stddev=0.03),
name=name+"_W")
    bias = tf.Variable(tf.truncated_normal([num_filters]), name=name+"_b")
    out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding='SAME')
    out_layer += bias
    out_layer = tf.nn.leaky_relu(out_layer, alpha=0.05)
    ksize = [1, pool_shape[0], pool_shape[1], 1]
    strides = [1, 4, 4, 1]
    out_layer = tf.nn.max_pool(out_layer, ksize=ksize, strides=strides,
padding="SAME")
    return out_layer


# filter size = 10 x 10, pool size = 4 x 4
layer1 = create_new_conv_layer(x_shaped, 1, 32, [10, 10], [4, 4], name='layer1')
layer2 = create_new_conv_layer(layer1, 32, 64, [10, 10], [4, 4], name='layer2')


flattened = tf.reshape(layer2, [-1, 10 * 10 * 64])
flattened = tf.nn.dropout(flattened, keep_prob=keep_prob)
wd1 = tf.Variable(tf.truncated_normal([10*10*64, 500], stddev=0.03), name='wd1')
bd1 = tf.Variable(tf.truncated_normal([500], stddev=0.01), name='bd1')
dense_layer1 = tf.matmul(flattened, wd1)+bd1
dense_layer1 = tf.nn.leaky_relu(dense_layer1, alpha=0.05)
dense_layer1 = tf.nn.dropout(dense_layer1, keep_prob=keep_prob)

wd2 = tf.Variable(tf.truncated_normal([500, 2], stddev=0.03), name='wd3')
bd2 = tf.Variable(tf.truncated_normal([2], stddev=0.01), name='bd3')
dense_layer2 = tf.matmul(dense_layer1, wd2) + bd2
dense_layer2 = tf.nn.leaky_relu(dense_layer2, alpha=0.03)
y_ = tf.nn.softmax(dense_layer2)

cross_entropy =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=dense_layer2, labels=y))
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
init_op = tf.global_variables_initializer()


with tf.Session() as sess:
    sess.run(init_op)
    # number of batches = 45
    total_batches = int(1800/40)
    for epoch in range(epochs):
```

```
        avg_cost = 0
        sampling_x, sampling_y = mini_batch_sampling(train_input_and_year_labels,
b_size=batch_size)
        for i in range(total_batches):
            batch_x, batch_y = pick_batch(sampling_x, sampling_y, i)
            batch_x = np.array(batch_x)
            batch_y = np.array(batch_y)
            batch_y = add_dimension(batch_y)
            _, c = sess.run([optimizer, cross_entropy], feed_dict={x: batch_x, y:
batch_y, keep_prob: 0.5})
            avg_cost += c/total_batches
            test_year_input = np.array(test_input_and_year_labels[0])
            test_year_labels = np.array(test_input_and_year_labels[1])
            test_year_labels = add_dimension(test_year_labels)
            test_acc = sess.run(accuracy,
                                feed_dict={x: test_year_input, y: test_year_labels,
keep_prob: 1})
        print("Epoch:", (epoch + 1), "cost =", "{:.3f}".format(avg_cost), "test
accuracy:", "{:.3f}".format(test_acc))
    print("\nTraining Complete")
    print(sess.run(accuracy, feed_dict={x: test_year_input, y: test_year_labels,
keep_prob: 1})*100, "%")
```