Done by: Sophie Sepp, Matrikel-Nr.: 03659032; Gerda Bortsova, Matrikel-Nr.: 03658412

Can Machines Learn Rules of Poker?

The Objective

The aim of the project is to come up with a good method to classify the "Poker Hand" data set[1] and, more importantly, gain a better understanding of the classification methods we learned in the course, namely, decision trees, kNN, SVM and neural networks, by applying them to the problem and trying to explain their performance. As the dataset in its original form is quite difficult to classify, we also considered several feature transformations.
We implemented our ideas in R[2].

The "Poker Hand" Dataset

In the dataset, each row encodes a "poker hand". There are 10 features corresponding to the five cards of the hand (a suit and a rank for each). E.g., (1, 1, 1, 13, 1, 12, 1, 11, 1, 10) corresponds to Ace (1), King (13), Queen (12), Jack (11) and Ten (10) of Hearts (1). The classes – 10 in total – represent the type of a hand. Refer to the *Table 1* for the details. The dataset is divided into training set (25,010 instances) and test set (1,000,000 instances) by the creators. The percentage of instances of classes in both sets roughly corresponds to the real distribution of probability of occurrence of hands.

Table 1

Class	Hand Name	Description	# of Instances	Proportion in the Dataset / Probability
0	Nothing in hand	not a recognized poker hand	12493	49.95202% / 50.117739%
1	One pair	one pair of equal ranks	10599	42.37905% / 42.256903%
2	Two pairs	two pairs of equal ranks	1206	4.82207% / 4.753902%
3	Three of a kind	three equal ranks	513	2.05118% / 2.112845%
4	Straight	five cards, sequentially ranked	93	0.37185% / 0.392465%
5	Flush	five cards with the same suit	54	0.21591% / 0.19654%
6	Full house	pair + different rank three of a kind	36	0.14394% / 0.144058%
7	Four of a kind	four equal ranks	6	0.02399% / 0.02401%
8	Straight flush	straight + flush	5	0.01999% / 0.001385%
9	Royal flush	{Ace, King, Queen, Jack, Ten} + flush	5	0.01999% / 0.000154%

One of the difficulties in classifying this dataset is that seemingly geometrically very distant samples may happen to be in the same class (e.g., (1, 2, 3, 4, 5)[3] and (5, 4, 3, 2, 1) are "straight"); while instances close to each other may be of different classes (e.g., (1, 2, 3, 4, 5) is "straight" and (1, 2, 3, 5, 5) is "one pair"). In order to understand how the dataset may look like geometrically, we visualized a simplified version of the problem: the case when we draw only three cards from the deck. On the *Fig. 1* the number of available ranks is restricted to 3 (e.g., as if we would draw cards from deck with only aces, kings and queens), on the *Fig. 2* it is 5. Blue represents "three of a kind", green is "one pair", and red is "not a hand'.

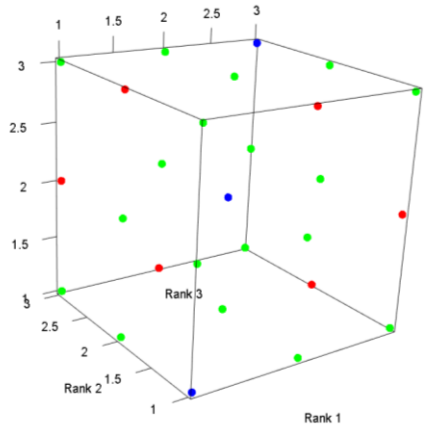


Figure 3

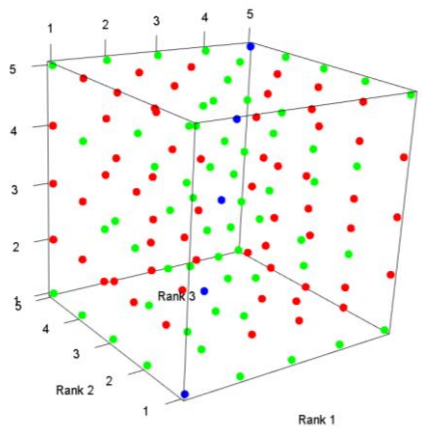


Figure 3

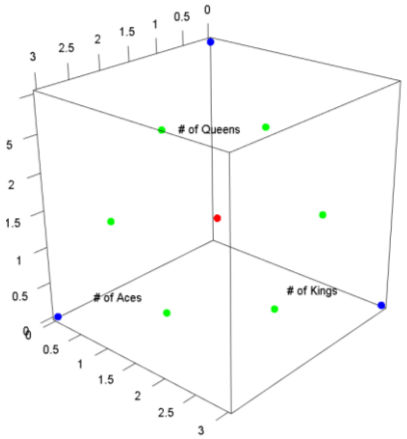


Figure 3

[1] The Poker Hand from the UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Poker+Hand
[2] The R Project for Statistical Computing: http://www.r-project.org/
[3] For brevity, we use only ranks in the examples.

Done by: Sophie Sepp, Matrikel-Nr.: 03659032; Gerda Bortsova, Matrikel-Nr.: 03658412

Transformations

While thinking of a feasible transformation, we considered a question: how would human approach the task? Human would count number of cards belonging to specific ranks and suits.

1. *"Summation" transformation*: a poker hand is transformed into a 17-dimensional vector, 1st to 13th components of which correspond to the count of cards of a respective rank; the same for suits. E.g., (1, 1, 1, 13, 1, 12, 1, 11, 1, 10) becomes (1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 5, 0, 0, 0).

A visualization of a toy case of this transformation, with 3 cards of 3 possible ranks is given on the *Fig. 3*. It appears that all samples lie on the same plane, as features' sum is always constant (3 for toy case, 5 for original problem). Furthermore, classes 0-3 form spheres, centered around a point of closest distance from origin to the plane. For each class, all samples have same Euclidean distance to the origin, which is different for each class. (For toy case, (1, 1, 1), i.e. "nothing", has distance of $\sqrt{3}$; for (2, 1, 0), i.e. "one pair", it is $\sqrt{5}$; for (3, 0, 0), i.e. "three of a kind", it is 3.)
However, in this transformation we used "domain knowledge": the fact that we should count cards. For this particular problem we can write a program for classifying the data with 100% accuracy, so applying this kind of transformation makes the problem less interesting, but we are still curious to see, how classification algorithms deal with it.
For the next transformation, the only thing we have to know to come up with it for an arbitrary problem is that features are categorical.

2. *"Expansion" transformation*: each rank and suit coded in 1-of-k fashion. E.g., suit 1 is mapped to (1, 0, 0, 0) and 4 to (0, 0, 0, 1). Ranks are mapped into 13-dimensional vectors, suits into 4-dimensional, the whole dataset becomes of dimensionality 13*5+4*5=85.

Also, it is of importance to note that instances of classes 0-3 constitute more than 99% of the dataset, and membership in these classes can be identified solely from ranks of cards in the hand. This gives us an opportunity to not consider suits while training and not include instances from classes 4-9 to the dataset. This gives a significant reduction in parameters for all algorithms we use, as number of inputs is reduced from 10 to 5 (from 17 to 13 for "summation" transformation, 85 to 65 for "expansion" transformation). SVM has to train 4 classifiers instead of 10, neural networks have 4 output neurons instead of 10.

Decision Trees and kNN

In this problem, geometrical proximity of samples does not mean membership in the same class and vice versa.
As could be seen from the visualizations, samples are very "mixed". For this reason decision trees and kNN (even with NCA, as it can only make linear transformations) are not feasible options. It is obvious, that some non-linear transformation required.
Another aspect why decision trees failed is the equality of variables. As only the number of variables is important for forming a class, it does not give any advantage in splitting the tree at variable "queen" compared to splitting at variable "ace".
Example of KNN (k=5):

true										
pred	0	1	2	3	4	5	6	7	8	9
0	302477	317814	32079	10230	2534	1756	1080	185	12	3
1	197116	103646	15451	10891	1351	240	336	45	0	0
2	1425	852	92	0	0	0	0	0	0	0
3	191	186	0	0	0	0	8	0	0	0

SVM

For our experiments with SVM, we concentrated on the most popular kernel: Gaussian. On the original data it gives rather poor results: with parameters $\sigma = 0.001, 0.01, 0.05$ and $C = 10, 100, 1000$ (we considered all combinations) it gives 60% training and 57% testing accuracy at maximum ($\sigma = 0.05$ and $C = 1000$). Taking into an account that class 0 ("nothing") constitutes almost 50% of the dataset, it is not a big improvement.
However, on the "expansion" transformation it performs quite well with some combinations of parameters. The results are summarized in the table below.

Table 2

σ	C	Training accuracy	Testing accuracy
0.05	10	100%	92%
0.1	10	100%	72%
0.05	50	100%	91%
0.1	50	100%	75%

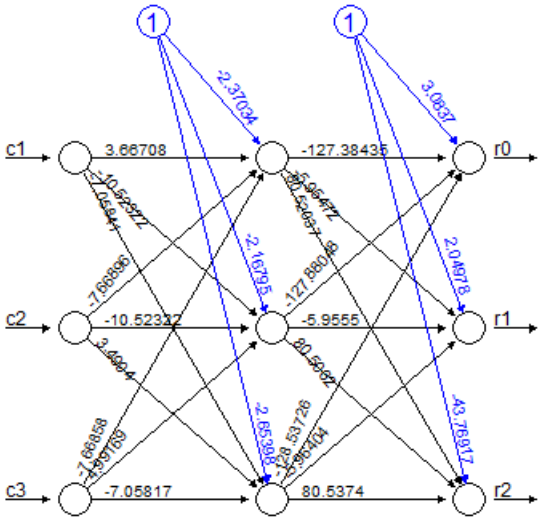
The "summation" transformation also gives very good results. It is linked to the fact that training samples from different classes form hyper-spheres centered at one point in this transformation, and with Gaussian kernel we create hyper-spherical contours. We also tried polynomial kernel, which performed slightly worse, especially on class 5 ("flush").
Performance of Gaussian kernel on the single classes:

true / pred	0	1	2	3	4	5	6	7	8	9
0	501209	0	0	0	144	333	0	0	3	0
1	0	422498	0	0	0	0	0	0	0	0
2	0	0	47622	0	0	0	84	0	0	0
3	0	0	0	21077	0	0	1096	215	0	0
4	0	0	0	0	3703	0	0	0	0	0
5	0	0	0	0	0	1663	0	0	9	0
6	0	0	0	44	0	0	244	0	0	0
7	0	0	0	0	0	0	0	15	0	0
8	0	0	0	0	24	0	0	0	0	0
9	0	0	0	0	14	0	0	0	0	3

Performance of Polynomial kernel on the single classes:

true / pred	0	1	2	3	4	5	6	7	8	9
0	501209	199	6162	0	90	1132	0	0	2	0
1	0	422251	12198	6466	0	2	1381	0	0	0
2	0	48	29262	0	0	0	0	0	0	0
3	0	0	0	14655	0	0	0	228	0	0
4	0	0	0	0	3795	4	0	0	8	0
5	0	0	0	0	0	851	0	0	2	0
6	0	0	0	44	0	0	43	0	0	0
7	0	0	0	0	0	0	0	2	0	0
8	0	0	0	0	0	6	0	0	0	0
9	0	0	0	0	0	1	0	0	0	3

Neural Networks



The toy case
In order to get some insight into how ANN work, we decided to inspect a toy case: when the hand consists of 3 cards of 3 possible ranks. As the input we are given a number of cards corresponding to each rank ("summation" transformation). On this data, we trained a simple NN with one hidden layer and three hidden units, which is shown on *Fig. 4*. This and all following networks are trained using resilient backpropagation (RPROP) with weight backtracking.
On the hidden layer, the network maps "nothing" samples, e.g., (1, 1, 1), into approximately (0, 0, 0) (i.e. very small numbers); "one pair" samples like (1, 2, 0) are transformed into (0, 0.06, 0) (or a permutation of it, depending on the order of values in the input); "three of a kind", e.g. (3, 0, 0), become (0.99, 0, 0). For each hidden unit, one incoming branch has a positive weight, and two others have negative ones of approximately equal quantity and are twice larger than that of the positive weight, which creates the aforementioned mapping. All the branches coming from hidden units to the output neuron "r0", which corresponds to the "nothing" class, have comparatively big negative weights (approx. -127), so that samples from "one pair" or "three of a kind" containing relatively large values in their hidden layer mapping (0.06 and 0.99) have significantly lower values at "r0" neuron, compared to "nothing", for which all hidden unit values are very small. At "r1" and "r2" values are computed in a similar fashion.

The real case
Training neural networks takes significant amount of time (for our dataset of 25,010 samples it was several hours on average), for which reason, in order to come up with a feasible NN configuration, we decided to evaluate performance of different configurations for variations of the toy problem. These variations are: 3 cards of 3-8 possible ranks; 4 cards of 3-4 possible ranks. For each problem we constructed a dataset of all possible combinations of cards (the class distribution is approximately the same, as in original problem) and, using it, trained neural networks with one hidden layer with 3-13 units

and with two hidden layers with 3-8 neurons on each layer (we tried all combinations). The table below shows network configurations for one- and two-layered cases which have least number of parameters, but still have satisfactory accuracy (0.8 training correctness).

Table 3

# of cards	# of ranks	One-layered neural network			Two-layered neural network			
		Hidden layer	**Parameters**	Correctness	Hidden layer 1	Hidden layer 2	**Parameters**	Correctness
3	3	4	**36**	1	4	3	**47**	1
3	4	8	**68**	0.92	5	3	**54**	0.84
3	5	8	**69**	0.99	6	4	**72**	1
3	6	9	**76**	0.87	6	4	**72**	1
3	7	9	**76**	0.86	6	4	**72**	1
3	8	8	**68**	0.82	6	4	**72**	1
4	3	10	**94**	0.92	6	5	**89**	1
4	4	>13	**>121**		7	5	**99**	0.82

From this results, we concluded that it is better choose two-layered network with number of neurons slightly greater on the first layer than on the second. This is because the number of parameters in one-layered case seem to grow faster with increase of number of inputs. For the last problem, with 4 cards of 4 ranks, neural networks with number of hidden units below 14 didn't give satisfactory results, although number of parameters was significantly larger (121), then if using two-layered architecture (99). For the two-layered case, alternative configurations with more neurons on the second layer than on the first could also be valid, but they need more parameters. Two-layered architecture was also chosen from an assumption that deeper networks can learn more complex rules.
For training, as well as for SVM, in order to reduce number of parameters, only ranks are used as inputs, and only instances of classes 0-3 were preserved.
With original data, we tried several configurations: (13, 5), (10, 10), (15, 10), (17, 13). The (17, 13) network appeared to give quite good results: training accuracy of 83% and testing accuracy of 82%. Below is the confusion matrix for the testing set.

true / pred	0	1	2	3	4	5	6	7	8	9
0	498449	111813	163	83	3799	1986	7	5	12	3
1	2760	310134	39864	12805	86	10	179	31	0	0
2	0	61	5945	188	0	0	652	8	0	0
3	0	490	1650	8045	0	0	586	186	0	0

Classification of dataset transformations
For "expansion" transformation, training accuracy 92% and testing accuracy 89% was achieved with neural network with two hidden layers with 13 and 5 neurons on them. The number 13 was chosen, as, in fact, all information necessary for classification of this 65-dimensional dataset (13 ranks * 5 cards) can be boiled down to 13-dimensional vector containing counts of cards of each rank ("summation" transformation). However, the neural network trained not necessarily does specifically this. The number of neurons on the second layer is identified by tuning: more than five neurons do not give a significant reduce in error, and less give less accuracy. For instance, a network with 4 neurons on the second layer, although it gives the same testing error of approximately 11%, misclassifies the majority of samples from the class 2, which tells about poorer generalization abilities.
Here is the confusion matrix of the test set:

true / pred	0	1	2	3	4	5	6	7	8	9
0	474009	32405	215	29	3815	1876	0	0	12	3
1	27200	385881	11339	4774	70	120	29	0	0	0
2	0	4212	36068	16318	0	0	1395	230	0	0

It can be seen that the majority samples from classes 0 ("nothing"), 1 ("one pair") and 2 ("two pairs") is classified correctly, although the rest of the classes are never identified.