Predicting the Danceability of Spotify Songs

Introduction

Data science, machine learning, and artificial intelligence are all buzzwords that have piqued the interest of statisticians, analysts, and scientists alike. However, these topics that form the intersection of computer science and statistics are accessible to more than your Jane Street quant. For this very reason, I sought to build a kernel that would be useful to the general populace by taking vast amounts of quantitative information and boiling it down into an evidence-based prediction. In order to achieve this goal, I first needed to choose a relevant dataset. I chose a Kaggle csv file that stored the features of more than 240k songs from Spotify. I eventually filtered down the data to ~4k songs. The criteria for a song to be included in the final data frame was that the genre needed to be either "Pop", "Dance", "Country", "Hip-Hop" and the popularity rating needed to be above 60 (out of 100). Each data point included features such as genre, acousticness, instrumentalness, loudness, danceability etc. Danceability was the feature that stood out the most to me, so I wrote three algorithms that sought to predict the danceability of songs from the dataset.

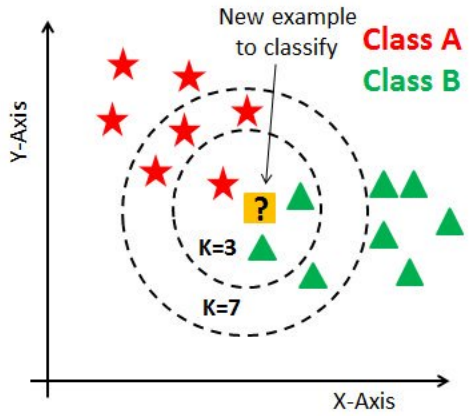
Given that danceability was recorded in the csv as a decimal value between 0 and 1, I implemented the regression algorithms that facilitated making a non-discrete prediction. I verified that the data set was not trivial by running linear regression. The accuracy associated this method was 22.7%. This low accuracy score indicates that the data was not trivial by linear separability. As for the application of this set, the success of this kernel has real world applications that include recommending songs for dance parties and playlists and determining what features of a piece factor into making it a danceable song. To measure performance, I will use scikit-learns "score" method which returns the coefficient R^2. This coefficient is given by 1 - u/v, where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). Additionally, I used the cross_val_score library that employs a variable number of splits on the data set to test the accuracy for my training more generally. Finally, I used the GridSearchCV library which assisted in finding the ideal set of hyperparameters for each of my different algorithms.

Description of the Algorithm

A) K Neighbors Regression

The first algorithm I implemented to predict the danceability was K Neighbors Regression. Unlike K Neighbors Classification that aggregates the classes of the k nearest neighbors to classify a query point, K Neighbors Regression uses supervised learning to make a prediction for features with continuous values. In our case, this feature was the danceability. In order to understand the functionality of this algorithm let us simplify our

dataset by graphing it on a 2d cartesian coordinate system. This reduction of the dataset calls for us to simplify each data point to only have 2 features for example, energy and instrumentalness. We take in k as a parameter that indicates how many points adjacent to the query point should be factored in to determine a value for our hypothesis class. We choose a distance method such as Euclidean or Manhattan distance to locate these adjacent points. Finally, we take the average of their associated danceability values and we use this as the danceability prediction of the query point. The following graphic illustrates the K Neighbors as described above:



The hyperparameters that were factored into making a determination of the hypothesis class include weights on the adjacent points, distance method, and k which represents the number of neighbors we will evaluate.

B) Kernel Ridge Regression

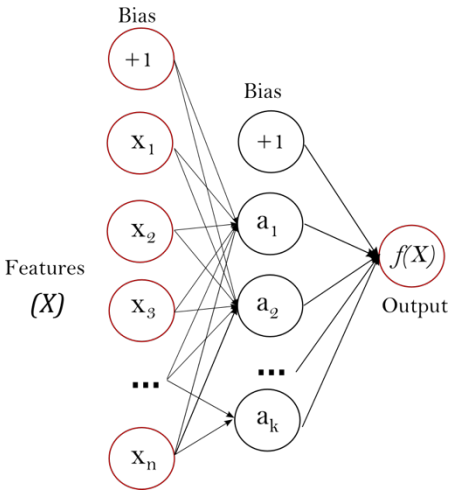
The second algorithm I implemented to predict the danceability was Kernel Ridge Regression. This algorithm employs least squares and regularization to learn a function given by a parameter called the kernel.

Tuning the hyperparameters proved to a crucial step of maximizing the accuracy of this algorithm. The hyperparameters factored into making a determination of the hypothesis class included the kernel, the degree (only used the polynomial kernel to define the degree of the polynomial), the alpha value used to reduce the variance of the estimates, and k which represents the number of neighbors we will evaluate.

C) Neural Network

The final algorithm I implemented to predict the danceability was a Neural Network. I used scikit learn's MLP Regressor class that learns the function $f(\cdot): R^m \rightarrow R^o$ which maps m dimensional input to o dimensional output. This algorithm utilizes a network of neurons

encapsulated in layers. The first layer is the input layer that includes all of the features. Between the output layer and the input layer are several layers known as hidden layers. In order to determine whether or not a nodes signal is passed on to deeper layers, we use the node's activation function. This activation function is determined the input value and the weights that either increase or decreases the nodes chance of propagation.



This algorithm witnessed a large number of hyperparameters that helped tune the accuracy of the prediction. Some of these hyperparameters include the activation function, the learning rate, the hidden layer size, the maximum number of iterations, and the solver function.

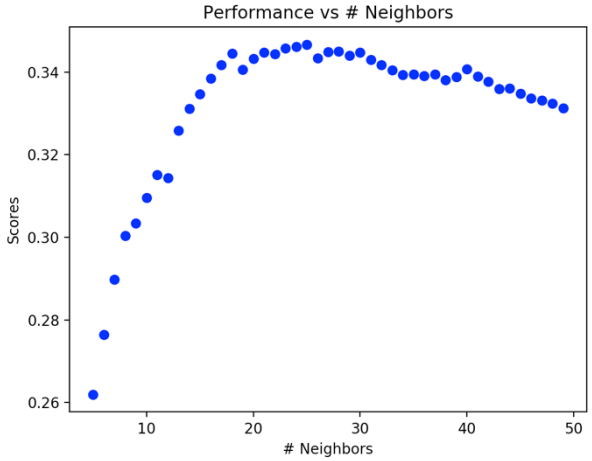
Tuning hyperparameters

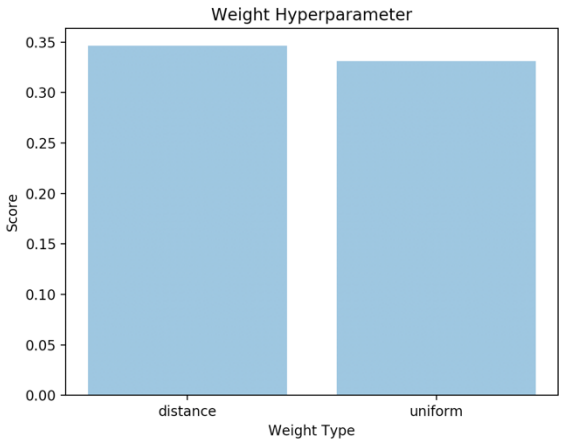
A) K Neighbors Regression

The most significant hyperparameter that I needed to evaluate for the K Neighbors Regression algorithm was the number of neighbors I would choose. I used GridSearchCV to tune this hyperparameter. This methodology helped me determine an ideal set of hyperparameters for the training data. GridSearchCV found the optimal number of neighbors to be 24, the ideal p value to be 1, and the weights parameter as distance. The output for the weights parameter indicates that it was beneficial to weigh neighbors whose proximity was closer to the query point more highly. As for the p parameter, the value of 1 indicates to use the Manhattan distance formula. This distance formula was most likely chosen because it performs well with higher dimensional datasets. I imagine the 10 features I was considering played a role in GridSearchCV coming to this conclusion. Finally, the number of neighbors chosen for the model was 24 which

In order to also evaluate and visualize the results of the GridSearchCV best params, I also use the "cross_val_score" library which allowed me to isolate different hyperparameters and graph their relative performance given a range of inputs. For example, the "Performance vs. # Neighbors" graph depicts the cross-validation score when I fit the model with 10 different splits at different neighbor counts between 5 and 50.

Beside the figure, the “Weight Hyperparameter” illustrates that the performance differential between uniform and varied weights assigned to neighbors adjacent to the query point.





The general trend for the # Neighbors graph shows that the best performance comes when the number of neighbors is between 20 and 30. As for the weight hyperparameter, the distance parameter performing better can be attributed to the variance in the data. As the data becomes more and more spread out, the likelihood that an outlier negatively influencing the accuracy of our prediction increases. Thus, by weighing the points closer to the query point higher, we can minimize the negative affect of these outliers.

Using these best parameters in my prediction yielded a result of 0.346

The execution time of the Kernel Neighbors Regression was: 5.869 seconds

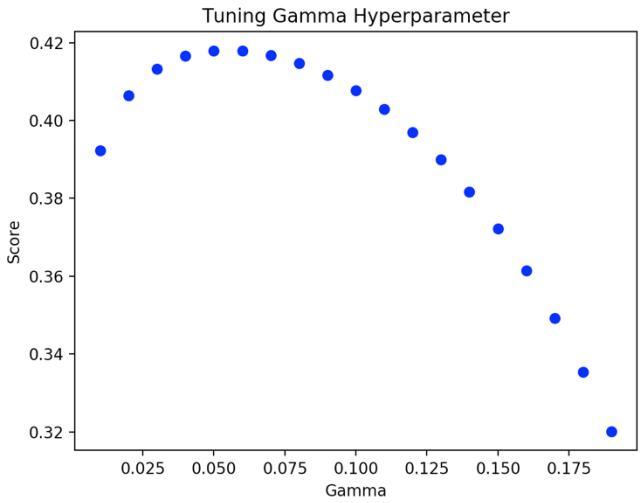
B) Kernel Ridge Regression

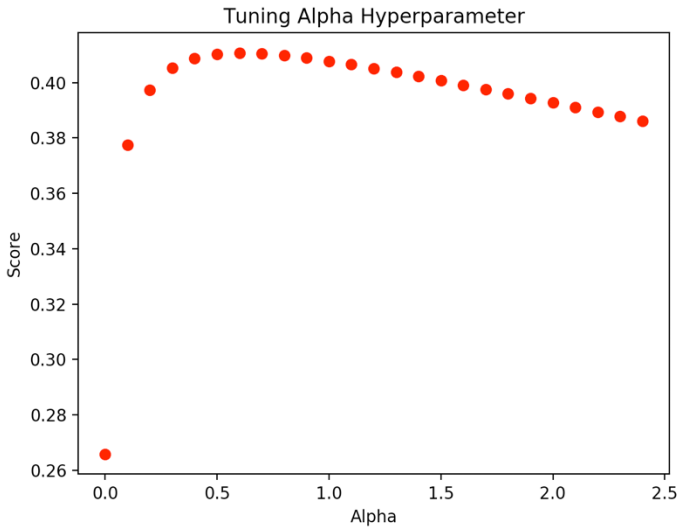
The significant hyperparameters for the Kernel Ridge Regression algorithm included gamma, alpha, and the kernel type. My methodology involved first tuning utilizing the GridSearchCV library to find the ideal set of these variables. Once I acquired this data, I performed my own isolated tests on the features to visualize and evaluate the conclusion that was drawn by GridSearchCV.

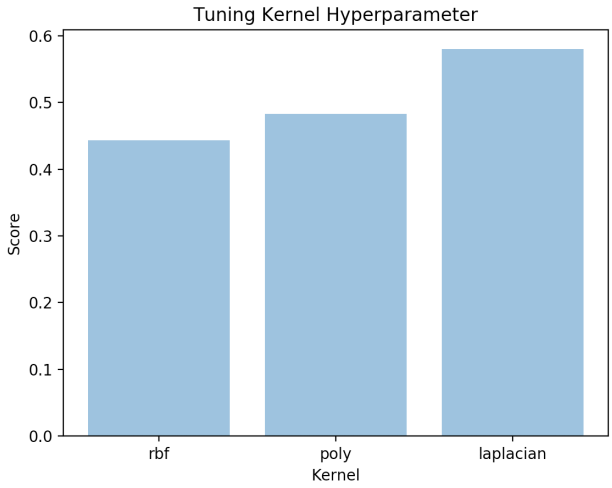
Tuning the non-discrete hyper parameters functioned similar to the way that I found the best parameters in the K Neighbors algorithm. The figures illustrate evaluating these

different parameters in isolation from one another. They are useful in that they employ a variable number of cross validations and splits on the data set using the “cross_val_score” library. The results of this search indicated that the best kernel was Laplacian, the alpha value was 0.7 and the gamma value was 0.04. As for the best kernel, both my isolated tests as well as the best params outputted by the GridSearchCV class determined that Laplacian was the optimized kernel choice.

Using these best parameters in my prediction yielded a result of 0.419. The execution time of the Kernel Ridge Regression was: 768.61 seconds.

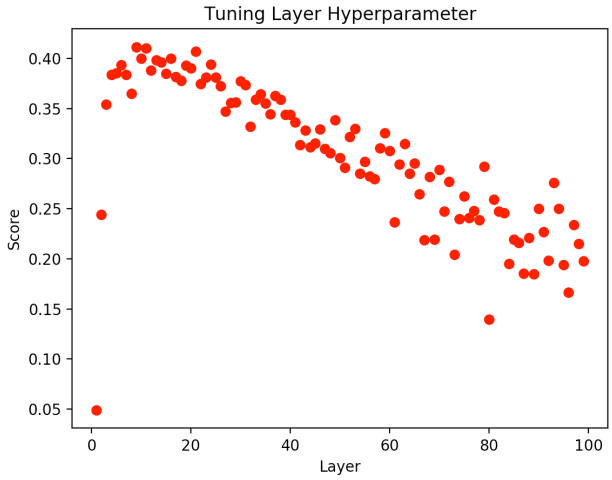


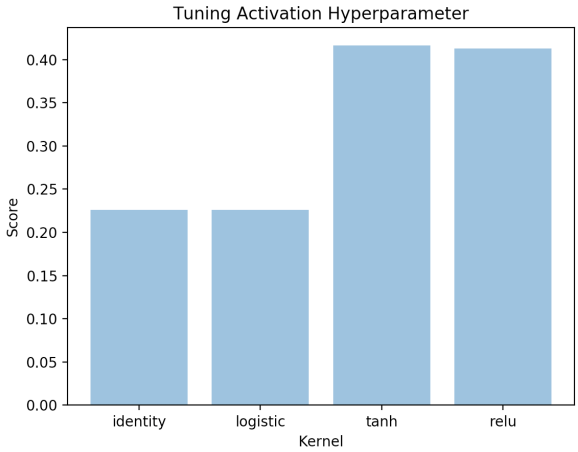


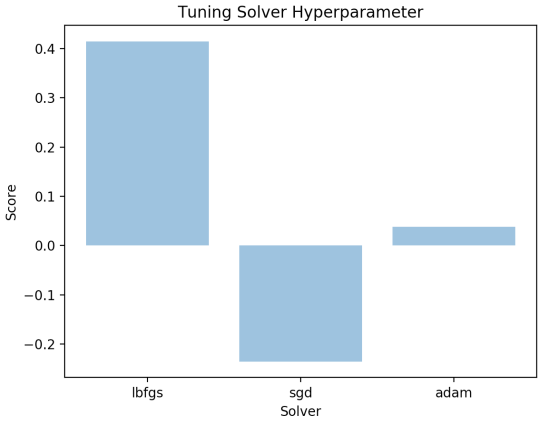


C) Neural Network

The significant hyperparameters that I factored into initializing my neural network included the activation function, the solver, and the hidden layers. The GridSearchCV produced the output which that preferred the “relu” activation function, the size of the hidden layer 2 layers of 10 neurons each, and the “lbfgs” solver.







Similar to the methodology used in the previous two algorithms, I employed the help of the GridSearchCV library, the MLPRegressor class, and isolating the hyperparameters to visualize their performance irrespective of the other parameters. Before evaluating each individual hyperparameter with its relative values, I used the GridSearchCV library to capture the ideal set of hyperparameters for my training dataset. The output of the best params after fitting my data proved that the number of hidden layers would be 2 layers with 10 neurons each, the best activation function would be relu, and the ideal solver was lbfgs.

The outputted number of hidden layers could likely be attributed to the high number of features that exist in the data. The Lbfgs is a logical solver for the dataset because lbfgs can handle both dense and sparse inputs. Relu, the activation function chosen by the GridSearchCV class, most likely was determined because the linearity of the slope of this activation function stays constant, so it does not saturate when x increases.

The performance of the Neural Network given the acquired best parameters was: Score: 0.396

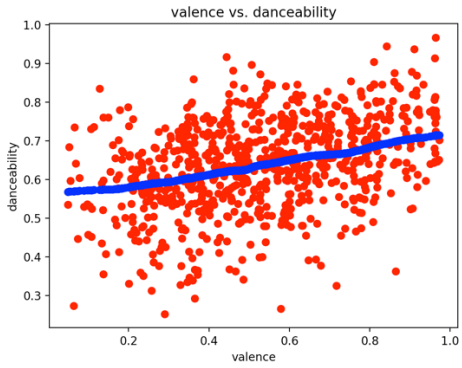
The execution time of the Neural Network was 255.80 seconds.

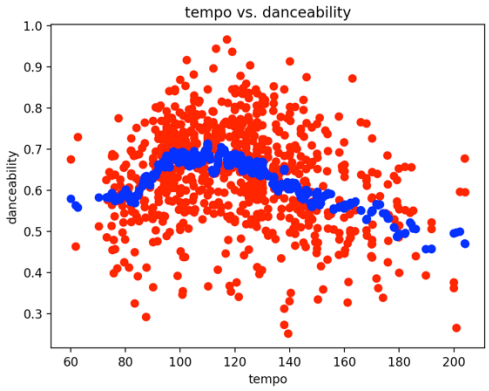
Comparing Algorithm Performance

Purely evaluating score, the most accurate algorithms were Kernel Ridge Regression (41.9%), Neural Network (39.6%), and K Neighbors Regressor (34.6%) respectively.

The tests that I performed to measure the included using the cross_val_score. I used the mean and the standard deviation of the cv scores as metrics to evaluate the efficacy and relative accuracy of each of the algorithms. The K neighbors algorithm had 26% accuracy given by cross_val_score when the cv param was 5 which is lower than the accuracy given by the built score method. This indicates that the algorithms were not as accurate when it came to generalize and predicting with other test sets. This was the case for the neural network as well the kernel ridge regressor (by a similar margin). The highest relative accuracy of the kernel ridge was likely attributed to wide range of values I fed into the GridSearchCV class wrapper. This would ultimately help better tune the hyperparameters This tuning performed well in comparison to the Neural network which would have drastically slowed if more potentially hidden layer sizes were fed into the GridSearch. As for the K Neighbors algorithm, the lower performance was most likely due to the variance within each feature. When I plotted individual features against the danceability metric I saw a large distribution of points. This would inevitable cause the K-Neighbors algorithm which is most effective when there are clusters within the features. Without these clusters, choosing the neighbors is fairly arbitrary and the core functionality of the algorithm is lost over the dataset. A reason why this algorithm performed significantly better than just using linear regression is likely due to the relationship between the features. So, while the 2d graphs I generated to compare the individual features to the danceability output did not depict many patterns. There may have been sets of features that combined formed patterns such as the energy feature combined with the loudness feature.

One important factor to consider when evaluating the music data set is within each individual feature there is a large amount of variance. This variance makes it difficult for the regressor to make an accurate prediction. Thus, in the effort to consider only the most important features I ran the Kernel Ridge Regressor on individual features to evaluate the accuracy of predicting the hypothesis class without noise from other features.





The visuals generated from these tests gave me some insight into some of the patterns that were naturally occurring in the data set. For example, the for the valence feature, as a general trend, as valence increase so does the danceability of a song. The temp vs danceability relationship saw more of a bell curve relationship. The maxima of this pattern was around one 110 and tapered off on both sides meaning the more a song's tempo varied from this peak, the more likely it would be for the song to be flagged as less danceable. The tests also gave me insight into what features would be important to remove such as the duration_ms column of the data frame.

If I had more time, I would map the features that have discrete values. For features such as the song's mode (whether or not the song was major or minor), I could have assigned a Boolean int 0 or 1 to indicate the value and use this as a potentially valuable feature. As for features with multiple label values, I could have created columns that represented flags of whether or not the data point fell into that category. For example, for genre, columns like "isRap", "isCountry", "isDance" potentially provide more useful metrics.

Conclusion

If I were to perform this experiment in the real world and choose the optimal algorithm, I would choose the Neural Network. While the Kernel Ridge Regressor had the highest accuracy, running the algorithm took more than 12 minutes in comparison to the neural network which only took ~240 seconds. This speed can most likely be attributed to the use of hidden layers which aids the network in choosing only the most pertinent information from the inputs and removing redundant data. Admittedly one component of the neural network that might see a drastic increase in time complexity comes when we use GridSearchCV to determine the number of hidden layers. While I optimal size of hidden layers out of a set of 10 values, when the complexity and size of the data set increases, so with the exhaustiveness of the GridSearch to find the ideal number of hidden layers. As the Additionally, using neural network provides a variety of benefits such as the use of the activation function which can help determine non-linear relationships between different features. While the has some benefits in regard to training, these advantages are ultimately outweighed by the potential of over fitting or underfitting the data. The existence of this

problem generates a sensitive balance when one must train the Kernel Ridge regressor to the training set.

The music dataset proved to be difficult in terms of making accurate danceability predictions based on the training data. At the same time however, each model significantly outperformed the standard linear regression score which indicates that tuning hyperparameters such as hidden_layer_size, alpha values, and neighbors count while increasing the complexity of the algorithm undoubtedly increase the efficacy. Ultimately, this exercise demonstrated the importance of first determining the relevant features to consider, scaling the data appropriately, and finally tuning hyperparameters to make a more accurate and generalized prediction.

Works Cited

MLP Regressor

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html
https://www.programcreek.com/python/example/93778/sklearn.neural_network.MLPRegressor

Kernel Ridge
https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge.html
https://www.programcreek.com/python/example/91966/sklearn.kernel_ridge.KernelRidge

K Neighbors Regressor
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

Cross Validation
https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation

GridSearchCV
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html