Poker Hand Induction

Alan Dodge, Spencer McEwan, Jessica Thomas

April 6,2015

Abstract

In this project, we attempt to predict poker hands from 5-card training tuples. Using a data set obtained from Kaggle, we generate rules based on a training data set and classify testing data based on these rules. Upon applying our rules to a testing data set we obtained 99.9843% coverage and 99.868% accuracy when we set unclassified hands to the 'not a valid poker hand' classification.

1 Introduction

This project focuses on determining poker hands from five playing cards by generating rules for classification. Our data set for this problem was obtained from Kaggle, a platform for data science competitions. Kaggle hosts various predictive modeling competitions ranging from detecting signs of disease to making predictions about the weather. These competitions range in difficulty and reward. As a result, these challenges attract individuals with varying levels of experience with respect to predictive modeling and analytics. The Kaggle competition we are exploring for this project is called "Poker Hand Induction."

Our goal was to predict poker hands given only the cards without assuming any rules of the game of poker. To accomplish this, we had to make some basic assumptions about a deck of cards. For example, we expect that a deck of cards contains unique suits, each card within a suit has a rank, and that the cards have an order. That said, we could not use our existing knowledge of poker to engineer features that would make classification easier. For instance, we can't teach the program that if a hand contains only two cards of the same rank, we have a pair. Similarly, we could not assume that a hand not falling into any other category could be classified as 'not a valid poker hand'. In general, a program with any hard-coded information specific to the game of poker is considered invalid; in theory, by the competition rules, our program should be able to determine the rules for any card game with named hands (and a classified training set) as long as the deck used adhered to the assumptions made above. In summary, we needed to make these discoveries through machine learning on a testing data set. The hand classifications provided by Kaggle are as follows:

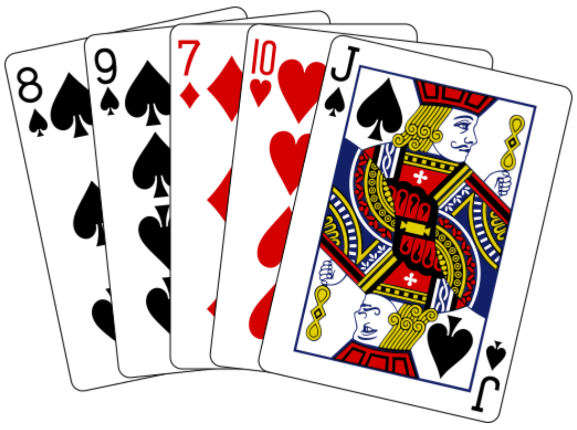
```
0: Nothing in hand; not a recognized poker hand
1: One pair; one pair of equal ranks within five cards
2: Two pairs; two pairs of equal ranks within five cards
3: Three of a kind; three equal ranks within five cards
4: Straight; five cards, sequentially ranked with no gaps
5: Flush; five cards with the same suit
6: Full house; pair + different rank three of a kind
7: Four of a kind; four equal ranks within five cards
8: Straight flush; straight + flush
9: Royal flush; {Ace, King, Queen, Jack, Ten} + flush
```

Figure 1: Kaggle poker hand classifications

Two .csv files containing suits and values for each of the five cards were provided for this competition. The first file contained 25,010 training poker hands that were already classified. The second file consisted of 1 million test hands that were not already classified, but would be used as the basis for the submission file for the competition. In order to be able to analyze the training data, we transformed the .csv file into python dictionaries.

Here is an example of a reformatted training hand: {’S1’:’4’, ’C1’:’8’, ’S2’:’4’, ’C2’:’9’, ’S3’:’2’, ’C3’:’7’, ’S4’:’1’, ’C4’:’10’, ’S5’:’4’, ’C5’:’11’, ’hand’:’4’}

The following is the same hand as its input representation and corresponding physical representation[5].



```
S1 C1 S2 C2 S3 C3 S4 C4 S5 C5 hand
4  8  2  9  2  7  1  10 4  11 4
```

Figure 2: Example straight

Our approach to this problem consisted of three steps. We generated rules for all the training hands. We then generalized the rules based on patterns between hand classifications. Finally, we tested our rules using k-fold cross-validation and then we classified the testing hands. Using both k-fold cross validation and the testing data we were able to have very good coverage and accuracy (very close to 100% in both cases). We will explore problems and challenges we encountered in the conclusion.

2 Relevant Work

In terms of relevant work, we found it most interesting to look at how others approached the same data set. Therefore, we will look at two sources that also generated rules for our training data set. We will also briefly explore how other users in the Kaggle Competition approached this problem, as accessed by the forum on the competition page.

The Kaggle competition provided a relevant paper by Robert Cattral, Franz Oppacher and Dwight Deugo from Carleton University[4]. In this paper they use a rule generation method called RAGA. This technique implements a genetic algorithm and a genetic programming engine to determine similarities between hands of the same class and to generate relevant rules. RAGA then uses a number of metrics to rank these rules in order to determine which are the most relevant.

Another approach, by Nikola Ivanić from the University of Belgrade[3], uses neural networks in the classification of this data set. This approach uses software called Neuroph Studio to generate learning neural networks of various sizes and dimensions to determine which produces the optimal results.

Upon exploring the Kaggle competition forum, we observed that other competitors on the Kaggle competition used of Random Forests and CART to classify the data. Competitors appeared to achieve varying results with both algorithms depending on how they were implemented. We also noticed varying degrees of feature engineering such as sorting the cards. Sorting the cards is an example of feature engineering because it is plausible that a card game could exist in which the order you receive your cards in your hand is important. However, in poker, order is irrelevant. We explore aspects of feature engineering later in the paper.

3 Data Mining

Our approach to predicting poker hands involved three parts. First, we generated rules for each hand in the training data set. We then generalized these rules based on hand classification. Finally, we evaluated the rules of our testing data set.

3.1 Generate

Due to the nature of the data set and the Kaggle competition, the data set needed very little preprocessing. However, in order to generate our rules in Python, we needed to import the card hands from the training data set into dictionaries. The generate phase of our analysis had us summarizing all possible information about each hand. We generated metrics between each pair of cards in each hand.

```
!: Rank not equal; cards do not have equal value (rank)
=: Rank equal; cards have same value
e: Number of equal ranks
m: Max rank in hand
n: Min rank in hand
v: Average of ranks in hand
+: Adjacent; cards have same value +/- 1 but are not equal
a: Number of adjacent cards
#: Suit equal; cards have the same suit
%: Suits not equal: cards have different suits
s: Number of same suits
```

Figure 3: Metric definitions and corresponding symbols

Our script looked at each pair of cards in each hand and kept track of whether the pair of cards had the same or different values and if they had the same or different suits. We kept a total of both the number of cards with the same suit and the number of cards with the same value. We defined two cards as adjacent if their values differed by one. We kept record of this relationship between card pairs and also kept a total number of adjacencies for the entire hand. Finally, we recorded the maximum card in the hand, the minimum card in the hand and also the average of all five card values in the hand.

```
THE POKER HAND: S1  C1  S2  C2  S3  C3  S4  C4  S5  C5  hand
                4   8   4   9   2   7   1   10  4   11  4

THE GENERATED RULE: {'4': [(1, '!', 2), (1, '!', 3), (1, '!', 4), (1, '!', 5),
                    (2, '!',3),(2, '!', 4), (2, '!', 5), (3, '!', 4),
                    (3, '!', 5), (4, '!', 5),('e', 0), ('m', 11), ('n', 7),
                    (1, '#', 2), (1, '%', 3), (1, '%', 4), (1, '#', 5),
                    (2, '%', 3), (2, '%', 4),(2, '#', 5), (3, '%', 4),
                    (3, '%', 5), (4, '%', 5),('s', 3),(1,'+', 2), (1, '+', 3),
                    (2, '+', 4), (4, '+', 5), ('a', 4),('S3', 2),('S2', 4),
                    ('S1', 4), ('S5', 4), ('S4',1), ('C3', 7), ('C2', 9),
                    ('C1', 8), ('C5', 11), ('C4', 10)]}
```

Figure 4: Example hand input and corresponding generated rule in the Generate phase

3.2 Generalize

We then filtered out all of the characteristics that are not common to all hands of the same class. In a sense, we implemented a backwards version of the Apriori algorithm. Instead of slowly building up a rule set, we start with a full rule set and eliminate elements one by one that are not common to the hands in question. We start with the rules in the first hand and keep track of the common elements between hands, slowly narrowing down our general rule set.

3.3 Evaluate

Finally, we took our set of generalized rules and used them to classify poker hands. Because the testing data did not include the hand classification, we used k-fold validation on the training data to test the effectiveness of our rules. We divided our data into ten parts and trained on nine of them. Testing on tenth part last allowed us to see if our rules were able to correctly classify poker hands. Once we were confident in our classification rules, we were able to use the script on the testing data set.

4 Conclusion

4.1 Results

In our k-fold cross validation, we found that our classification had an average success rate of 99.864% when we used $k = 10$ and performed the 10-fold cross validation ten times.

Once submitted to the Kaggle competition, we found that our program accurately classified 99.868% of the data, as expected. At the time of submission, our program ranked 37/117, below 23 submissions achieving 100% accuracy. It has been suggested in some submission comments that many high-accuracy submissions have heavy feature-engineering, such as sorting or permuting cards (which could be used to generate more training data) and encoding Ace wrapping as we will discuss in the next section.

4.2 Challenges

As we described in the Introduction, the main challenge of this project is that we could not incorporate any information about the actual game of poker; it would be very easy to obtain 100% accuracy if we could encode the rules we knew about each hand classification.

In an early version of our program, we found that we were misclassifying some straights because of the duplicitous property of Aces: they can either count as a high card (eg. 10, Jack, Queen, King, Ace; in which case our program recognizes three adjacencies), or a low card (eg. Ace, 2, 3, 4, 5; where our program finds four adjacencies). Because of the way we were generalizing rules, we filter out adjacencies as being irrelevant to this hand classification (when as humans we can see that it is the most important part). This property results in all straights being classified as ‘not a valid poker hand’. To fix this issue, we encoded that cards could ‘wrap’ around to the beginning after the highest rank, that is, the highest card is adjacent to the lowest card. This allowed us to classify the straights that we had been missing, but it introduced a new problem: an Ace occurring in the middle of a consecutive sequence of cards is considered a valid straight by our program, but not in an actual poker game (eg. Queen, King, Ace, 2, 3). Even with more training data, our script will always misclassify this type of hand because it fits the rule of having five adjacent cards. In order to fix this and remove the wrapping feature engineering, we could attempt to generalize differently by using rule coverage and rule accuracy, similar to the research paper provided by Kaggle[1].

Another problem we encountered is that the training set data set was too small and did not involve enough hands of each classification type. As a result, in our generalize step, there were quite a few hand classifications that resulted in superfluous rules that all the hands of a certain type just happened to have that were irrelevant. The extra rules caused some hands that were unable to be classified. Since generalize created a set of rules even for the class 0 (not a valid poker hand), hands that could not be classified could not just be set to 0. Instead, we had a -1 class that was used for this kind of hand. We considered setting such hands to 0 was making assumptions about what weren’t poker hands. However, the Kaggle competition did not have an accepted format or class to describe these hands. Therefore, for our competition submission we set all hands classified as -1 to 0. Any valid value could have been used here because none of these hands were actually of class 0 (we checked!).

Notably, 4-of-a-kind hands in the testing data set were often classified as -1. This is because in the training hand there were only six hands that we a 4-of-a-kind. Coincidentally, the second and third card both ranked the same as each other. This caused a problem when running the rules over our training set as many hands that should have been classified as 4-of-a-kind were instead classified as -1 as they did not meet the unnecessary condition created by the training data.

4.3 Future Steps

In the future, we would like to address the problems encountered in the Challenges section. We have determined that in order to achieve an extremely high degree of accuracy, the training data set for this challenge would need to contain more instances of some of the less common poker hands. As we saw in our section on Challenges, there were only six training hands for 4-of-a-kind which caused problems when we applied the rules to the testing data set. Royal flushes are another kind of example of a problematic hand classification. We would also like to look at other methods of generalization so that the wrapping feature could be removed but straights could still be classified with high accuracy.

Another interesting direction in which we could take this problem is applying it to other card games. In theory, we avoided making assumptions about the game of poker including even the size of the hand. Therefore, we could try to predict hands for other card games. These games would not need to have the same structure as poker, but would need to follow the basic assumptions we made about a deck of cards. However, it would be possible to adapt the assumptions to a variety of games like Gin or Rummy, which have named hands, or even games like Uno or Crazy Eights, where a hand could be classified based on the types of "special" cards it contains.

5 References

[1] Kaggle Competition. "Poker Rule Induction". [Online]. Available: https://www.kaggle.com/c/poker-rule-induction

[2] Bache, K. and Lichman, M. (2013). "UCI Machine Learning Repository". Irvine, CA: University of California, School of Information and Computer Science.

[3] Ivanić, Nikola. "Predicting Poker Hands With Neural Networks". [Online]. Available http://neuroph.sourceforge.net/tutorials/PredictingPokerhands/Predicting%20poker%20hands%20with%20neural%20networks.htm

[4] Cattral, R., Oppacher, F., and D. Dwight. (2002) "Evolutionary Data Mining With Automatic Rule Generalization". *Recent Advances in Computers, Computing and Communications*: 296-300,

[5] Aguilar, C. "Vectorized Playing Cards". [Online]. Available https://code.google.com/p/vectorized-playing-cards/

Appendix: Code Listing

Here we provide our Python code from this project.

```
import sys
import csv
import fnmatch

"""
Dictionarys created by csv.DictReader for training file have the following format:
{'S3': '2', 'S2': '4', 'S1': '4', 'S5': '4', 'S4': '1', 'C3': '7', 'C2': '9',
'C1': '8', 'hand': '4', 'C5': '11', 'C4': '10'}
So Keys are: S1, S2, S3, S4, S5, C1, C2, C3, C4, C5 and hand (which is our class value)
"""

"""
Takes a dictionary describing a hand, and the size of the hand.
Generates a list of card values that are the same and different.
Returns this list and the total number of equal relations.
Eg. Given: {'S3': '2', 'S2': '4', 'S1': '4', 'S5': '4', 'S4':
'1', 'C3': '7', 'C2': '9', 'C1': '8', 'hand': '4', 'C5': '9', 'C4': '10'}
Will return: [(2, '=', 5)]
'=' = same value
'!' = different value
'e' = number of equal cards (includes duplicate counting)
"""
def equal_card( training_dict , hand_size ):
    same_set = []
    num_equ = 0

    #For each card in the hand
    for card in range(1,hand_size):
        #Look at the rest of the cards in the hand
        for next_card in range(card+1,hand_size+1):

            #If the current card is the same as the next card, add them to the list
            if training_dict['C'+str(card)] == training_dict['C'+str(next_card)]:
                same_set += [(card, '=', next_card)]
                num_equ += 1 #Add to the total number that are equal
            else: #Else, add that they are different
                same_set += [(card, '!', next_card)]

    same_set += [('e', num_equ)]

    return same_set

"""
Takes a dictionary describing a hand, and the size of the hand.
Generates a list containing the max, min, and avg card values. Assumes cards can not
have a negative value.
Returns max, min and avg.
Eg. Given: {'S3': '2', 'S2': '4', 'S1': '4', 'S5': '4', 'S4': '1', 'C3': '7', 'C2': '9',
'C1': '8', 'hand': '4', 'C5': '9', 'C4': '10'}
Will return: [('m', 10), ('n', 7), ('v', 8.6)]
'm' = max
'n' = min
'v' = average
"""
def maxmin_card( training_dict , hand_size ):
    cur_max = -1
    cur_min = -1
    num_avg = 0

    for card in range(1,hand_size+1):
        #Handles the first round through
        if cur_max == -1:
            cur_max = training_dict['C'+str(card)]
        if cur_min == -1:
            cur_min = training_dict['C'+str(card)]
        #Sets max and min
        if training_dict['C'+str(card)] > cur_max:
            cur_max = training_dict['C'+str(card)]
        if training_dict['C'+str(card)] < cur_min:
            cur_min = training_dict['C'+str(card)]

        num_avg += training_dict['C'+str(card)] #Total value seen

    num_avg = num_avg / hand_size #Calculate avg

    return [('n', cur_min),('m', cur_max), ('v', num_avg)]

"""
Takes a dictionary describing a hand, and the size of the hand.
Generates a list of card suits that are the same and those that are different.
Returns this list and the number of same suit.
Eg. Given: {'S3': '2', 'S2': '4', 'S1': '4', 'S5': '4', 'S4': '1', 'C3': '7',
'C2': '9', 'C1': '8', 'hand': '4', 'C5': '9', 'C4': '10'}
Will return: [(2, '=', 5)]
'#' = same suit
'%' = different suit
's' = number of same suit (includes duplicate counting)
"""
def equal_suit( training_dict, hand_size ):
    same_suit = []
    num_equ = 0

    #For each card in the hand
    for card in range(1,hand_size):
        #Look at the rest of the cards in the hand
        for next_card in range(card+1,hand_size+1):
            #If the current suit is the same as the next suit, add them to the list
            if training_dict['S'+str(card)] == training_dict['S'+str(next_card)]:
                same_suit += [(card, '#', next_card)]
                num_equ += 1 #Increment same suits.
            else: #Else not the same suit.
                same_suit += [(card, '%', next_card)]

    same_suit += [('s', num_equ)]

    return same_suit

"""
Takes a dictionary describing a hand, and the size of the hand.
Generates a list of card values that are adjacent.
Eg. Given: {'S3': '2', 'S2': '4', 'S1': '4', 'S5': '4', 'S4': '1', 'C3': '7',
'C2': '9', 'C1': '8', 'hand': '4', 'C5': '9', 'C4': '10'}
Will return: [(1, '+', 2), (1, '+', 3), (1, '+', 5) (2, '+', 4), (4, '+', 5)]
'+' = Adjacent ( +/-1 from each other)
'a' = Number adjacent
"""
def adjacent_card( training_dict, hand_size, max_rank ):
    adj_set = []
    num_adj = 0

    #For each card in the hand
    for card in range(1,hand_size):
        #Look at the rest of the cards in the hand
        for next_card in range(card+1,hand_size+1):

            #If the current card is one greater or one less than the next card, add them
            to the list
            if (training_dict['C'+str(card)] == (training_dict['C'+str(next_card)]+1))
            or (training_dict['C'+str(card)] == (training_dict['C'+str(next_card)]-1)):
                adj_set += [(card, '+', next_card)]
                num_adj += 1
            #Handles circular potential for 13 card suits, would need to add detection to
            calculate number to mod by
            elif((training_dict['C'+str(card)]%(max_rank-1)) == (training_dict['C'+
            str(next_card)]%(max_rank-1))) and (training_dict['C'+str(card)] !=
            training_dict['C'+str(next_card)]):
                adj_set += [(card, '+', next_card)]
                num_adj += 1

    adj_set += [('a',num_adj)]

    return adj_set

"""
Takes␣a␣list␣of␣dictionaries␣from␣training␣file.
Generates␣rules␣based␣on␣simple␣assumptions␣for␣every␣hand,␣and␣adds␣them␣to␣a
dictionary␣of␣hand␣rules.
Returns␣this␣dictionary␣of␣hand␣rules.
"""
def generate( training_reader ):
    hands = {}
    max_card = 0

    #For each hand in the training set
    for line in training_reader:
        num_cards = 0
        card = []

        #Turn all the values in line into integers
        for key in line:
            line[key] = int(line[key])

            # get rid of the hand key, and create a list of the cards and their values
            if key != 'hand':
                card += [(key, line[key])]

            #Count number of cards in hand
            if fnmatch.fnmatch( key, 'C?'):
                num_cards += 1
                if line[key] > max_card:
                    max_card = line[key]

        equ = equal_card( line, num_cards ) #Same and different values
        maxmin_num = maxmin_card( line, num_cards ) #Max, min and avg
        same = equal_suit( line, num_cards ) #Same and differnt suits
        adj = adjacent_card( line, num_cards, max_card ) #Adjacent cards

        #Add rules to respective hand classifications
        if str(line['hand']) not in hands.keys(): #First time a class has been seen
            hands[str(line['hand'])] = [equ+maxmin_num+same+adj+card]
        else: #Every other time
            hands[str(line['hand'])].append(equ+maxmin_num+same+adj+card)
    return hands

"""
Takes␣a␣dictionary␣of␣hand␣classes,␣which␣refer␣to␣lists␣of␣rules␣for␣indevidual␣hands.
For␣each␣class,␣for␣each␣hand␣in␣that␣class,␣compairs␣rule␣by␣rule␣and␣eliminates␣all
rules
that␣are␣not␣common␣to␣both.
Produces␣a␣minimum␣list␣of␣rules␣required␣to␣identify␣a␣hand␣classification␣with␣100%
coverage.
"""
def generalize( hands ):
    generalized_rules = {}

    #For each classification option for hands
    for key in hands:
        #Prints number of hands of each class used in training
        #print ('Hand : ' + key + ' Number: ' + str(len(hands[key])))

        #Set the base rules to the rules of the first hand
        rules = hands[key][0]

        #For each hand of class 'key'
        for hand in hands[key]:
            #Duplicate the list of rules
            new_rules = list(rules)
            #Go through each rule
            for rule in rules:
                #If the rule does not exist in the current hand, remove it from the list of rules.
                if rule not in hand:
                    new_rules.remove(rule)

            rules = list(new_rules) #Update the list of rules.

        generalized_rules[key] = rules #Add the rules to the generalized set after going
        through all the hands for that class.

    return generalized_rules

"""
Takes␣a␣list␣of␣hands␣to␣be␣classified,␣and␣the␣rules␣generated␣in␣training.
Generates␣rules␣for␣each␣hand␣to␣be␣classified␣and␣attempts␣to␣classify␣them␣using␣the
rules.
Assumes␣-1␣is␣not␣a␣class.
Returns␣a␣list␣of␣dictionaries␣where␣each␣has␣an␣'id'␣corresponding␣to␣the␣order␣found
in,
and␣a␣'class'␣corresponding␣to␣the␣classification
-1:␣Could␣not␣classify.
0:␣Nothing␣in␣hand;␣not␣a␣recognized␣poker␣hand
1:␣One␣pair;␣one␣pair␣of␣equal␣ranks␣within␣five␣cards
2:␣Two␣pairs;␣two␣pairs␣of␣equal␣ranks␣within␣five␣cards
3:␣Three␣of␣a␣kind;␣three␣equal␣ranks␣within␣five␣cards
4:␣Straight;␣five␣cards,␣sequentially␣ranked␣with␣no␣gaps
5:␣Flush;␣five␣cards␣with␣the␣same␣suit
6:␣Full␣house;␣pair␣+␣different␣rank␣three␣of␣a␣kind
7:␣Four␣of␣a␣kind;␣four␣equal␣ranks␣within␣five␣cards
8:␣Straight␣flush;␣straight␣+␣flush
9:␣Royal␣flush;␣{Ace,␣King,␣Queen,␣Jack,␣Ten}␣+␣flush
"""
def classify( test_list, rules ):
    count = 0
    max_card = 0
    classified = []
    # generate a rule for each hand in test list
    for hand in test_list:
        num_cards = 0
        card = []
        count += 1

        #Turn all the values in line into integers
        for key in hand:
            hand[key] = int(hand[key])
            # get rid of the hand key, and create a list of the cards and their values
            if key != 'hand':
                card += [(key, hand[key])]

            #Count number of cards in hand
            if fnmatch.fnmatch( key, 'C?'):
                num_cards += 1
                if hand[key] > max_card:
                    max_card = hand[key]

        equ = equal_card( hand, num_cards ) #Same and different values
        maxmin_num = maxmin_card( hand, num_cards ) #Max, min and avg
        same = equal_suit( hand, num_cards ) #Same and differnt suits
        adj = adjacent_card( hand, num_cards, max_card ) #Adjacent cards

        hand_rules = equ + maxmin_num + same + adj + card #Compiled list of rules for the
        hand

        classification = '-1' #Indicates unclassified

        #Simmilar to generalize, finds matching set of rules
        for key in rules:
            new_rules = []
            for rule in rules[key]:
                if rule in hand_rules:
                    new_rules += [rule]

            #If the final set of rules is the same as the rules for the current class being tested:
            if new_rules == rules[key]:
                #If this is the first time it matches, set it to this class
                if classification == '-1':
                    classification = key
                else:
                    #If this classification is more specific (longer) than the previous
                    #class, set it to the new class
                    if len(rules[key]) > len(rules[classification]):
                        classification = key

        classified += [{ 'id':count, 'hand':classification}] #Add it to the classified
        list

    return classified

"""
Takes a list of classified dictionaries, and the list of hands used for classification.
Checks the classification in the dictionary to the known classification from the testing
list.
Tracks the number of correct for each class, and the total number of that class checked.
(Can print that out.)
Returns the total number of correct and the total number of incorrect.
c# = Number of correct for # class.
s# = Number total for # class.
"""
def evaluate( classification, test_list):
    #Initialize totals to 0
    num_right = 0
    num_wrong = 0
    c0 = c1 = c2 = c3 = c4 = c5 = c6 = c7 = c8 = c9 = 0
    s0 = s1 = s2 = s3 = s4 = s5 = s6 = s7 = s8 = s9 = 0

    #For every hand in the test list
    for i in range(0,len(test_list)):
        #Increment the hand class
        if int(test_list[i]['hand']) == 0:
            s0 += 1
        if int(test_list[i]['hand']) == 1:
            s1 += 1
        if int(test_list[i]['hand']) == 2:
            s2 += 1
        if int(test_list[i]['hand']) == 3:
            s3 += 1
        if int(test_list[i]['hand']) == 4:
            s4 += 1
        if int(test_list[i]['hand']) == 5:
            s5 += 1
        if int(test_list[i]['hand']) == 6:
            s6 += 1
        if int(test_list[i]['hand']) == 7:
            s7 += 1
        if int(test_list[i]['hand']) == 8:
            s8 += 1
        if int(test_list[i]['hand']) == 9:
            s9 += 1

        #Check the classification vs. the actual class of each hand
        if int(test_list[i]['hand']) == int(classification[i]['hand']):
            #Increment correct if it was correct
            if int(test_list[i]['hand']) == 0:
                c0 += 1
            if int(test_list[i]['hand']) == 1:
                c1 += 1
            if int(test_list[i]['hand']) == 2:
                c2 += 1
            if int(test_list[i]['hand']) == 3:
                c3 += 1
            if int(test_list[i]['hand']) == 4:
                c4 += 1
            if int(test_list[i]['hand']) == 5:
                c5 += 1
            if int(test_list[i]['hand']) == 6:
                c6 += 1
            if int(test_list[i]['hand']) == 7:
                c7 += 1
            if int(test_list[i]['hand']) == 8:
                c8 += 1
            if int(test_list[i]['hand']) == 9:
                c9 += 1
            num_right += 1
        else:
            num_wrong += 1
            #print (test_list[i])
            #print ('Expect: ' + str(test_list[i]['hand']) + ' Produced: ' +
            str(classification[i]['class']))


    #Used to print the number of correct and total for each classification
    print ('0: Right: ' + str(c0) + ' out of: ' + str(s0))
    print ('1: Right: ' + str(c1) + ' out of: ' + str(s1))
    print ('2: Right: ' + str(c2) + ' out of: ' + str(s2))
    print ('3: Right: ' + str(c3) + ' out of: ' + str(s3))
    print ('4: Right: ' + str(c4) + ' out of: ' + str(s4))
    print ('5: Right: ' + str(c5) + ' out of: ' + str(s5))
    print ('6: Right: ' + str(c6) + ' out of: ' + str(s6))
    print ('7: Right: ' + str(c7) + ' out of: ' + str(s7))
    print ('8: Right: ' + str(c8) + ' out of: ' + str(s8))
    print ('9: Right: ' + str(c9) + ' out of: ' + str(s9))


    return num_right, num_wrong

"""
Takes list of dictinaries corresponding to hands in the training file
and a current mod_num (number to split on)
and k, the variable for k-fold cross-validation
Separates out 1/kth of the training data as classified testing data.
Returns as list of training hands and a list of testing hands.
"""
def generate_test_training(training_list, mod_num, k):
    test_list = []
    train_list = []

    # every mod_numth line goes to a test list, the rest to a training set
    for index in range(len(training_list)):
        if index % k == mod_num:
            test_list.append(training_list[index])
        else:
            train_list.append(training_list[index])

    return test_list, train_list

"""
Tests our classification accuracy using k-fold cross validation
Input: list of dictionaries corresponding to hands in the training file
Output: classification accuracy in terms of right/wrong for each chunk
"""

def run_k_fold_test(training_list):
    # how many chunks to split training set into for k fold cross-validation
    k = 10

    # divide training list 10 different ways and perform classification on each fold
    for i in range(0,k):
        test_list, train_list = generate_test_training(training_list, i, k) #Separate
        training and testing
        hands = generate( train_list ) #Generate rules for each hand
        rules = generalize( hands ) #Generalize the rules into minimum sets

        classification = classify(test_list, rules) #Classify the testing set
        right, wrong = evaluate(classification, test_list) #Evaluate correctness of
        classification of testing set

        print ("Right␣=␣", right, "␣Wrong␣=␣", wrong)

"""
Classifies␣all␣tuples␣in␣the␣test␣set␣using␣our␣rules␣from␣the␣training␣data.
Input:
Output:␣a␣file␣of␣our␣submission
"""
def gen_competition_submission(training_list, test_list):
    # GENERATE RULES FOR TEST DATA SET:
    # use training data to generate saturated rule set
    hands = generate( training_list )
    # generalize saturated rule set
    rules = generalize( hands )

    # classify test_set using the general rules
    classification = classify( test_list, rules )

    out_file = open('output.csv', 'w')
    fieldnames = ['id', 'hand']
    out_writer = csv.DictWriter(out_file, fieldnames=fieldnames)

    out_writer.writeheader()
    for line in classification:
        out_writer.writerow(line)

#WHEN RUNNING: First arg is training file, second is classification file. Will create a csv file call
def main():

    if len(sys.argv) < 2:
        print ("There␣are␣two␣uses␣for␣this␣code:")
        print ("1:␣When␣a␣single␣file␣is␣input,␣it␣will␣run␣10-fold␣cross␣validation
␣␣␣␣␣␣␣␣␣␣␣␣on␣that␣file␣and␣print␣some␣results.")
        print ("2:␣When␣two␣files␣are␣input,␣it␣will␣run␣training␣on␣the␣first␣one
␣␣␣␣␣␣␣␣␣␣␣␣and␣classify␣the␣second,␣outputing␣the␣results␣into␣a␣file␣called
␣␣␣␣␣␣␣␣␣␣␣␣output.csv.")
    elif len(sys.argv) == 2:
        training_file = sys.argv[1]

        #Creates a dict from rows in csvfile, using first row as keys
        training_csv = open( training_file )
        training_reader = csv.DictReader(training_csv)

        #Add all training hands to a list
        training_list = []
        for line in training_reader:
            training_list.append(line)

        run_k_fold_test(training_list)

    else:
        training_file = sys.argv[1]
        test_file = sys.argv[2]

        #Creates a dict from rows in csvfile, using first row as keys
        training_csv = open( training_file )
        training_reader = csv.DictReader(training_csv)

        # creates a dict for the test file as well
        test_csv = open( test_file )
        test_reader = csv.DictReader( test_csv )

        #Add all training hands to a list
        training_list = []
        for line in training_reader:
            training_list.append(line)

        # Add all testing hands to a list
        testing_list = []
        for line in test_reader:
            testing_list.append(line)

        gen_competition_submission(training_list, testing_list)


if __name__ == "__main__":
    main()
```