Poker Rule Induction (Group P-1)

(Type 2 project)

University of Ottawa

6th of December 2019

Dilanga Algama (#8253677)

Mitchell Chatterjee (#8598617)

Professor: Caroline Barrière

Course: CSI4106

Link to repository: https://dev.azure.com/mchat022/CSI4106-Project

Timesheet

Team Member	Task	Hours Spent	Challenge
Mitchell C.	Read Article by William Cattral et al	2 hours	A very technical paper. Did not explain clearly how to translate the concepts to application.
	Read article by Ahmed Gad	2 hours	Translating the example problem and concepts to my own problem took substantial revision.
	Defined the Rule class	5 hours	Learned how to implement crossover and mutations. Learned how to define a rule. Had to modify the class based on modifications to the GeneticProgramming class. Testing.
	Defined the GeneticProgram class.	10 hours	Learned how to test fitness of rules. Learned how to define each generation using the genetic operations. Modified class to conform to changes in Rule class. Testing
	Feature Extraction	4 hours	Determined and developed methods for extracting features from the given hand. Found in the CommonMethods file.
Dilanga A.	Read comments by Todd Vance and Soumil Jain on Kaggle	3 hours	Comments about features that could help distinguish different hands in poker.
	Implement features and created the SVM model	15 hours	Creating and testing all the feature functions/sub-functions and creating 2 cvs files to add the features for each hand in the training and testing set .
	Training and testing model	5 hours	Since the test set didn't have the hand column which mentions what hand each row is I had to implement what hand each row is in the test set

			and add this column to the data set to compare the model's results to the actual results.

Introduction

The goal of this project was to train a model to induce the rules of a poker hand without any knowledge of the rule itself, as defined by the Kaggle competition "Poker Rule Induction". Given a set of labelled hands in our training set that represented the different types of poker hands (Straight, Royal Flush, etc), the context of the problem is to train a model to induce the rules that identify each hand uniquely. This problem is important as it can be extended to a diverse set of problems where explicitly programming each rule would be time consuming. Instead, having a model that we could adapt to learn these rules given simple feature extraction, from a large dataset ,would be hugely advantageous.

Dataset Description

The dataset we used for our two approaches originated from a Kaggle competition called "Poker Rule Induction" (https://www.kaggle.com/c/poker-rule-induction/overview). We were given two data set, a training set including the hand value and a test set that does not include the hand value. For our two approaches the only two csv files we used are the training and test data sets.

For the training set we were given 11 columns, 5 of which were the suits for the 5 cards in each hand and another 5 columns for the ranks for the 5 cards in each hand (in a game of poker every hand has five cards). Lastly, we were given a hand column that indicated the integer identity for each hand (e.g. 1 for a One Pair hand, or 6 for a Full House hand etc). The Training set contained 25,010 rows and as mentioned above 11 columns.

For the test set we were given 11 columns as well, where the first column is a unique id for each row in the set and the rest is very similar to the training set where we are given 5 column with the suits for each hand and another 5 columns with the ranks for the 5 cards in the hand. The test set contains 1,000,000 rows and as mentioned above 11 columns.

Both the training and test set didn't require any cleaning, the format of both files was in a good format for both our approaches to read. We ran both our approaches on the whole set to see how well our approaches would perform when given large datasets. Furthermore, we weren't given any features in either dataset other than the columns mentioned above.

Methods Used

Method 1: Genetic Programming

The first method used for inducing poker rules was a genetic approach. We define a rule as a conjunction of 3 different features which have been extracted during the feature extraction process. A rule resembles the following example (Where the rule number represents the hand we are attempting to classify):

$$\text{Rule}(\#) = \text{feature}(1) \land \text{feature}(2) \land \text{feature}(3)$$

A genetic algorithm defines in each generation a set of solutions that are derived from the previous generation. In the first generation we define solutions at random, we consider this our seed generation.

In the next generation we define the first two rules as the best two rules from the previous generation. The third rule is then defined by a crossover operation between our best and second best rule. The fourth rule is defined as a mutation of our best solution. The final rule is defined as a new randomly generated solution. This combination of operations ensures that we maintain our best solutions throughout each generation while providing a degree of randomness when deriving new rules from predefined operations, or randomly.

This approach was selected as it lent itself well to defining rules for a poker hand. Due to the nature of the problem that defined a rule as a conjunction of testable features, a genetic algorithm could be adapted to the problem. In this case a genetic algorithm suitably defined a generation of rules and outlined the process by which to adapt a rule to increase accuracy while maintaining a current maximal solution. Thus, with minimal feature extraction, we are capable of defining a rule. Most importantly, however, the genetic algorithm in its implementation is unique as it provides in the end an optimal, human readable, solution that can then be verified by a human for correctness. When compared to the SVM classifier which does not provide us with an interpretable answer by which to classify poker hands, the genetic approach provides solutions that can be qualitatively analyzed for their correctness. Thus, we can emphasize the difference between the two approaches.

Method 2: SVM Classifier

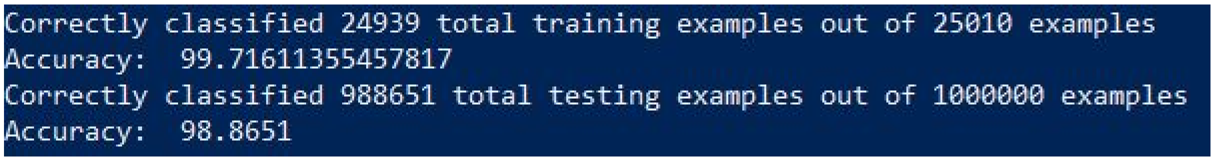
The second method used for inducing poker rules is an SVM approach. SVM classifiers works using features to categorize data points. After all the data points are categorized, a separator between the categories is found and drawn, this separator is called the hyperplane. The hyperplane acts like a boundary dividing all the different categorizes, where in our project are different hands in the game of poker.

We chose SVM to be one of our approaches because we wanted to evaluate how well a discriminative learner would perform in the scenario like our project and compare it with the genetic learner and visualize how well one performs compared to the other. We also wanted to analyze how different the learning process is between the two approaches.

For the SVM approach we used many libraries available for python. One of the main libraries we used, especially for training, was sklearn. The sklearn library is a simple and efficient tool for data mining and data analysis. After we had implemented the different features that we think will help the classifier classify the hands correctly, we used the sklearn classifier on our training set with our features for the classifier to learn the pattern in our data and differentiate the data in a way where is can correctly predict any given hand based on what is learnt from the training set. Another very useful library we used for the SVM approach is the Pandas library which helped us and our model analyze the data we were given from the data set.

After we trained the model using training set features with the sklearn library, the SVM approach algorithm was analyzed by comparing the predicted value for each hand from the model with the hand column in the dataset which indicated what hand that specific row is. We printed out of all the rows how many the model classified correctly.

When we ran the model on the training set where we got a 99.7% accuracy as stated in figure 1 and when we ran the model on the test set where we got a 98.9% accuracy as stated in figure 1.



```
Correctly classified 24939 total training examples out of 25010 examples
Accuracy:  99.71611355457817
Correctly classified 988651 total testing examples out of 1000000 examples
Accuracy:  98.8651
```

Figure 1: the Training set and Test Set SVM Model accuracy

Feature Engineering

Method 1: Genetic Programming

All features extracted for the genetic programming approach made use of the absolute difference of values in the hand. First we separated the hand into ranks and suits and sorted both arrays. Next we calculated the absolute difference between all values in the array and returned the array containing the absolute difference. The features extracted from this array include:

- numZeros(hand) == #
  - This method calculates the number of zeros in the absolute difference array representing either suits or ranks. The number of zeros represents the number of adjacent elements that have the same value. [7, 8, 12, 12, 12] → [1, 4, 0, 0]
- separationOfZeros(ranks)
  - This method returns a boolean value that indicates whether the zeros in the absolute value array are separated. If they are separated this indicates that we have multiple instances of identical items in the hand. [1, 1, 2, 2, 3] → [0, 1, 0, 1]
- elemsAscDesc(ranks, isRank=True)
  - This method tests whether the cards in the hand are ascending or descending according to the absolute difference array. This indicates that we have a straight. [1, 2, 3, 4, 5] → [1, 1, 1, 1]

The qualitative importance of these features is that they allow us to consider the many variations of a specific poker hand with minimal features.

The quantitative importance of these features can be readily demonstrated by inspecting the output of the genetic algorithm as it seeks to discover new features to add to the rules in order to classify the hand.

Method 2: SVM Classifier

We implemented 6 features in total for both the training set and the test set (Flush, Rank1, Rank2, Highest, Lowest, Straight). We got the idea of few of these features by reading comments on Kaggle's comment section specially comments by Todd Vance and Soumil Jain (1).

The different features we chose, their qualitative and quantitative importance is mentioned below.

- Flush - This feature on our features csv file is a boolean value where it would be 1 if the current hand is a flush and 0 otherwise. A flush is a hand that contains five cards all of the same suit, not all of sequential rank, such as [K♣ 10♣ 7♣ 6♣ 4♣].
- Rank1 - This feature contains the most frequent card/rank. We created this feature mainly to recognize a four-of-a-kind, three-of-a-kind, two-pair or one-pair. It also acts like a helper function for the Rank2 feature to recognize a full house.
- Rank2 - This feature contains the second most frequent card/rank for each hand. We create this feature to work together with Rank1 for the model to recognize a full house. A *Full House* is any three cards of the same number or face value, plus any other two cards of the same number or face value, such as [K♣ K♥ K♠ 4♠ 4♣].
- Lowest - This features contains the lowest card in every hand. We created this feature to recognize a royal flush when all the numbers are low numbers. A *royal flush* five cards in numerical order, all of identical suits,such as [1♠ 2♠ 3♠ 4♠ 5♠].

- Highest - This feature contains the highest card in every hand. We created this feature as a helper for the model to recognize the difference between a royal flush and straight flush.
- Straight - This feature contains a boolean value where it would be 1 if the current hand is a straight hand and 0 otherwise. (Just explain what a straight is in this context). A *straight* is a hand that contains five cards of sequential rank, not all of the same suit, such as[7♣ 6♠ 5♠ 4♥ 3♥].

Analysis of Results

Method 1: Genetic Programming

The genetic programming algorithm was analyzed by predicting the accuracy of the solution in correctly predicting each hand. The correct classification of each hand fell under exactly one of two possibilities. First the solution has labelled this hand as being of the specified type (ie. Royal Flush) and then hand value indicates that this is true. Second, the solution has labelled this hand as being of a different type (ie. Other) and the hand value is different from the one we are trying to induce.

Thus we define the accuracy of the model in two parts. One, its ability to define the hands in this group, which are calculated as the number of correct classifications divided by the total number of instances of this hand. Two, its ability to define the hands not in this group, which are calculated as the number of correct classifications divided by the total number of instances of every other hand. Together these two criteria allow us to define the accuracy of the solution.

In order to test the accuracy of the results, we extract the best rule for each hand after running the genetic program and place them in a hierarchy of if statements from most to least specific. This allows us to test the accuracy of the rules on the test hand at classifying the hand. We can also qualitatively analyze the results by comparing the resulting rules to those that we have predefined for their accuracy. For example, the rule produced by the genetic program for the classification of hand 1 (One-pair) was:

{'ClassificationOfTargetHand': 1.0, 'ClassificationOfOtherHands': 1.0}
cm.numZeros(ranks) == 1 and not cm.separationOfZeros(ranks) and True

Using the same features defined in the program, the actual rule for this hand is:

cm.numZeros(ranks) == 1 and True and True

When calculating the accuracy of the extracted rules considering a hierarchy of rules as the solution. Our genetic program gave an extremely good result as follows:

Correctly classified 946410 total test examples out of 1000000 examples
Accuracy: 94.641%

The rules that were extracted and placed in the hierarchy can be seen in the classifyHand method In the GeneticProgram file.

Conclusion

Each approach took an extremely different concept and applied it to the problem of classifying and inducing the rule of a hand in poker.

The genetic approach allowed us to define rules as a conjunction of features in much the same way that they are defined in their essence. This in turn took a much more intuitive approach to the problem as it randomly created rules and attempted to modify the most suitable rules in order to obtain an exact match. The SVM by comparison approached the problem by attempting to classify each hand using hyperplanes that divide the features into separate groups.

Each approach required feature extraction and feature engineering in order to train the model. In the case of the genetic approach, all features revolved around the existence of an array containing the absolute difference of the hand with respect to its rank and hand values. While the SVM utilized a much wider array of features for each hand with respect to the ranks and suits.

The accuracy of the genetic approach in classifying each hand varied on each iteration of the algorithm as it was randomized at the beginning, this lead to a non-deterministic output for the final solution of each rule. The SVM, however, was consistent in the way that it classified points throughout multiple iterations of the algorithm and returned a constant accuracy for both data sets.

In conclusion, both approaches worked well for the classification of Poker Hands, but only the genetic approach provided us with a tangible solution. The SVM however, attained a greater accuracy.

References:

1. “Poker Rule Induction.” *Kaggle*, https://www.kaggle.com/c/poker-rule-induction/discussion/11177#latest-72369.
2. “Evolutionary Data Mining With Automatic Rule Generalization” http://www.wseas.us/e-library/conferences/crete2002/papers/444-494.pdf
3. “Example Genetic Programming Library” https://commons.apache.org/proper/commons-math/javadocs/api-3.1/org/apache/commons/math3/genetics/package-summary.html
4. “Genetic Algorithm Implementation in Python” https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6