Classifying Poker Hands: Not a Gambler's Task

Julie Lenzer
MSML 651 – Big Data Analytics
College Park, Maryland
jlenzer@umd.edu

***Abstract*—For this project, I attempted to predict the type of poker hand represented by a random draw of five cards out of a standard 52-card deck using a variety of classifiers. The "best" model was then used to predict the hands of 1,000,000 draws in the test dataset. Given the large number of draws in the test set, Spark MLlib was used to train the selected model and predict the hands of the test data.**

I. Introduction

For the final project, I used the Poker Hand dataset [1]. The dataset comprises 10 features and an associated classification of the type of poker hand. The objective of the project was to predict the type of hand the five cards drawn from a standard deck of 52 cards represent. The training set provided included 25,010 possible hands while the test set comprises 1,000,000 draws of five cards. Specific features used to train and predict the hand include:

Suit, represented as: 1 – Hearts, 2 – Spades, 3: Diamonds, 4: Clubs

Rank, represented as A – 1, 2 – 10 for non-face cards, 11 – Jack, 12 – Queen and 13: King

Altogether, these yield ten total features making this a multivariate classification problem. Both the training and test data are labelled with one of 10 possible poker hands, as follows, which also makes this a supervised learning multi-class classification problem.

0. Nothing in the hand
1. One pair
2. Two pair
3. Three of a kind (single rank, all different suits)
4. Straight (cards of two or more suits in a sequence of five)
5. Flush (all cards of the same suit)
6. Full house (three of one rank, two of another rank)
7. Four of a kind (all four cards of a single rank)
8. Straight Flush (cards of same suit in sequence of five)
9. Royal Flush (10-J-Q-K-A of the same suit)

II. Overview of the Data

The data for this project is particularly difficult for a classification problem for two reasons. First, the distribution of classes, as shown in Figure 1 is extremely imbalanced. Indeed, the first two classes (*Nothing in the hand* and *One pair*) represent 92% of the classes in the training data. The testing data follows a similar imbalance.

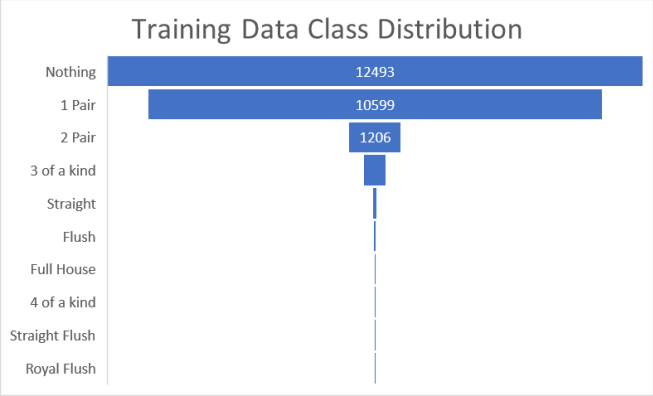

Figure 1: Class Distribution of Training Data

The approach for generating this data was intentional so as to represent the actual probability distribution of the various possible hands from a five card draw out of a standard set of 52 cards. Only the last two classes – *Straight flush* and *Royal flush* – are oversampled in the training data. Similarly, only the *Royal flush* was overrepresented in the testing data. This was due to their very small probability of being drawn in a real hand. However, following that model exactly would not have produced enough data to train a model to ever detect those hands. As it is, there are 311,875,200 possible hands so training with only 25,010 seems insufficient.

The second challenging aspect of this data set lies in the way the cards are represented. As shown in Figure 2, the data is structured as if sequence of the draw matters. Each of the hands depicted contains the same cards in a different position, yet in this dataset, each represent one distinct instance of class 1 – *One Pair*.

Figure 2: Poker Hand Representation

In a real game of poker, the order in which the cards are drawn is not significant for determining the hand (*class*). This introduces an additional and unnecessary complexity to the classification problem. If, however, we were looking to predict the probability of getting a certain hand based on the first *n* cards drawn, this dataset might be more helpful. One possibility for future exploration of this dataset would be to investigate transformation of the data into a structure that does not consider sequence.

III. Model Selection

Using the training data provided, I opted to run a test of several difference machine learning classifiers in order to determine which one worked the *best* on this dataset. The models tested included K-Nearest Neighbors, Support Vector Machine, Decision Tree, and Random Forest. As part of the evaluation, I did a 70/30 split on the training set with 70% of the data being used for training and the remaining 30% for validation.

The criteria for determining which model was *best* included looking at a map between the actual and predicted values (noting number of classes predicted), accuracy, and F1 score. I also performed cross validation on each to determine if the train/test split had a significant impact on the results. Each model tested, along with its results, are described briefly in the following sections.

A. K-Nearest Neighbor

K-nearest neighbor (k-NN) assigns a class to a hand based on the classes of the hands of its *k* nearest neighbors. This method assumes that points in close proximity to each other are part of the same class. In addition, choosing different values for *k* could change the results so I ran the training algorithm with values of 2-8 for *k* in fine tuning this model and determined that 6 yielded both the highest accuracy and F1 score as shown in Figure 3. However, it is generally better to use an odd *k* so that there is some way to break a tie, so 5 was the next best value so that was the one that was used in model comparison.

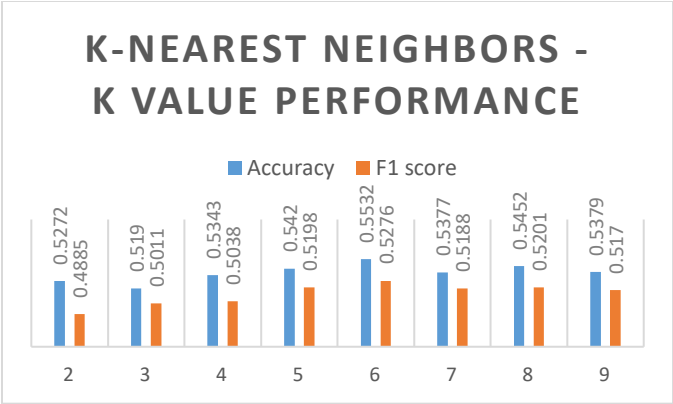

Figure 3: K-value performance comparison

The k-NN model results with *k*=5 are shown in Table 1. Given the cross validation standard deviation shown below, the train/test split appears to have had an insignificant impact in the performance. It was initially surprising to not see all of the classes predicted, which I will address in section V *Conclusions* on page 5. In this case, only four were predicted.

Table 1: k-NN Performance Metrics

K-Nearest Neighbors, k=5	
Accuracy	0.53285
F1 Score	0.51159
CV Std Dev	6.46e-3
Number of Classes Predicted	4

B. Support Vector Machine

The Support Vector Machine (SVM) algorithm seeks to separate the various classes of the data through hyperplanes and can be used for both regression and classification. Like K-NN, data where proximity of data maps to class similarities lends itself well to SVM.

The SVM model results are shown in Table 2. Given the cross validation standard deviation shown below, the train/test split appears to have had an insignificant impact in the performance. Only class *0 – Nothing in the hand*, was predicted.

Table 2: SVM Performance Metrics

SVM	
Accuracy	0.50246
F1 Score	0.66885
CV Std Dev	3.46E-05
Number of Classes Predicted	1

C. Decision Tree

Decision trees used in machine learning classification can be simply described as an if-then-else tree structure. It can also be thought of as a potentially complex tree-structured flow chart.

The Decision Tree model results are shown Table 3 . Given the cross validation standard deviation shown below, the train/test split appears to have had a larger impact in the performance than the other models, but still not significant enough to change the conclusions of this experiment. This is the only model to have predicted all nine classes.

Table 3: Decision Tree Performance Metrics

Decision Tree	
Accuracy	0.45715
F1 Score	0.46
CV Std Dev	1.47E-02
Number of Classes Predicted	9

D. Random Forest

Random Forest combines the outputs of several *decision trees* to create a model that can predict the dependent variable based on the values of the independent variables. A random forest model returns the mode of the classes from all the trees as the predicted value [2].

The Random Forest model results are shown in Table 4. Given the cross validation standard deviation shown below, the train/test split appears to have had an insignificant impact in the performance.

Table 4: Random Forest Performance Metrics

Random Forest	
Accuracy	0.5528
F1 Score	0.528
CV Std Dev	8.40E-03
Number of Classes Predicted	5

E. Model Comparison

A comparison of the performance of the various models is shown in Figure 4.

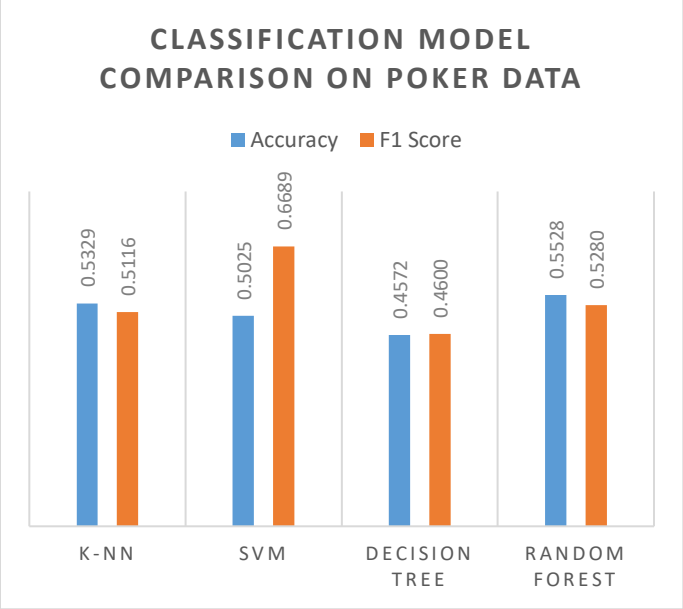

Figure 4: Model Metrics Comparison

Because of the extreme imbalance in the poker dataset, it became clear that just looking at one measure of performance would not be enough so even though I captured accuracy, looking at the weighted F1 score and the comparison of actual value versus prediction by class provided more insight into the appropriateness of each model against the poker dataset. For example, while the SVM model returned the highest accuracy, it essentially put everything into one bucket, which also happened to be the class (0) with the highest percentage of data in the training set (49.95%) and half the classes in the test set. So, while the accuracy for the most prevalent class, 0, was 100% and the F1 score was the highest from amongst the models, this appeared to be overfitting the majority class and ignoring all the others. As a result, I did not consider this a plausible model for further exploration.

In general, I was surprised to see many of the models unable to predict classes that were sparsely represented in the data, such as the *Royal flush* with only five represented in the entire test set. None of the models were able to predict that class. And while not surprisingly the Decision Tree classifier was able to predict all nine classes, it did so with the lowest accuracy and F1 score. Of all the models, the Random Forest had the highest accuracy and F1 score once the SVM was dismissed as described prior, so that was the model used against the larger test dataset.

IV. Model Training

Once the Random Forest model was selected, I moved over to the Spark MLlib to predict the large (1,000,000 row) test dataset. Spark is an engine *"designed for large-scale distributed data processing…"* [3] or in other words, makes processing models on large datasets accessible. Spark MLlib provides libraries to perform common machine learning tasks such as feature scaling, model training, and prediction at scale.

Since each tree can be trained independently in a random forest, multiple trees can be trained in parallel [4], which nicely leverages the benefits of using Spark MLlib.

There were several steps required to prepare the datasets to be used with the Spark machine learning workflow, train the model, and run the model against the test data. Following are details about each of these steps, including some experiments I tried, and the optimal values found.

A. *Initiate a Spark session to load the data*

The first step in the process was to initiate a Spark session. Once initiated, both the training and test data were read into Spark DataFrames for further processing.

B. *Convert Features to Vectors*

Since the data was provided already broken into training and testing, the next steps involved preparing both sets of data to be used in training and prediction respectively.

Random Forests in MLlib require all features to be contained in a single vector within a Spark DataFrame, so this was the first step in preparing the data. Since the poker dataset consists of all categorical data that is represented by integers, I only needed to use the VectorAssembler function to add a feature vector representing each data element's values in the DataFrame.

C. *Index Features and Labels*

One option with training a model using MLlib is to index both the features and the labels. For several initial experiments in training the model, I used the indexing and one-hot-encoding.

Specifically, I used StringIndex function to convert the value of each categorical variable and label to an index associated with the frequency of that value appearing in the data. I also tried using one-hot-encoding (using OneHotEncoder) on the features of the data to see if a different representation would yield better results. With these transformations, however, the performance of the model actually decreased by approximately 5-7% and increased the number of features from 10 to 85, adding model complexity. As a result, both of these steps were removed from the final workflow.

D. *Training the Model*

There are several options available when training a random forest model that can be used to tune the model for optimal performance. Specific parameters I played with include maxDepth, maxBins, and featureSubsetStrategy.

1) maxDepth controls how deep the tree can go. This setting, out of all of them, had the lsrgest impact on model performance. At the minimum and recommended starting value of 4, the F1 score dropped to 0.3994 and only the first two classes were predicted. I was able to improve the performance significantly by increasing the value of this parameter. At maxDepth of 20, the accuracy and F1 score were close to optimal, but only eight classes were predicted. By setting this parameter at the maximum of 30, which is currently the largest allowed in Spark MLlib, the algorithm reached what appears to be an optimal balance of accuracy and F1 score.

2) maxBins controls how many bins the algorithm can use to fork the tree. I started at 210, but worked to optimize the minimum number of bins to have maximum performance with minimum run time. The minimum value of 13, which is the maximum number of categories for a card's face value turned out to be the optimal as anything higher showed no improvement in performance (accuracy or F1 score).

3) featureSubsetStrategy controls how many of the features are used in training the model. Given that each of our features represents a distinct characteristic of the hand that is generally crucial for classifying the data, at least in the current representation of the data. However, setting this value to *True* provided a fairly insignifanct increase in accuracy and F1 score (a gain of 0.00158 and 0.00284 repspectively).

E. *Use the Model to Predict Test Classes*

After several tweaks of the model parameters as described in the previous section, the best model performance metrics are shown in Table 5 followed by the confusion matrix in raw form (Figure 5) and full visual in Figure 6. This performance was obtained without any indexing and using the model values discussed in the previous section.

Table 5: RF Performance with Spark MLlib

Random Forest – Best Performance	
Accuracy	0.599915
F1 Score	0.57336
Number of Classes Predicted	9

Predicted Result	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
Actual Result										
0	384449	116599	129	17	7	7	1	0	0	0
1	206508	213963	1737	251	34	1	2	0	2	0
2	12840	33520	1131	121	4	0	6	0	0	0
3	4365	16007	393	349	3	0	1	3	0	0
4	606	3223	41	3	11	0	0	0	0	1
5	1645	342	0	0	0	9	0	0	0	0
6	97	1188	113	22	1	0	3	0	0	0
7	8	183	24	15	0	0	0	0	0	0
8	3	7	0	0	2	0	0	0	0	0
9	0	3	0	0	0	0	0	0	0	0

Figure 5: Confusion Matrix: Raw Data

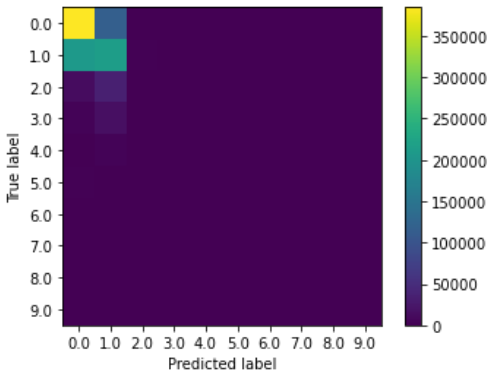

Figure 6: Confusion Matrix

V. Conclusions

The nuances of the data in the poker dataset made this an interesting project to work on. First, the data represented to include sequence of draw added unnecessary complexity, adding to the number of permutations and combinations needing to be considered.

Additionally, the imbalanced distribution of data – with 92% of the test data in two of the ten classes – made the classification more challenging. Most of the models evaluated failed to predict several classes. On the other extreme, SVM just assigned all test data to the most prevalent class, making it perform no better than chance but with the highest F1 score. However, predicting more classes in and of itself didn't mean a better performing model as it tended to reduce the accuracy and F1 score, as seen with the Decision Tree model. However, because of the unbalanced class distribution, more attention was paid to F1 score over accuracy as a more valid measure of performance.

Once the Random Forest model was chosen, two types of transformations were used in initial experiments aimed at improving the model performance. Unfortunately, these attempts actually reduced the performance and added processing time, storage requirements, and complexity. The best bet for improving the performance of classifying the poker dataset perhaps lies in transforming the data in other ways to remove the constraints of draw sequence or utilizing neural networks. For this project, it was clear that the simple approach resulted in the best performance. Still, reaching a high F1 score of 0.57 means this may not be the best task for a gambler.

VI. References

[1] https://www.kaggle.com/rasvob/uci-poker-hand-dataset

[2] Chakure, Afroz. 2019. "Random Forest Regression." Medium. https://medium.com/swlh/random-forestand-its-implementation-71824ced454f.

[3] Damji, Jules, Wenig, Brooke, Das, Tathagata, and Lee, Denny. Learning Spark. 2nd ed. O'Reilly Media Inc., 2020.

[4] https://databricks.com/blog/2015/01/21/random-forests-and-boosting-in-mllib.html