Kent Yamamoto
CS 4641

Final Project Report

Introduction

The final project for this class was to find a non-trivial dataset that was 1. interesting and important to me and 2. usable from a machine-learning perspective, then choose either regression or classification techniques and analyze the results. I found a dataset of 2016 songs that someone retrieved from Spotify's API. Each song has 13 features, or "attributes": acoustics, danceability, duration (ms), energy, instrumentalness, key, liveness, loudness, mode, speechiness, tempo, time signature, and valence. These were all given a numerical score for each song, as well as a binary classification for each song, "1" if the user liked the song and "0" if the user did not like the song.

I decided to use this dataset because as a die-hard music lover, I've always been interested in how Spotify and SoundCloud recommend songs and create their "Discover Weekly" playlists. Being able to see the backend of Spotify and then personally going through three different classification methods with this Spotify dataset was something I really wanted to experience. By doing this project with this dataset, I can potentially do the same thing with a dataset of my own songs that I like/don't like and see why I like the songs that I do and vice versa, as well as determine whether I might like a song before even listening to it.

To prove that the dataset is nontrivial, I followed the guidelines provided in the project description pdf: my dataset consists of 2016 datapoint (v.s. > 1000), each datapoint has 13 different features, and the data set has a non-trivial distribution, i.e. it is not linearly separable. This was established by training a SVM with a linear kernel, and evaluating the model with both the training and testing dataset:

Training Data Confusion Matrix

360	132
190	326

Accuracy: 0.68

Testing Data Confusion Matrix

374	131
222	282

Accuracy: 0.65

We can say that these scores are low enough to make an assumption that the dataset is not linearly separable.

Supervised learning can be split into two groups, classification and regression. The objective with supervised learning is to solve for the simplest yet most accurate (bias-variance tradeoff) hypothesis class, or function, given a set of data and its classes. The dataset needs to be split into two: a training and testing dataset, where the training will be used to construct the function that should be able to correctly classify the testing dataset. We need to then implement cross validation to compare the performance of the three different algorithms (Random Forests with Bagging, SVMs with a non-linear kernel, and NN with at least two hidden layers), to estimate the parameters for each method and test the algorithm. Once the testing data is fed into the three tuned algorithms, we will obtain an error value or score in which we can compare and determine the best classification algorithm for this given dataset.

Description of the Algorithms

Random Forests with Bagging

This machine learning algorithm creates an X **number of decision trees** and implements Y **features at each split for each tree** to create a wide variety of different trees (the X and Y variables will be our hyperparameters for this classification method). Bagging refers to fitting the training dataset to each tree created, and return the class with the most occurrences from all the tree.

The hyperparameters for Random Forests contribute to the complexity of the hypothesis class in two ways. First, increasing either the number of trees and/or the features implemented in the formation of each tree will increase the time it takes for training and will make the hypothesis class too complex, which may result in overfitting when fitting the testing dataset. Additionally, having too few trees and implementing few features in each tree will not provide the hypothesis class enough specificity, which can result in underfitting, or a poor accuracy score when fitting the testing data set.

Support Vector Machines with Nonlinear Kernel

We were introduced to SVM's in HW2, but never worked with them from a coding perspective. A linear SVM classier draws a straight, linear line between two classes, which will separate the two and classify them. However, we are working with an SVM with a non-linear kernel, which maps the dataset outside of a linear dimension space (a higher dimension) so that we can find a plane in the higher dimension that can separate the samples. Examples of a non-linear kernel is the radial basis function kernel, RBF Kernel, and the polynomial kernel.

The hyperparameters we will be tuning are **c, gamma, and the type of kernel**. C determines the tradeoff between accuracy in the classification training and straight/smooth decision boundaries. A greater value of C will result in a more complex decision function but with greater accuracy, and a smaller value of C will result in a simpler and straight decision boundary while accuracy is decreased. Gamma determines how far the influence of a training point makes on the decision boundary. A greater value of gamma results in the points closer to the decision boundary having a greater influence resulting in a more jagged decision boundary to account for the data points that seem to “bleed over”. A lower value of gamma results in the points farther from the decision boundary to have a greater influence, making the decision boundary less jagged and more linear. A more complex, jagged decision boundary will result in a more complex hypothesis class, but we also want to maintain a certain level of accuracy as well without oversimplifying the hypothesis class.

Neural Network with at least 2 Hidden Layers

Neural network classification consists of perceptrons (nodes) and edges in multiple layers that are interconnected. There’s an input layer, an output layer, and all layers in between are called hidden layers. The learning process starts from the input layer, where the features are fed through the first hidden layer and are assigned weights. At each node an activation function will introduce nonlinearity and the “activation” is outputted to the next hidden layer. This process is called forward propagation. Once it reaches the output layer, it will observe whether the classifier predicted the correct class. If not, then it will modify the weights that are assigned and go through the entire forward propagation process again until the predictions converge to the proper class.

The hyperparameters we want to tune are: the type of activation function (e.g, ‘identity’, ‘logistic’, ‘tanh’, ‘relu’), the number of hidden layers, the number of perceptrons in each layer, and the learning rate. An inadequate activation function will result in unnecessarily more iterations of the forward propagation process, which will increase the complexity of the model. The number of hidden layers and the perceptrons in each layer are related, in which having few perceptrons with multiple hidden layers might converge faster, or the same can be said with multiple layers but few perceptrons in each layer. These are the types of tests I would have to run to determine the most optimal combination for my dataset. Additionally, I would like to see whether the default learning rate of 0.001 can be further optimized by increasing it to 0.005 or 0.01 without losing accuracy and/or precision.

Tuning Hyperparameters

For each GridSearch iteration, I decided to set cv = 5, because I ran cv = 10 and took too long to run, and I believed that since I was looking at the average score anyways, a cv of 5 would allow me to still get enough folds to look at the dataset in different groupings while also saving time for each test.

Random Forests with Bagging

Test 1 - Exhaustive GridSearchCV (testing all max_features parameters)

My first test just to see a ballpark of where I should start consisted of 325 different tests (max_features: np.arange(1,14,1), n_estimators: np.arange(5,500,20)).

I chose these parameters for two reasons: 1. Since I don't have too many features I thought just narrowing the optimal number of features first would help me narrow down the number of estimators later. 2. I didn't want to create more than 500 trees for the Random Forest, as it would just result in an unnecessarily complex model. Below are the ten combinations that yielded the highest test score:

Rank	max_features	n_estimators	mean_test_score	std_test_score
1	**3**	**365**	**0.303785178**	**0.03588653**
2	3	485	0.303633506	0.037913229
3	3	425	0.303553589	0.036620163
4	3	405	0.303537842	0.036591174
5	3	465	0.3035378	0.036631864
6	3	385	0.303454044	0.036560376
7	3	345	0.303427725	0.036219192
8	3	445	0.303299724	0.036984262
9	3	325	0.302723666	0.03690457
10	3	305	0.30253503	0.037830952

Time it took to complete test: 441.34 sec (~7.4 min) (omitted: time elapsed for each run, "Bootstrapping?" Column because I stated that we are conducting RandomForests with Bagging, test scores of each fold, because the average test score is enough to determine the proper hyperparameters). We can see that the max_features parameter is optimal at 3 based off of the top 10 results, so we can keep it at 3 for the next round of tests. Now we can do a second test and narrow down the optimal n_estimators value, which floats around the 350 and 480 range, so now we can design our second experiment given these results.

<u>Test 2 - Narrowing Down n_estimators Parameter</u>

Running another GridSearchCV with only 1 option for max_features, we set the n_estimator values from 350 - 490 in steps of 5, inclusive (Based off test 1 results):

Rank	max_features	n_estimators	mean_test_score	std_test_score
1	3	435	0.370411609	0.02525208
2	3	475	0.370364755	0.025344024
3	3	430	0.370321747	0.025010145

Rank	max_features	n_estimators	mean_test_score	std_test_score
4	3	415	0.370264239	0.025089895
5	3	480	0.370198255	0.025046843
6	3	470	0.370185825	0.025355137
7	3	465	0.370139772	0.025471886
8	3	460	0.370082323	0.025432747
9	3	425	0.370081579	0.025214928
10	3	405	0.370022234	0.025096828

Time it took to complete test: 177.61 sec (~2.96 min) (omitted the same columns as the first table). From the top 10 results, we can see that the n_estimators parameter fluctuates, but the scores are only changing by +/- 0.00005, thus we can say that an approximate range of [415 - 480] will not make a significant difference in our final tuned model. Just to be safe, I have used 435 as the "most ideal" parameter and trained the model:

Tuned Model and Results

I created a RandomForests model with the optimal hyperparameters (max_features = 3, n_estimators = 435) and evaluated the testing dataset. Below was the result:

Confusion Matrix

370	100
133	406

Accuracy: 0.77

Afterwards, I conducted a (k = 10) cross_validation with the testing dataset and obtained an **average score of 0.790,** with a confidence interval of **+/- 0.0270,** resulting in a range of **[0.763, 0.817]** for the RandomForest Classification method.

SVM with RBF Kernel

After testing for 'rbf', 'poly', and 'sigmoidal' in the GridSearch, I found that it was taking more than 7 hours. With that said, I ran them individually and found that the RBF kernel took less than 10 seconds every time while the other two individually didn't converge even after running for 5 hours. Therefore, I modified the hyperparameters that I will be tuning to just **C** and **Gamma** and kept the type of kernel constant at RBF.

Test 1 - Determining Optimal C and Gamma Values

Based off previous experiments and examples online, I decided to set my initial hyperparameter values to the following: C: (0.01, 0.1, 1.0, 10, 100, 1000), gamma: (1000, 100, 10, 1, 0.1, 0.01, 0.001). Below are the results for the top 10 highest scoring results:

Rank	C	Gamma	mean_test_score	std_test_score
1	**1000**	**0.1**	**0.740091621**	**0.023280443**
2	100	0.1	0.734126398	0.017089254
3	10	1	0.732160977	0.026473645
4	100	1	0.724240185	0.029935701
5	1	1	0.719245357	0.011119845
6	10	0.1	0.714270233	0.017295159
6	1000	0.01	0.714270233	0.015501766
8	1000	1	0.6994286	0.030580893
9	1	10	0.696443525	0.025244069
10	10	10	0.689498054	0.026992234

Time it took to complete test: 7.09 sec (omitted: time elapsed for each run, “kernel” Column because I did not set the kernel type as a tuning hyper parameter and set it constant as tuning strictly “RBF” kernels, test scores of each fold, because the average test score is enough to determine the proper hyperparameters). We see that the first-ranking score and the second ranking score have a significant difference (compared to when running GridSearch for RandomForests), so I will say that the set of parameters that gave me the highest score in this iteration of tests will suffice. Therefore, our tuned parameters are: C = 1000 and gamma = 0.1.

Tuned Model and Results

I trained the classifier with the optimal parameters and training set, and ran the testing dataset through the model and obtained the following:

Confusion Matrix

390	121
137	361

Accuracy: 0.77

I ran a k=10 cross validation with the testing dataset and obtained an average score of **0.737,** with a confidence interval of **+/- 0.0275**, resulting in a range between **[0.710, 0.765]** for the SVM with RBF-Kernel Classification method**.**

Neural Network with 2+ Hidden Layers

Test 1 - Determining Optimal Activation Function and Hidden Layer Size

To simplify the GridSearchCV process, I decided to first determine the optimal number of hidden layers by keeping a constant value for the size of each hidden layer (set equal to number of features as recommended) while modifying the number of hidden layers. Just to make sure that the score does not improve based on a certain combination of the number of hidden layers and the size of each layer, I added the same set of parameters for when the size of each layer is +/-1 (three different sizes per layer, 4 different number of hidden layers). Additionally, I decided to incorporate the activation function parameter into this test as well, as there are only 4 possibilities:

Rank	Activation Function	# Hidden Layers, Size	Learning Rate	Max Iterations	mean_test_ score	std_test_ score
1	relu	4, 13	0.001	1000	0.707334614	0.01921158
2	relu	2,14	0.001	1000	0.69940397	0.019456542
3	relu	2, 13	0.001	1000	0.697428698	0.026297564
4	tanh	5, 13	0.001	1000	0.696423821	0.024292931
5	relu	3, 13	0.001	1000	0.696418896	0.018837917
6	relu	4, 14	0.001	1000	0.695468204	0.020697035
7	relu	5, 14	0.001	1000	0.695463278	0.026189056
8	relu	3, 12	0.001	1000	0.693443673	0.012529948
9	relu	5, 12	0.001	1000	0.691488104	0.031333154
10	tanh	4, 12	0.001	1000	0.688503029	0.013547721

Time it took to complete test: 52.41 sec (omitted: time elapsed for each run, test scores of each fold, because the average test score is enough to determine the proper hyperparameters). From the results above, we can see that the "relu" activation function is the most optimal. Additionally, we see that 12 as the size of the hidden layer is on the lower end of scores, which proves that we should aim for sizes that are either equal to or greater than the number of features. Finally, we can see that the number of hidden layers varies, but having fewer hidden layers (e.g. 2) provides us with scores that aren't too far off from those of 4 and 5. We will keep this in mind when running the next set of tests for further optimizing the number of hidden layers, size of each layer, and the learning rate.

<u>Test 2 - Determining # of Hidden Layers, Size/Layer, and Learning Rate</u>

For this test, I kept my activation function constant as 'relu', then added different parameters to the learning rate: 0.001, 0.005, 0.01. I started with 0.001 as it was the default learning rate, then increased as shown above but did not exceed 0.01, as I thought it would result in underfitting. Additionally, I keep the same parameters for the # of Hidden Layers and Size/Layer, except I removed the parameters with hidden layers of size 12 and replaced them with 15, as I wanted to see how the scores would change with hidden layer size greater than the number of features. Below are the top 10 results:

Rank	Activation Function	# Hidden Layers, Size	Learning Rate	Max Iterations	mean_test_ score	std_test_ score
1	relu	2, 15	0.01	1000	0.700448254	0.047879176
2	relu	2, 13	0.005	1000	0.700384218	0.037747549
3	relu	2, 13	0.01	1000	0.698408945	0.023995607
4	relu	2, 14	0.01	1000	0.695463278	0.035282162
5	relu	2, 15	0.005	1000	0.692492981	0.037253862
6	relu	4, 13	0.005	1000	0.692463425	0.033600939
7	relu	2, 14	0.005	1000	0.6924585	0.033887845
8	relu	3, 15	0.01	1000	0.691478252	0.027425368
9	relu	2, 14	0.001	1000	0.690488153	0.023408103
10	relu	3, 14	0.001	1000	0.688517807	0.020004343

Time it took to complete test: 36.13 sec (omitted: time elapsed for each run, test scores of each fold, because the average test score is enough to determine the proper hyperparameters). From the results above, we can see that clearly the fewer hidden layers there are, the greater the score is. Additionally, 0.001 learning rate may be too slow, as 0.005 and 0.01 result in higher scores. One interesting thing to point, however, is that the third best set of parameters includes a learning rate of 0.01 with the lowest standard deviation of scores, and the average score does not change significantly. I believe a final test narrowing down the size of each layer and learning rate is required.

Test 3 - Finalizing Size/Layer, and Learning Rate

I narrowed down the number of layers to 2 based off the previous test results, added more parameters to the size per layer, drastically increasing the size to see what the relationship is if the size is significantly greater than the number of features. Below are the top 10 results:

Rank	Activation Function	# Hidden Layers, Size	Learning Rate	Max Iterations	mean_test_ score	std_test_ score
1	relu	(100, 100)	0.005	1000	0.709339441	0.026951777
2	relu	(50, 50)	0.005	1000	0.706349441	0.027718208
3	relu	(16, 16)	0.001	1000	0.705379045	0.028403425
4	relu	(14, 14)	0.001	1000	0.695497759	0.038620514
5	relu	(20, 20)	0.001	1000	0.695468204	0.041338996
6	relu	(100, 100)	0.01	1000	0.69348308	0.046294903
7	relu	(50, 50)	0.001	1000	0.693433821	0.056344655
8	relu	(100, 100)	0.001	1000	0.693394414	0.045341505
9	relu	(16, 16)	0.005	1000	0.691483178	0.037374542
10	relu	(50, 50)	0.01	1000	0.690443821	0.026693519

Time it took to complete test: 30.36 sec (omitted: time elapsed for each run, test scores of each fold, because the average test score is enough to determine the proper hyperparameters). From the table above, we can see that a size of 100 per layer was actually the highest scoring parameter with a learning rate of 0.005. Although this might make the model more complex because the size in each layer is greater, the standard deviation in the score is significantly less than the others while having the highest average. Therefore, I am going to choose the first ranking set of hyperparameters as my tuned parameters: **Activation Function: Rectified Linear Unit, # of hidden layers - 2, size of each layer - 100, and learning rate - 0.005**

Tuned Model and Results

I trained the classifier with the optimal parameters and training set, and ran the testing dataset through the model and obtained the following:

Confusion Matrix

381	117
146	365

Accuracy: 0.74

I ran a k=10 cross validation with the testing dataset and obtained an average score of **0.750,** with a confidence interval of **+/- 0.0350**, resulting in a range between **[0.715, 0.785]** for the 2-layer Neural Net Classification Algorithm**.**

Comparing Algorithm Performance

Below is a table showing the cumulative results across the three different algorithms I tuned and evaluated:

Algorithm	Test Run Time (in order) (sec)	Number of Hyperparameters	Mean Score (CV, K = 10)	Confidence Interval	Score Range
RandomForests	441.34, 177.61	2	0.790	+/- 0.0270	[0.763, 0.817]
SVM (RBF Kernel)	7.09	3	0.737	+/- 0.0275	[0.710, 0.765]
NN (2 Layers)	52.41, 36.13, 30.36	4/5*	0.750	+/- 0.0350	[0.715, 0.785]

We can observe that the the highest average score obtained was 0.790 from the RandomForests method with the smallest confidence interval, demonstrating accuracy and precision (especially with only 2 hyperparameters). However, it took significantly more time than the other algorithms. On the contrary, SVM training time was the fastest by a significant amount with only an extra hyperparameter compared to RandomForests and the confidence interval did not differ much. However, it was the lowest-scoring algorithm. Neural Nets took longer than SVM but not as long as RandomForests with more hyperparameters, and the score was also in the middle. However, it had a notable increase in confidence interval, which implies that NN may not be the most precise. I've created another table below summarizes the results in a more simplified fashion:

Fastest Computation Time	SVM
Most Accurate	RandomForests
Most Precise	RandomForests/SVM

I would like to conclude that if training time is not an issue, then RandomForests would be the most ideal algorithm to use. However, if computation time is an important factor in the application, then SVM will result in a significantly faster training time but some accuracy will be sacrificed while still maintaining relatively the same confidence interval as RandomForests (difference of 0.0005). I would not recommend NN for my application, as it took decently long and didn't show either a high average score or low confidence interval.

Conclusion

In this project, I took a dataset of songs with 13 features and a classification of whether the listener liked them or not. This can be done for anyone's favorite/least favorite songs to create a classifier that can predict whether someone would like a song or not, which I believe is an important factor in music-streaming platforms' "Recommended" sections. I took this dataset and tuned hyperparameters for three different classification algorithms (RandomForests, SVMs, and NNs) and compared them to determine which algorithm was the most optimal for my dataset after tuning the hyperparameters.

For my specific application, I believe that the RandomForests classification algorithm is the best. Although it did take the longest to train, it had the highest average score, and I would value accuracy over training time for determining whether I would like a song or not. If I was given a list of 10 songs that were determined songs that I may like, I would prefer to have 8 songs that I actually liked instead of 7 (RandomForests v.s. SVM, respectively).

Acknowledgements:

General resources I used:

- Piazza
- Udacity videos on related topics
- Professor Hrolenok's slides
- Peers to **discuss** high-level concepts (***no*** code was copied)

Dataset:
https://www.kaggle.com/geomack/spotifyclassification

RandomForests:
https://chrisalbon.com/machine_learning/trees_and_forests/random_forest_classifier_example/
https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

SVM:
https://towardsdatascience.com/understanding-support-vector-machine-part-2-kernel-trick-mercers-theorem-e1e6848c6c4d
https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74
https://www.youtube.com/watch?v=Toet3EiSFcM (Statquest)
https://www.youtube.com/watch?v=m2a2K4lprQw (Hyperparameters)
https://www.geeksforgeeks.org/svm-hyperparameter-tuning-using-gridsearchcv-ml/

NN:
https://www.youtube.com/watch?v=P2HPcj8lRJE
https://www.kaggle.com/hatone/mlpclassifier-with-gridsearchcv
https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html?fbclid=IwAR2JEP023hfLdESXQupml9nRpo4iSYg357hO4t2rnEqmDnNz-jtBywUjuQw

Confidence Intervals:
https://www.mathsisfun.com/data/confidence-interval.html