Predicting Hit Songs:

A Scikit-Learn Approach

David Wiley
Dr. Vasilis Zafiris

College of Sciences & Technology
University of Houston - Downtown
United States of America
December 2019

Abstract

In this project, computational experiments are performed on each and every phase of the entire data science pipeline. Specifically, starting with the 55,000+ Song Lyrics[3] data set and a data set collected from the Hot 100 list on Billboard[4]; we utilize descriptive statistics and visualization to better understand the data. We then perform data transformations to better expose the structure of the prediction problem to the modeling algorithms, and apply Logistic Regression Classification, Stochastic Gradient Descent Classification, k-Nearest Neighbors Classification, Random Forest Classification, and Linear Discriminant Analysis algorithms on the data and select those that perform better for further mining and understanding of the data. Finally, we use algorithm tuning and select the methods to get the most out of the well-performing algorithms on the data.

Contents

Chapter 1

Introduction

The IFPI's (International Federation of the Phonographic Industry) Global Music Report 2019 [1] states the total revenue for 2018 was US$19.1 billion. The report also states the global recorded music market grew by 9.7% since 2017. With an increase in recorded music and the potential revenue it generates it is evident that being able to predict if a song will be a hit could prove to be a powerful tool. The purpose of this project is to explore this practice using data analysis and machine learning methods.

There are multiple ways to implement and program machine learning. One could program a machine learning model from scratch but there are also various software programs and packages that make life a lot easier. The objective of this project is to get use a familiar applying machine learning methods using the Scikit-Learn [2] package on data.

Chapter 2

Data

The data used in this project was a combination of 55,000+ Song Lyrics [3] data set downloaded from `www.Kaggle.com` and a scraped data set from the Billboard Hot 100 [4]. In 1958, the Billboard published the Hot 100 chart. The chart is released every Saturday and has since become the industry standard. The Hot 100 will be used in this project as a measurement of popularity. If the track has appeared on the list, it will be considered as popular and referenced as such throughout this paper. If the track was in the Billboard data set, it was given a popularity value of 1 and 0 otherwise.

Popular Data

The popular data was scraped from the Billboard Hot 100 website using Python methods in Jupyter. Since the Hot 100 list is released every week on Saturday, the dates focused start from January, 1, 2000. This specific date was used because musical styles and preferences change throughout the years, so we did not want to reference songs that were a little "outdated" in comparison to today's musical styles. The latest date used is December 29, 2018, the last Saturday of that year. This is because our unpopular data, the data set from Kaggle, only extends to the year 2018. The data extracted includes: track name, artist name, current position for that week, position the previous week, highest position it has reached, weeks it has been on the chart, the year of the position week, month of the position week, and day of the week. Since songs can stay on the chart for multiple weeks at a time and our objective does not concern when the song is on the chart, only that

it is on the chart, duplicates were removed without regard of week. After removing duplicates there were ∼7,800 songs.

Unpopular Data

The unpopular data is pulled from Kaggle. This data set has 57,650 songs where each include: artist name, track name, and lyrics. The data was designed for Natural Language Processing analysis, so it includes the lyrics for those songs and this specific data was deleted because it is not needed for our purposes. To get the dates for these songs we used the `spotipy` [5] package to search and populate each song with its release date. The songs were then filtered for the years 2000-2018. This left over 15,000 songs in the unpopular data set. Using Python methods, songs were selected random, with an even amount from each year, to match the size of the popular data set.

Features

The features that will be used in our model will be extracted from Spotify using the `spotipy` package in Python. The table and definitions for the measurements used for these features are provided by Spotify[6]. The calculations on how these measurements are taken are unknown for this project.

Table 1.
Audio Features Object

Key	Value Type	Description

energy	*float*	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
liveness	*float*	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
tempo	*float*	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.

speechiness	*float*	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
acousticness	*float*	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
instrumentalness	*float*	Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.
time_signature	*int*	An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

danceability	*float*	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.
key	*int*	The 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.
duration_ms	*int*	The duration of the track in milliseconds.
loudness	*float*	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typical range between -60 and 0 db.
valence	*float*	A 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).

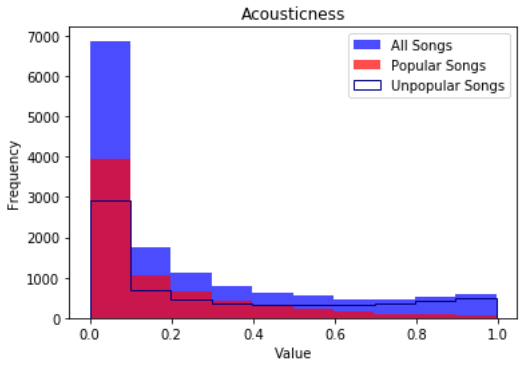
mode	*int*	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.

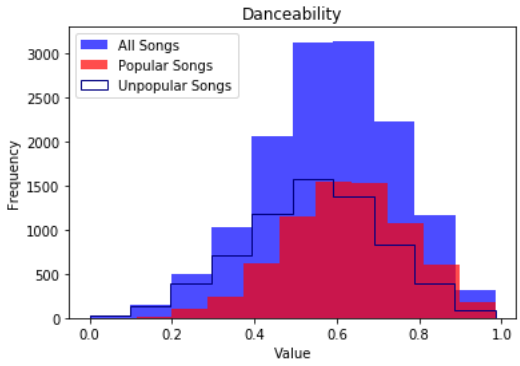
Chapter 3

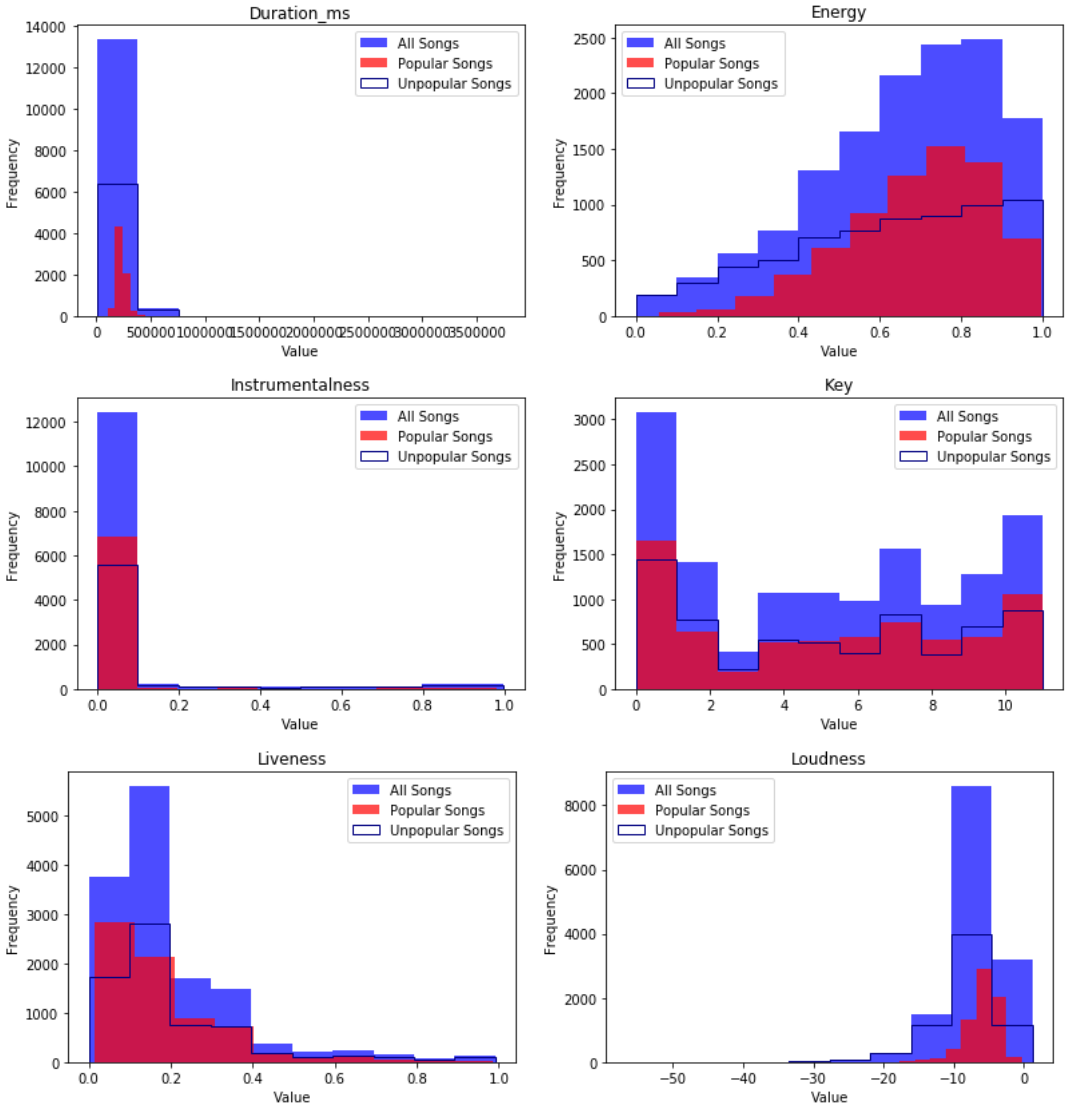
Data Analysis

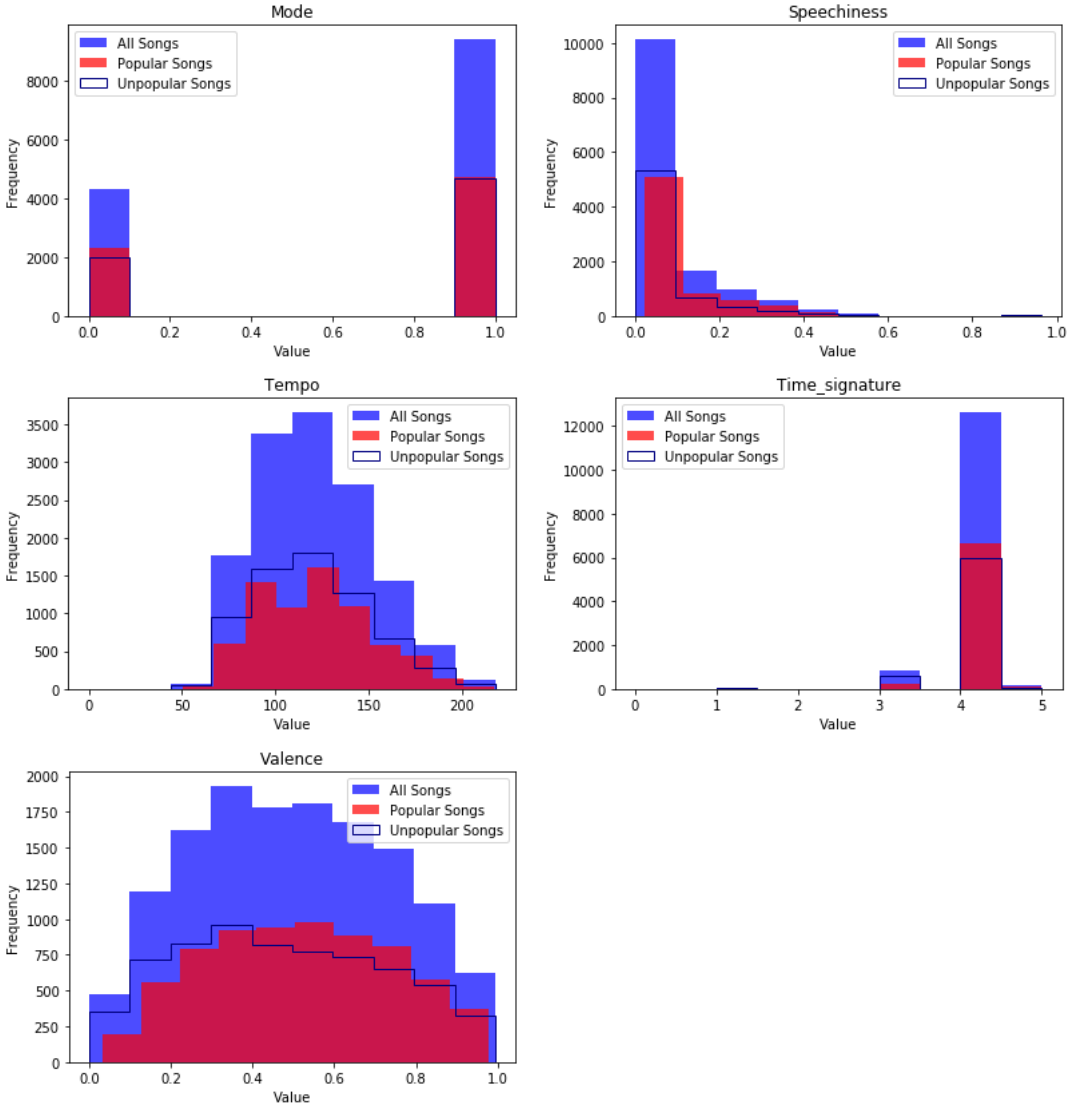
Distributions of Features

To get a better understanding of the features, the distributions of each were graphed. This allows us to visualize the difference in values between the popular and unpopular songs.






Duration_ms
Energy
Instrumentalness
Key
Liveness
Loudness
All Songs
Popular Songs
Unpopular Songs
Frequency
Value



Feature Correlation

To ensure the model will work properly and most efficiently, analysis of the data is a big part of machine learning. A model with too many unnecessary variables or features could lead to over-fitting the data which would perform well on the training set but poorly on the testing set. A model with not enough features could lead to under-fitting which over simplifies the models

and produces wrong predictions. To get a better idea of which features would work best with the model, the correlation between the popularity and each feature was calculated. Shown in Figure 3-1 below:

popularity	1.000000
loudness	0.299264
danceability	0.253639
energy	0.165695
time_signature	0.105563
speechiness	0.092811
valence	0.090136
tempo	0.030475
key	0.015335
mode	-0.029732
duration_ms	-0.040850
liveness	-0.065040
instrumentalness	-0.224926
acousticness	-0.242218

Figure 3-1. Correlation between popularity and features.

Too simplify the model and keeping the complexity to a minimum while trying to prevent under-fitting, the features with the smaller correlations were dropped from the data set. From analyzing the correlation; the tempo, key, mode, and duration_ms features were dropped. Leaving the features: loudness, danceability, energy, time_signature, speechiness, valence, liveness, instrumentalness, acousticness. These are the features that will be used in the model.

Chapter 4

Methods

Overview

There are numerous types of models the Scikit-Learn package offers that can be applied which allows users to choose the best model for their data. In this project, we will be using the *supervised learning* method because we have a known dependent variable, popularity, in our data set. More specifically, the *classifying* method will be used because we are classifying each song as popular or unpopular. Since we will be classifying a song, we will be utilizing the Logistic Regression, Stochastic Gradient Descent Classifier, k-Nearest Neighbors, Random Forest Classifier, and Discriminant Analysis classifying models. Each model has multiple parameters that can be changed and adjusted to fine tune the model to fit best with the data and get the desired results.
The data is split into the training set and testing set using the `.train_test_split` function within the Scikit-Learn package. The function allows users to choose the size of their sets by using a percentage. For this model, 80% of the combined data set was used for training and 20% was used for testing. The function shuffles the data then randomly selects the designated percentage for the training and testing subsets.

Scikit-Learn Model Parameters

Each model has multiple parameters that can be changed and adjusted to fine tune the model to fit best with the data for desired results.

Penalty Functions

Penalty functions are a hyperparameter within each function to apply regularization. Regularization prevents over-fitting which happens when there is too much noise in the data. According to the documentation there are three types used in the Scikit-Learn package: L1, L2, and Elastic-net.

- L1 (Lasso Regression) - Adds "square magnitude" of the coefficient as the penalty term to the loss function. It attempts to calculate the coefficients that are unimportant as a zero which eliminates unnecessary features.
- L2 (Ridge Regression) - Adds "absolute value of magnitude" of coefficient as the penalty term to the loss function. It will have small coefficients for unimportant features but it will not eliminate those features.
- Elastic-net - Essentially combines L1 and L2

Solvers

In the Scikit-Learn package there are multiple solvers[10]:

- newton-cg (Newton-Conjugate Gradient) - Newton methods use an exact Hessian matrix. It's slow for large data sets because it computes the second derivatives.
- lbfgs (Limited-memory Broyden-Fletcher-Goldfarb-Shanno - Approximates the second derivative matrix updates with gradient evaluations. It stores only the last few updates, so it saves memory. It isn't fast with large data sets.
- liblinear (Library for Linear Classification) - Uses a coordinate descent algorithm. Coordinate descent is base on minimizing a multivariate function by solving univariate optimization problems in a loop. In other words, it moves toward the minimum in one direction at a time.
- sag (Stochastic Average Gradient Descent) - A variation of gradient descent and incremental aggregated gradient approaches that uses a random sample of previous gradient values. Fast for big data sets.

- Saga - Extension of sag that also allows for L1 regularization. Should generally train faster than sag.

Logistic Regression

The underlying function in Logistic Regression is a Linear Regression function. *Linear Regression* takes the weighted sum of input features from data and returns a value. The *Logistic Regression* takes that weighted sum and passes it through another function, the *Sigmoid function*. In Hands-On Machine Learning with Scikit-Learn and Tensorflow [7], Geron uses the Sigmoid function as a function of `t`, as seen below in Figure 4-1:

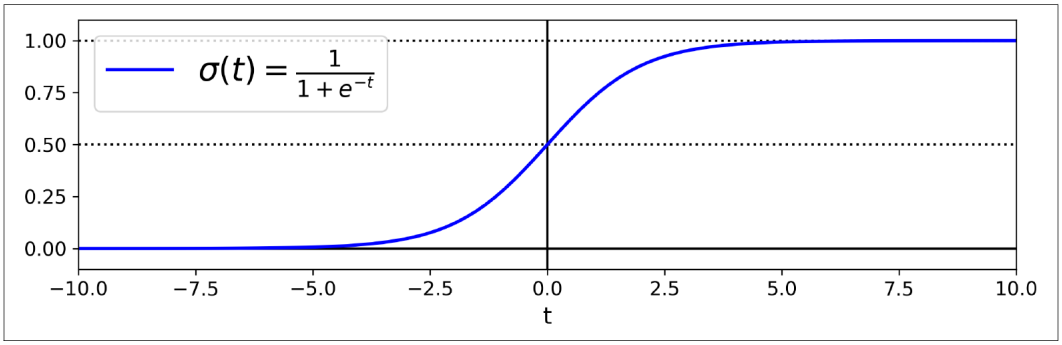


Figure 4-1. Logistic function in Geron's book.

This allows the Logistic Regression to model the probability of an event being a success or failure, 1 or 0 respectively, based on the features in the data. In this case we will be modeling the probability whether a song will be a hit based on the characteristics of that song. Thus, we are predicting the binary response variable, song popularity, based off the effect of the audio features of the song, provided by Spotify.

Stochastic Gradient Descent

The basic idea behind Stochastic Gradient Descent is *Gradient Descent*. Gradient Descent is an optimization algorithm that iterates to adjust the parameters within a function until it finds the minimum of the cost function. This

is achieved by calculating the partial derivative of the cost function for the whole data set. Note that the main parameter to be adjusted within a gradient descent is the *learning rate*. The learning rate is the step size taken when descending toward the minimum (see Figure 4-2). However, using Gradient Descent can very computationally costly if the data set is very large. Stochastic Gradient Descent can be used to cut down on time and computational cost.

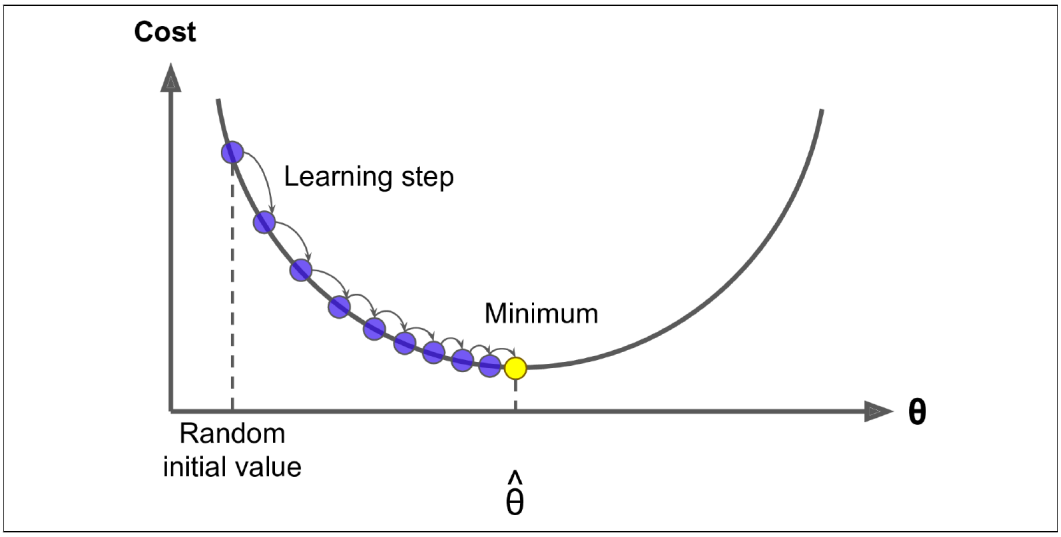


Figure 4-2. Gradient Descent pictured in Geron's book.

In *Stochastic Gradient Descent*, a random instance of the data set is used to compute the descent. This will be more efficient than calculating the descent for the whole data set. Also, within Scikit-Learn the step size is initially chosen at random and the algorithm will begin there and update the descent as it iterates. Scikit-Learn also takes a parameter of the maximum iterations to complete until it finishes. By default this is set to 1000 iterations but can be changed by the user.

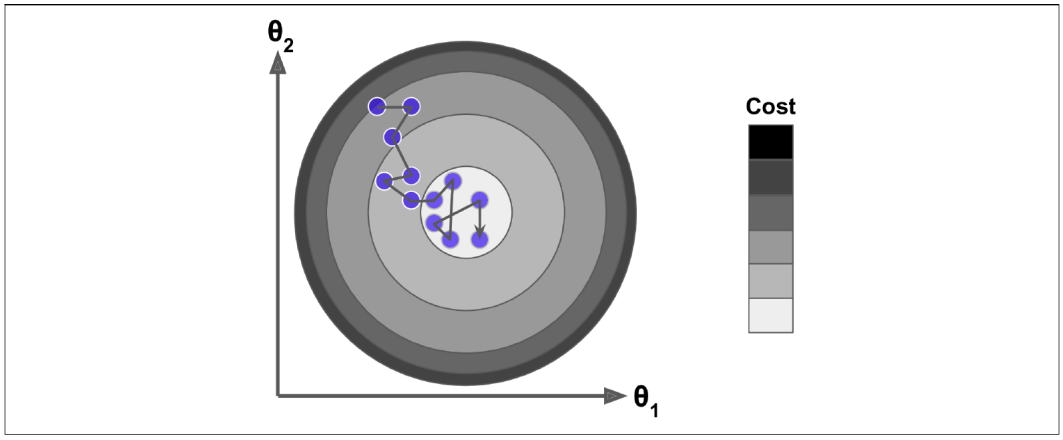


Figure 4-3. Stochastic Gradient Descent pictured in Geron's book.

Scikit-Learn Learning Rates

According to the documentation, the learning rates in Scikit-Learn are:

- Constant = eta0
- Optimal = 01.0 / alpha * (t + t0)) where t is chosen by a heuristic proposed by Leon Bottou.
- Invascaling = eta0 / pow(t, power_t)
  Where,
  - Eta0 is the initial learning rate (default is 0)
  - Power_t used in invascaling (default to 0.5)

k-Nearest Neighbors

In the *k-Nearest Neighbors* model, the user selects a value, k, so that the algorithm will know how many *neighbors*, or data points, to compare to the data point being predicted. Then the distance is calculated between each data point and the predicted data point. The distances are essentially sorted in increasing order and the model sets the class of the predicted data point to the most common class of the first k number of neighbors. For example, as shown in Figure 4-3 [8], if we wanted to classify the green circle using k=*3*,

we would classify the circle to be a triangle. However, if we choose *k=5* or *k=11*, we would classify the circle to be a square.

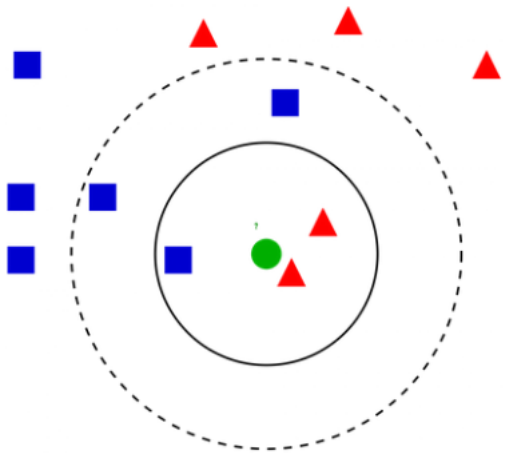


Figure 4-4. A visualization of k-Nearest Neighbors.

Random Forest Classifier

The *Random Forest Classifier* in an ensemble of multiple Decision Tree Classifiers. A *Decision Tree Classifier* divides the data space into separate class identifying groups. One can imagine a Decision Tree as a flowchart[9] of decisions of certain criteria. The issue with Decision Trees is due to their simplicity they can lead to over-fitting. Therefore a Random Forest Classifier solves this issue by combining multiple Decision Trees and decides on the final class of the data point being predicted.

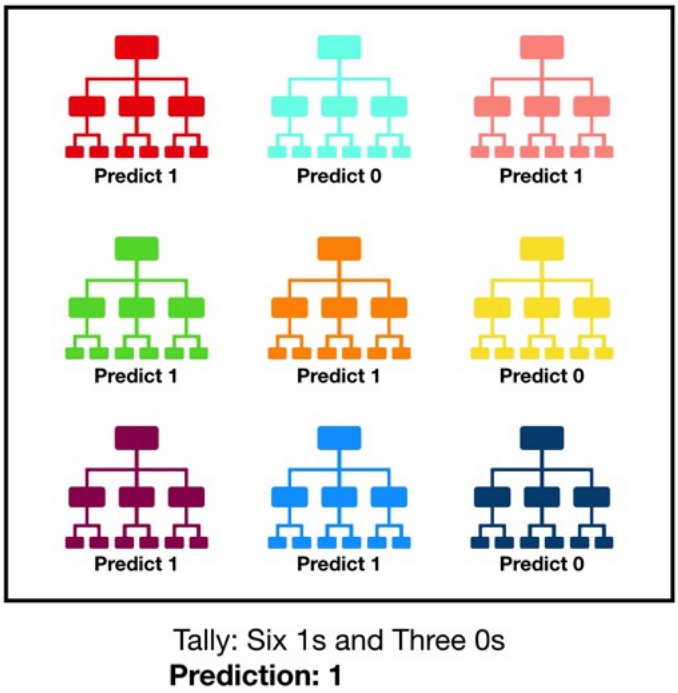


Figure 4-5. A visualization of a random forest.[9]

Discriminant Analysis

Discriminant Analysis is a type of dimensionality reduction classification. As explained in Geron's book, Discriminant Analysis takes the most descriptive features of the training set data and projects it onto a hyper-plane defined by those features. The purpose of the projection is to keep distinct classes as far apart as possible for each of classification.

Parameters

- **svd**(default) - Singular value decomposition. Good for data sets with a lot of features.
- **lsqr** - Least squares solution
- **eigen** - Eigenvalue decomposition

Chapter 5

Results and Conclusion

For each method, multiple parameters for each model were iterated through to test which would provide the best prediction. The results for each model were then sorted by decreasing order and the top few best results were chosen. Note that for some methods there were hundred of results. The accuracy is calculated by using the function `.accuracy_score` in the Scikit-Learn package which calculated the percentage of correctly labeled predictions.

Results

Logistic Regression Classifier

Penalty	**Solver**	**Accuracy**
L2	Limited-memory Broyden-Fletcher-Goldfarb-Shanno	70.16%
L2	Newton-Conjugate Gradient	70.13%
L1	Library for Large Linear Classification	70.13%
L2	Library for Large Linear Classification	70.13%
L2	Stochastic Average Gradient Descent	69.91%
L1	saga	69.87%
L2	saga	69.76%

Stochastic Gradient Descent

Penalty	Alpha	Rate	Accuracy
Elastic-net	0.0291	Optimal	69.25%
Elsatic-net	0.0171	Optimal	69.03%
L1	0.0001	Optimal	68.89%
Elastic-net	0.0181	Optimal	68.71%
Elastic-net	0.0121	Optimal	68.67%
L1	0.0091	Optimal	68.63%

k-Nearest Neighbors

Neighbors	Weights	Accuracy
250	Uniform	62.91%
250	Uniform	62.88%
200	Uniform	62.66%

Random Forest Classifier

Estimator	Accuracy
80	71.91%
375	71.77%
175	71.66%
800	71.66%
300	71.66%
200	71.66%

Discriminant Analysis

Solver	Accuracy
Singular Value Decomposition	69.47%
Least Squares	63.79%
Least Squares	63.79%
Least Squares	63.5%
Eigenvalue Decomposition	63.5%

Conclusion

From the results we conclude the best performing method is the Random Forest Classifier with an accuracy of ∼72%. The worst performing method was the k-Nearest Neighbors Classifier with an accuracy of ∼63%. To improve the models, future considerations could include but are not limited to tweaking parameters or using other models that are available within the Scikit-Learn package. The data set could also possibly extend the dates observed.

References

[1] International Federation of the Phonographic Industry (2019, April 2). *IFPI Global Music Report 2019*. Retrieved from https://www.ifpi.org/news/IFPI-GLOBAL-MUSIC-REPORT-2019

[2] Pedregosa et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research(12)*, pp. 2825-2830. Retrieved from http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[3] Kuznetsov, S. (2016). *55000+ Song Lyrics* (1) [CSV file with songs, artists and lyrics]. Retrieved from https://www.kaggle.com/mousehead/songlyrics

[4] Top 100 Songs: Billboard Hot 100 Chart (2000-2019). Retrieved from https://www.billboard.com/charts/hot-100

[5] Lamere et al. (2014). *Welcome to Spotipy!*. Retrieved from https://spotipy.readthedocs.io/en/latest/

[6] Spotify (2019). *Audio Features Object*. Retrieved from https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/

[7] Geron, A. (2019). *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. Sebastopol, CA: O'Reilly Media, Inc.

[8] Srivastava, T. (2018, March 26). *Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python & R)*. Retrieved from https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

[9] Yiu, T. (2019, June 12). *Understanding Random Forest - Towards Data Science*. Retrieved from https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[10] Hale, J. (2019, October 18). *Don't Sweat the Solver Stuff - Towards Data Science*. Retrieved from https://towardsdatascience.com/dont-sweat-the-solver-stuff-aea7cddc3451