Poker Hand Dataset: A Machine Learning Analysis and a Practical Linear Transformation

Walinton Cambronero
College of Computing, Georgia Institute of Technology
wcambronero3 {at} gatech.edu

ABSTRACT

The Poker Hand dataset [1] has two properties that makes it particular challenging for classification algorithms: it contains only categorical features (suite and rank of a card) and it's extremely imbalanced (2 out of 10 classes constitute 90% of the samples). This makes it an interesting dataset for studying and evaluating various of the well-known Machine Learning (ML) classification algorithms. As part of Georgia Tech's Machine Learning CS 7614[1] class, some of these algorithms are evaluated against this dataset. This paper describes the methodology used to create classifiers than can classify a 5-cards poker hand entirely based in Machine Learning as opposed to classical rule-based programming. The results obtained for each of the following algorithms are discussed in detail: Multi-layer Perceptron Neural Network, Support Vector Machines, Decision Trees and K-Nearest Neighbors. Additionally, a simple linear transformation for the dataset is proposed, which significantly improves the performance of the classifiers.

1 INTRODUCTION

Machine Learning classifier algorithms struggle with categorical features because typical distance (a.k.a. similarity) metrics can't be naturally calculated for such features. Categorial features need first to be encoded in a real-valued format before distance metrics can be even calculated, but even with real-valued numbers, typical metrics such as the Euclidean distance may not make sense for such features. E.g., if we have "card suite" encoded in real-valued numbers, what does the Euclidean distance of hearts to spades mean? This problem has been studied for a while. Several authors such as Boriah et.al. [2] have developed comparative models to evaluate the performance of some of the proposed methods. Another kind of challenging datasets for classification algorithms are those that are imbalanced, i.e. there is disproportionate ratio of samples of each class. "*Imbalanced classifications pose a challenge for predictive modeling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class*"[2]. The Poker-hand dataset [1] has both properties: it's extremely imbalanced and its features are categorical. A detailed description of the dataset is provided in the next section. This particular dataset is described by the authors as "*challenging dataset for classification algorithms*"[3].

This paper is written for the Machine Learning class CS 7614, from the Master of Computer Science program at Georgia Tech's Computing Systems School. The goal of the paper is not to solve the problems faced by the classifier algorithms when dealing with this kind of datasets, but to provide a comprehensive analysis of the results achieved with various popular classifiers, the methodology used to reach such results and the challenges faced along the way.

The classifiers covered are: Multi-layer Perceptron Neural Network (**MLP**), Decision Trees (**DT**), K-Nearest Neighbors (**KNN**) and Support Vector Machines (**SVM**). For each classifier, I first show how is the data pre-processed and the methodology followed to pick the algorithm's hyper-parameters. Additional considerations on a per-algorithm basis are discussed along with a Model Complexity[4] analysis. Finally, the results obtained are analyzed using various visual and tabular reports. The reader is assumed to be comfortable with the basic Machine Learning theory and to have a good understanding of the algorithms under study. The paper does not attempt to elaborate on these topics, instead, it focuses in the analysis of the obtained results from such algorithms.

In addition, a novel linear transformation is proposed for the dataset. The transformation makes the dataset more suitable for processing by the different Machine Learning algorithms. The results achieved by the classifiers when using both the transformed and original data are discussed in the paper.

Python is used as the programming language, Numpy[5] is used for data processing and Scikit-learn[6] is the Machine Learning library of choice. Calculations are run in both a PC without GPU support, and in Google's Colab[7] with GPU support.

2 DATASET DESCRIPTION

The dataset is divided in training and testing sets. There are 1M and 25K samples in each set, respectively. This is a 11-dimensional dataset: 10 attributes and 1 label (a.k.a. class or feature). All attributes are categorical. There are no missing values. Each sample represents a 5-cards poker-hand. Each card has two attributes (a.k.a. features): suite and rank.

Encoding

Suite: **1**: Hearts, **2**: Spades, **3**: Diamonds, **4**: Clubs
Rank: **1**: Ace, **2**:2, ..., **10**: Ten, **11**: Jack, **12**: Queen, **13**: King
Label: **0**: Nothing, **1:** Pair, **2**: Two pairs, **3**: Three of a kind, **4**: Straight, **5**: Flush, **6**: Full house, **7**: Four of a kind **8**: Straight Flush **9**: Royal Flush

[1] https://www.cc.gatech.edu/~isbell/classes/2009/cs7641_spring/
[2] https://machinelearningmastery.com/what-is-imbalanced-classification/
[3] https://archive.ics.uci.edu/ml/machine-learning-databases/poker/poker-hand.names
[4] Model Complexity refers to the number of terms (variables) needed in a particular model
[5] https://numpy.org/
[6] https://scikit-learn.org/
[7] https://colab.research.google.com/

Class Distribution

The dataset is very imbalanced. There are two dominant classes: 0 (Nothing in hand) and 1 (One pair). This distribution isn't random. It follows the actual distribution in the true game domain. The dominant classes account for over 90% of the samples. **Table 1** shows the class distribution.

Table 1

0: Nothing in hand: 49.95202%
1: One pair, 42.37905%
2: Two pairs, 4.82207%
3: Three of a kind, 2.05118%
4: Straight, 0.37185%
5: Flush, 54 instances, 0.21591%
6: Full house, 36 instances, 0.14394%
7: Four of a kind, 0.02399%
8: Straight flush, 0.01999%
9: Royal flush, 0.01999%

Credit: https://archive.ics.uci.edu/ml/datasets/Poker+Hand

3 HOW ARE THE RESULTS MEASURED

Classification Reports[8] are generally used in Machine Learning to measure the performance of classification algorithms. A Classification Report analyzes correct vs incorrect predictions and produces a series of metrics. From these metrics, the **macro F1 score** provides unweighted results per class, i.e. it does not take imbalance into account. "*In problems where infrequent classes are nonetheless important, macro-averaging may be a means of highlighting their performance*"[9]. This prevents the good results obtained in dominant classes -alone- to be treated as a good result overall. E.g., for a classifier that correctly classifies 100% the 2 dominant classes but incorrectly classifies 100% of the other classes, a weighted metric would find that 90% of the results were correct, given that the 2 dominant classes represent 90% of the data, but clearly an algorithm that miss-classifies 8 out of 10 poker hands is a bad one In addition to the macro F1 score, a **Confusion Matrix**[10] is used to visualize analyze the results of the classifiers. The results of this paper are discussed in function of both the macro F1-score and the Confusion Matrix. Where appropriate, classifier **Training Time** is also measured.

4 TRAINING, VALIDATION AND TESTING SETS

The classifiers don't have access to the **testing-set** during learning. The testing-set is exclusively used for post-learning evaluation of a classifier. During learning, some algorithms require a **validation dataset** to tune hyper-parameters (e.g. for Cross-Validation) or as input to a model fitness function. The validation-set is also used for Model Complexity analysis. For this purpose, the **training-set is split into training and validation** (80/20). Before the split, the **data is shuffled and stratified**. Stratification is made as a function of the label, to make sure that both sets proportionally receive labels from all classes.

5 MULTI-LAYER PERCEPTRON NEURAL NETWORK

The first algorithm to evaluate is a as Multi-layer perceptron[11] (MLP). MLP is a type of feedforward Artificial Neural Networks (ANN). The choice of initial configuration and other hyper-parameters, as well as the results obtained after further tuning these values is described in this section.

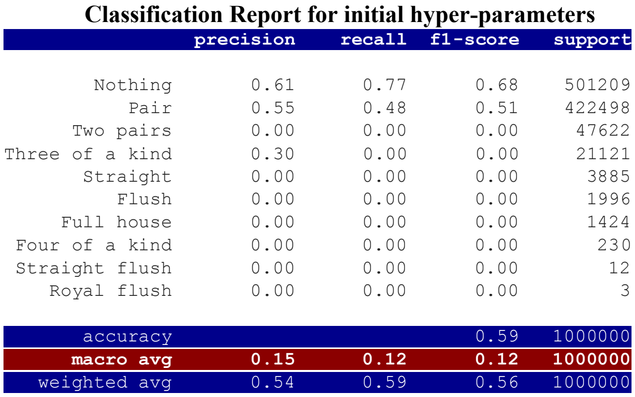
Data pre-processing

Per SciKit-learn documentation, "*Multi-layer Perceptron is sensitive to feature scaling, so it is highly recommended to scale your data*"[12]. The data is scaled with Scikit-learn's StandardScalar[13] using the recommended range [0, 1].

Initial hyper-parameters

Mitchell suggests [3] that a network of 3 layers (1 output and 2 hidden layers) can be used to model any arbitrary function. And according to Heaton [4], a rule of thumb to choose the number of neurons per hidden-layer is to pick a value in that is "*between the size of the input layer and the size of the output layer*" [4]. The input layer for this dataset has 10 features and the output is the 10 possible classes (poker hands). Scikit-learn's documentation mentions that *"for relatively large datasets, Adam [solver] is very robust"*[14]. Following these recommendations, **a network of 2 hidden layers of 10 neurons** each is used with the **Adam solver**. **ReLU** as the activation function. The other hyper parameters are Scikit-learn's defaults. The classification report for these results are shown in **Table 2**. Most classes were not classified at all (as noticed in the 0.00 recall values). During training, the 200 max-iterations limits was hit on every epoch, this means the algorithm wasn't learning. The results are clearly disappointing.

Table 2



Classification Report for initial hyper-parameters

	precision	recall	f1-score	support
Nothing	0.61	0.77	0.68	501209
Pair	0.55	0.48	0.51	422498
Two pairs	0.00	0.00	0.00	47622
Three of a kind	0.30	0.00	0.00	21121
Straight	0.00	0.00	0.00	3885
Flush	0.00	0.00	0.00	1996
Full house	0.00	0.00	0.00	1424
Four of a kind	0.00	0.00	0.00	230
Straight flush	0.00	0.00	0.00	12
Royal flush	0.00	0.00	0.00	3
accuracy			0.59	1000000
macro avg	**0.15**	**0.12**	**0.12**	**1000000**
weighted avg	0.54	0.59	0.56	1000000

Figure 1 compares the accuracy obtained for each class as a function of the number of samples of that class (a.k.a. "support"

[8] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
[9] https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics
[10] https://en.wikipedia.org/wiki/Confusion_matrix
[11] http://deeplearning.net/tutorial/mlp.html
[12] https://scikit-learn.org/stable/modules/neural_networks_supervised.html#tips-on-practical-use
[13] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
[14] https://scikit-learn.org/stable/modules/neural_networks_supervised.html#tips-on-practical-use

in the Classification Report). The graph shows that the classifier is biased towards the dominant classes. Only the most-dominant class is correctly classified better than chance.

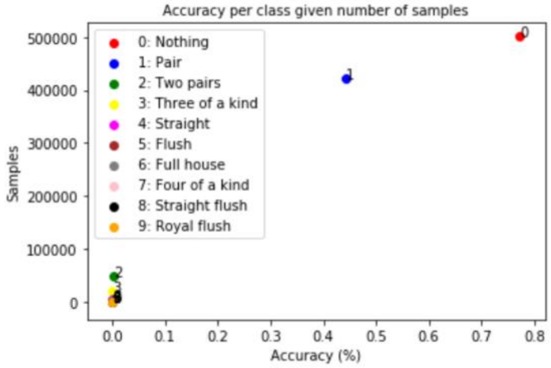


Figure 1

The **Confusion Matrix** is shown in **Figure 2**. It shows that the least dominant-classes are completely miss-classified as one of the dominant-classes. The dominant classes are 0 and 1. It can be observed in the Confusion Matrix that all predictions (squares with color) ended up in either the 0 or 1 column.

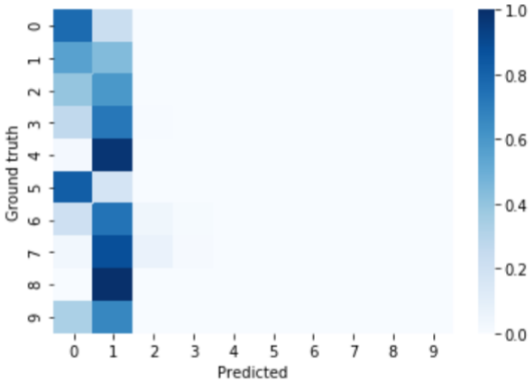


Figure 2

Model complexity analysis

The **F1-score** is analyzed after running the classifier with various settings of **topology** (number of hidden layers and neurons), **alpha** (regularization parameter) and **learning rate**. Varying the **topology** can show how a more or less complex network can better represent the function, and **alpha** helps "*avoiding overfitting by penalizing weights with large magnitudes*"[15]. The analysis is done exclusively against the training data, which is split 80/20 as train-validation. The values analyzed for topology are one layer with 100 neurons, 2 layers with 10 neurons, 2 layers with 100 neurons and 3 layers with 100 neurons. The values analyzed for alpha is in the range recommended in Scikit-learn's documentation [1e-1, 1e-6]. A properly tuned learning rate can help the model converge. The results are shown in **Figure 3.** The results suggest that tuning hyper-parameters in isolation is not going to help, e.g. arbitrarily choosing network topologies without proper tuning of other parameters isn't improving the results. The two most complex networks resulted in the highest level of overfit. Overfit is observed when the results over the training-set (blue) are much better than their testing-set (green) counterpart. The third most complex (10, 10) was largely defeated by the simplest one (100). The results also suggest that for small values of alpha, the score remains stable, but these parameters should be validated in conjunction with other hyper-parameters.

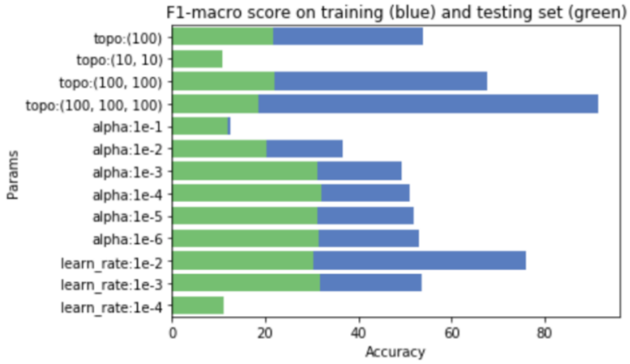


Figure 3

Model analysis and comparison with grid-search

Manually choosing the params from the previous section that provided the best results, did not significantly improve the performance. E.g. with 2-layer **topology (100, 100)**, **alpha = 1e-4** and **learning rate = 1e-3,** resulted in a **25% F1 score** (vs 12% from previous exercise)**.** In an effort to study how the results change when multiple hyper-parameters are tested in conjunction, a grid search was started with a wider range of values including **tolerance** and **maximum iterations**. The grid came back with the following hyper-parameters and an impressive **80% F1 score**: 3 hidden layers of 100 neurons each, alpha=0.0001 and learning rate=0.01 with 100 max-iterations. The result of this experiment was reproduced multiple times to validate its consistency. The Confusion Matrix is shown in **Figure 4**.

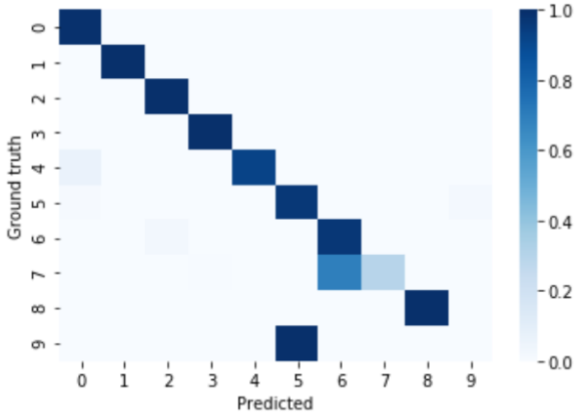


Figure 4

This more complex network when tuned along other parameters provided a remarkable increase in performance. In this particular example, class 9 was completely miss-classified. The reason is that class 9 (royal flush) has only 5 training samples out of 25K total in the dataset, and the generalization that the algorithm could achieve is not enough to correctly classify that label. Notice that, while proportionally speaking, the vast

[15] https://scikit-learn.org/stable/modules/neural_networks_supervised.html#regularization

majority of samples were classified correctly (there is over a million samples in the testing set), and only a few, from the non-dominant classes were miss-classified, yet the macro F1-score is still reporting only 80% success. This confirms that the F1-score is an appropriate metric for this dataset.

Data transformation for a simpler network topology

The results obtained in the previous section aren't great. Theoretically these results can be improved as a neural network is capable of modeling any arbitrary function, but this might require a more complex model. This section proposes a linear transformation to the dataset that provides better results even for the simpler topologies that previously resulted in bad predictions. The transformation is based in the fact that the order in which the cards appear (in a hand) doesn't matter (to classify the hand), and that a more important attribute for classifying a hand is the number of cards (i.e. cardinality) with the same rank or suite that appear in the hand. The original dataset model gives an artificial importance to the order in which the cards appear (samples are ordered lists of 5 cards) and it does not explicitly encode the cardinality of each suite or rank. The premise is that by making this attribute explicitly available in the data, a Neural Network is able to better classify the dataset, in comparison to the same Neural Network when using the original model in which the attribute is hidden. To validate this premise, the neural-network model that provided bad results when using the original dataset is trained again with the new dataset. The results are discussed in this section.

Linear transformation

The following is a linear transformation from the original 11D space to a new 18D space. A linear transformation is preferable due to its reduced computational requirements. The new dimensions and descriptions are:

Attributes 1 through 13: The 13 ranks, i.e. 1: Ace, 2: Two, 3: Three, …, 10: Ten, 11: Jack, 12: Queen, 13: King.
Attributes 14 through 17: The 4 suites, i.e. 14: Hearts, 15: Spades, 16: Diamonds, 17: Clubs
Domain: [0-5]. Each dimension represents the rank or suite cardinality in the hand.
Last dimension: Poker hand [0-9] (unchanged).

Example transformation for the Royal Flush of Hearts

Representation in original dimensions (11D)

Data: 1,1,1,10,1,11,1,12,1,13,9
Encodes: Hearts-Ace, Hearts-Ten, Hearts-Jack, Hearts-Queen, Hearts-King, Royal-Flush

Representation in new dimensions (18D)

Data: 1,0,0,0,0,0,0,0,0,1,1,1,1,5,0,0,0,9
Encodes: 1st column = 1 ace, 10th through 13th columns = 10, Jack, Queen and King, 14th column = 5 cards are hearts, and 18th column a Royal Flush.

Figure 5 shows a visual representation of the transformation for the Royal Flush of Hearts.

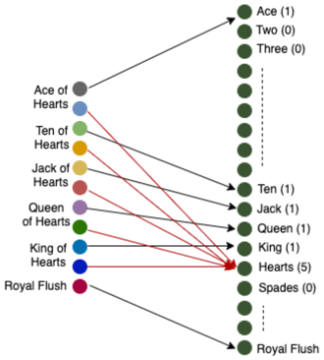


Figure 5

The new model represents any given a combination of 5 cards the same way regardless of order and explicitly exposes information useful for Poker hands such as the cardinality of each rank and suite.

Results with the transformed data

A new grid search was started but limiting the topology to the two that previously provided very poor results (**under 15% macro-avg F1 score**). The result was that a network of a single layer with 100 neurons resulted in **72% macro-avg F1** score, i.e., a simple data transformation allowed for a significant performance increase using a less complex neural network. **Figure 5** shows the Confusion Matrix. Notice that even the least dominant class (class 9) was correctly classified 100% of the time.

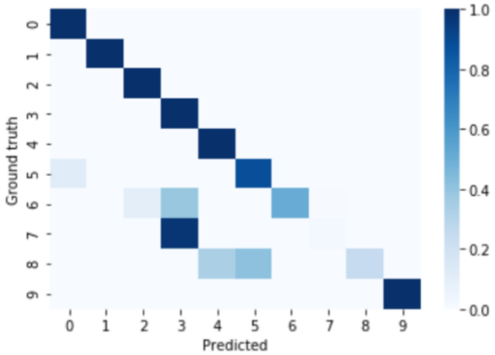


Figure 6

A 2-layer (100 neurons each) MLP results in ~86 accuracy (macro-avg F1 score). **Figure 7** shows the Confusion Matrix.

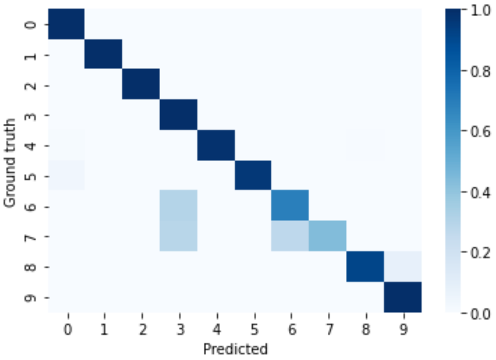


Figure 7

The result shown in Figure **7** using 2-layers is equivalent or better than the result achieved a 3-layers MLP with the original data (refer to previous section). In terms of **training-time**, the transformed dataset completes training in only 13 seconds while it takes 20+ seconds for the original data, an improvement of ~40% in training-time.

This is an example of a method in which, a highly imbalanced and purely categorical dataset can be still successfully processed by a classification algorithm. This method is of my own invention and applicable only to this dataset. As opposed to using a similarity metric that is applicable to categorial features, this method transforms the data in a way that it becomes non-categorical. More general approaches and similarity metrics are studied in [2].

General results

An MLP neural network with 3 hidden layers of 100 neurons each, alpha=0.0001 and a learning rate=0.01, achieved a ~80% F1-macro average score. This is a remarkable improvement over the initial hyper-parameters that before proper tuning yielded a 12% score. The testing set has over 1M samples of which over 95% were classified correctly but given that the metric in use (macro F1-score) does not give more weight to the dominant classes, the overall score is significantly lower. This proves that the F1 macro-average metric is appropriate for this dataset.

In addition, it was observed that the original dataset model is not the most appropriate for the classification task at hand. In order to learn the underlying classification function, the neural network needs to learn some hidden attributes. A linear transformation over the dataset is proposed, which makes some of these attributes explicitly available in the data. The end result is a simpler model to learn and hence, a simpler neural network is able to achieve comparable results to the more complex network that uses the original dataset.

6 DECISION TREES

Pre-analysis of the dataset

In order to classify a Poker hand, a player needs all 5 cards revealed. A single one card can totally re-classify a hand. E.g. the first 4 cards can be classified as class nothing, but the fifth card can make the hand become a flush, pair, straight and others. This is a particular hard problem for a Decision Tree (DT). There will be splits that miss-classifies a whole bunch of hands. A probabilistic result seems to be more appropriate, e.g. having 3 given cards reduces the domain of possible hand. On decision-splits, I expect that the dominant classes will be chosen more often simply because they get more votes.

Initial hyper-parameters analysis

SciKit's DecisionTreeClassifier is used. The most interesting default settings (in parenthesis) are: **max_depth** (no-limit), **min_samples_split** (2), **min_samples_leaf** (1), **class_weight** (uniform) and the **split method** (Gini). The DT is expected to overfit if the max_depth is set to no-limit. On one hand, the DT to should be able to generalize (e.g. smaller max_depth), but in the other hand it must be able to go deep-enough to classify the 5-cards correctly. Given the number of samples for some of the non-dominant classes (have 6 or fewer instances), the min_samples_leaf and min_samples_split can't be too strict, otherwise there is no hope for the DT to learn those classes (e.g. min_sample_leaf to be smaller than samples exist). The class_weight should be set to take in account the imbalance nature of this dataset. Otherwise, the non-dominant classes won't have a chance. A DT is definitely not a good classifier for this particular dataset. The experiments below confirm this.

The following are the initial parameters chosen for this classifier: **max_depth**: 10 (same as used the number of attributes), **class_weight** balanced (class imbalance in dataset), **min_samples_leaf** and **min_samples_split**: 3 (less than the minimum number of samples for the least represented class).

Performance of initial parameters vs default settings

When the tree is trained using the default parameters and tested against the training-set, it obtains a perfect F1-score (1.0)**. This is due to overfitting.** When tested against the testing-set, the performance is dramatically decreased. **Figure 8** and **Figure 9** are the Confusion Matrices of the default and initial parameters, respectively, against the testing-set.

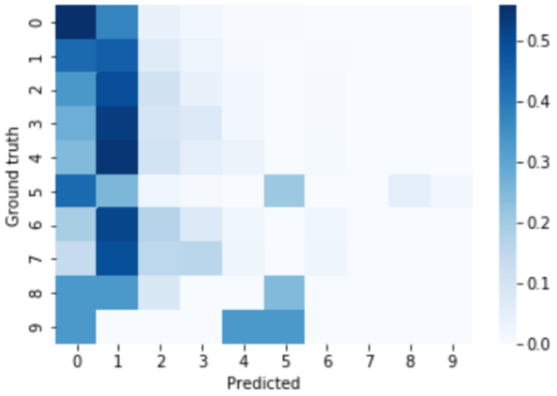


Figure 8

Figure 8 shows the results using the default settings. It is observed that the poker-hands are often miss-classified as one of the dominant classes (0 & 1) as indicated by the strong color in those columns. This is expected given that no class-balancing is configured in this test.

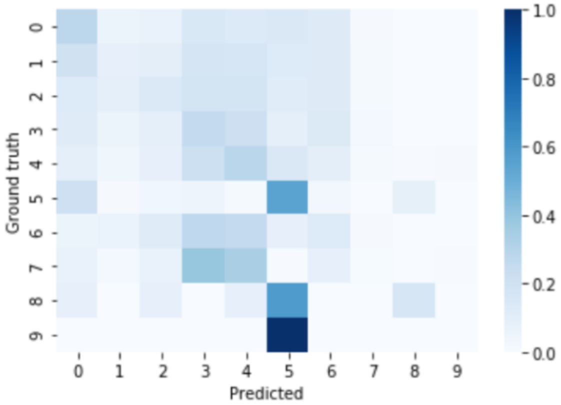
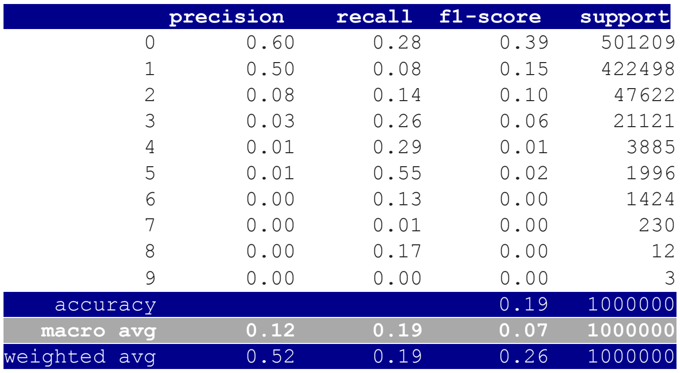


Figure 9

Figure 9 shows that the tree isn't longer biased towards the dominant classes when using the initial parameters. In **Table 3** it can be seen that recall percentage is spread across multiple classes.

Table 3



	precision	recall	f1-score	support
0	0.60	0.28	0.39	501209
1	0.50	0.08	0.15	422498
2	0.08	0.14	0.10	47622
3	0.03	0.26	0.06	21121
4	0.01	0.29	0.01	3885
5	0.01	0.55	0.02	1996
6	0.00	0.13	0.00	1424
7	0.00	0.01	0.00	230
8	0.00	0.17	0.00	12
9	0.00	0.00	0.00	3
accuracy			0.19	1000000
macro avg	0.12	0.19	0.07	1000000
weighted avg	0.52	0.19	0.26	1000000

Model complexity analysis

For the **min_samples_leaf** and **min_samples_split**, besides the default values, a choice of 4 and 10 are made. These corresponds to a number lower than the number of training samples for the least represented class (5 samples) and a higher one (10). For the **max_depth**, the default value is used (no-limit) as well as a value that is lower and exactly the number of attributes (5 and 10) and an arbitrary larger number (25). For comparison, results for the **default and the initially proposed settings are also shown**.

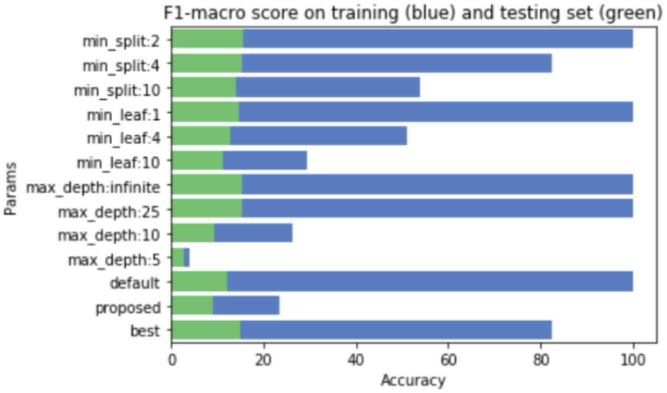


Figure 10

Figure 10 shows the F1-macro score for train vs validation sets for the different models. As expected, the validation set score is not good, since it is not expected that a decision tree classifier can generalize well on this dataset. For the 3 cases of the default values, the score against the training set is 100% (overfitting). No performance increase is observed in neither configuration. The validation set score decreases as the default settings are changed. This indicates that the tree is unable to generalize for the classes with fewest samples, and as the tree is less overfit it can't longer classify correctly the dominant classes either. These account for the 90% of samples, hence reducing the overall score. The last row (best) was added after seeing the results from all previous settings

Learning curve analysis

The learning curve analysis permits to observe how (and if) the classifier improves its score, as more training samples are used for learning. A good classifier would show that the cross-validation score improves along with the training size as the classifier is able to generalize better. The choice of parameters for the test is hard, as there is no direct evidence of a better result from the previous section. The default settings do have the best result in terms of training and test validation, but they also have a perfect F1 score against the training-set, which is an indicator overfitting. For this reason, **min_split=4** and **max_depth**=**25** (prevents overfitting) was chosen. The training-set is split in 5 different subsets (train and validation sets for cross-validation) with sizes of 10%, 32.5%, 55%, 77.5% and 100%. The folds are stratified using **StratifiedKFold**[16], this ensures that each fold gets at least one sample from each class. **Figure 11** confirms our previous findings. The score against the training-dataset (red-line) is very good (a sign of overfit) but the score against the testing-set is bad (a sign that the classifier is unable to predict from this dataset). Both scores remain stable as more training-samples are used, a sign that the DT classifier is unable to generalize from this dataset.

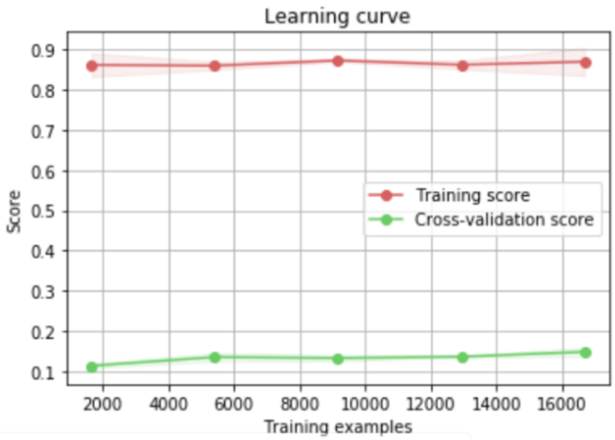


Figure 11

Results with the transformed dataset

In the previous section, a linear transformation was applied to the dataset which made it more fit for an MLP classifier. It is expected that the DT classifier performs better with this dataset as well. A quick test was performed using this dataset. A hyper-parameters grid-search was started against both the original and the transformed dataset. The parameters found by the search for the transformed data produced a classifier that reaches a **69% F1 macro score**, vs a **13% score** obtained by the classifier that was trained using the original data. This is an indication that the transformation fulfills its purpose of being a better machine representation of the data. It can be better classified by the decision tree because the count of a given symbol now matters more than the order on which the symbols appear to form a hand. A split decision can be made on a single attribute that says how many times a given card or suite appears in the sample, as opposed to depending on the same value (card rank) to appear in multiple attributes of the same sample. No further tests were

[16] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

performed as it is outside the scope of this paper. **Figure 12** shows the results obtained the DT classifier trained using the transformed dataset.

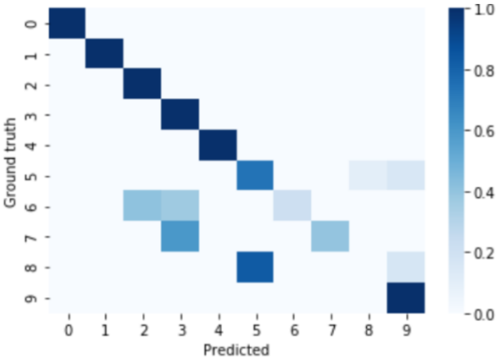


Figure 12

General results

It is shown in this section that a DT is not an appropriate classifier for this dataset. It struggles big time with the dominant classes as a DT is forced to make decision splits that end up miss-classifying entire sets of the non-dominant classes. When trained with the transformed dataset, which is more machine-learning friendly, a significant improvement in accuracy is obtained. An increase from 13% to 69% is achieved by simply feeding the same classifier with the transformed dataset.

7 SUPPORT VECTOR MACHINES

Notes and recommendations about SVMs

The following are some recommendations[17] from Scikit-learn's documentation about working with their SVM implementation: Scale the data as SVMs aren't scale-invariant. In SVC with imbalanced classes set class_weight=balanced and/or try different C parameters. Search C and gamma for values spaced exponentially far apart. For large datasets (tens of thousands) consider using LinearSVC.

Linear vs non-linear quick test

Following the advice, considered using **LinearSVC** for large datasets and compared it with **RBF** (not a linear kernel). Both using the default settings for multi-class problems. As expected, the **linear kernel** yielded awful results (**7% F1 score**) as the function isn't linearly separable. The **RBF** kernel with default setting's result was **35% F1 score**.

Choosing the initial hyper-parameters and kernels

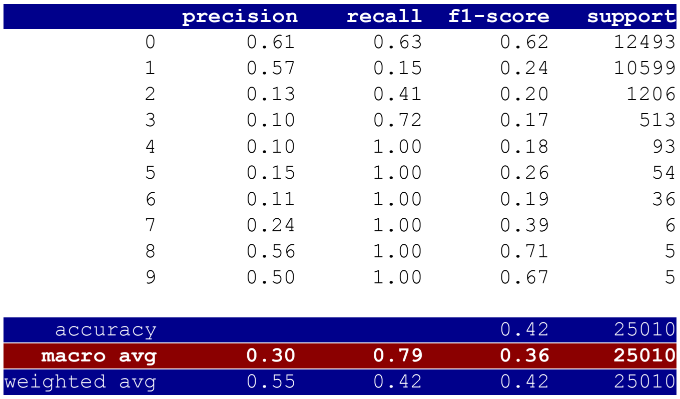
Two pre-set values of **gamma** are possible: **auto** and **scale**. The **scale** value produces higher gamma values when the variance of the training set is higher, and lower gamma otherwise. When variance is 1.0, scale and auto is the same (see formula[18]), and since the data is normalized (as per recommendation) for the SVM, the default **auto** value will be used. Lower gamma causes that single training samples have more influence, and vice versa[19]. **C** is a tradeoff in between how many training examples are correctly classified vs how relaxed the decision boundary is, e.g. a large C causes that more training examples are correctly classified at the cost of risking overfit, since the decision boundary is too close to the training boundary. The default value is **C** is used (**1.0**). Two kernels are tested: **RBF** and **Polynomial**. The **class weighting** method is being set to '**balanced'** to account for class imbalance in the dataset. Both kernels use one-vs-one scheme when the problem is multi-class.

Comparing RBF and Polynomial kernels

F1 score for the RBF and Polynomial kernels against the training set. **RBF** performed better with **36%** vs **16% F1 score**.

Table 4 shows results exclusively for the RBF kernel.

Table 4



	precision	recall	f1-score	support
0	0.61	0.63	0.62	12493
1	0.57	0.15	0.24	10599
2	0.13	0.41	0.20	1206
3	0.10	0.72	0.17	513
4	0.10	1.00	0.18	93
5	0.15	1.00	0.26	54
6	0.11	1.00	0.19	36
7	0.24	1.00	0.39	6
8	0.56	1.00	0.71	5
9	0.50	1.00	0.67	5
accuracy			0.42	25010
macro avg	**0.30**	**0.79**	**0.36**	**25010**
weighted avg	0.55	0.42	0.42	25010

The classification report in **Table 4** shows that all classes are called significantly. The classifiers aren't biased to a particular dominant-class, as has happened with other classifiers. Classes 4-9 actually have a perfect recall, which means that all samples from those classes were classified correctly. The non-perfect precision indicate that the other classes were incorrectly miss-classified as one of these.

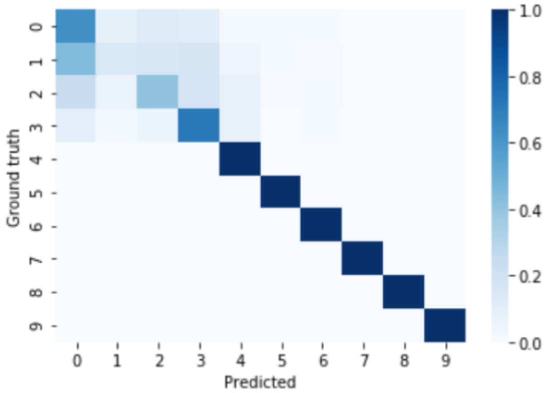


Figure 13

In **Figure 13** it can be seen that the classes with fewest samples (4-9) are all classified perfectly and the miss-classification happens in the classes with more than 100 samples. The way that SVM for multi-class works explains this. Basically, individual classifiers are created for each class that is fit for its

[17] https://scikit-learn.org/stable/modules/svm.html#tips-on-practical-use
[18] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
[19] https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

samples, where each classifier calculates its own decision boundary function with those samples. It is not surprising that the resulting function for the classes with fewest samples is less complex and probably overfits more than the decision function of the other classes where more samples allow for better generalization. Keep in mind that this report was generated using the training-set.

Model complexity analysis

According to Scikit's documentation for C and gamma, "*a logarithmic grid from 1e-3 to 1e3 is usually sufficient*"[20]. C and gamma will be analyzed with values in the recommended range **[1e-3, 1e3]**. Model complexity uses exclusively the training-set which is split 80/20 to generate a validation set. The same split is used for all tests of the hyper-parameters, so the results using the different parameters are comparable.

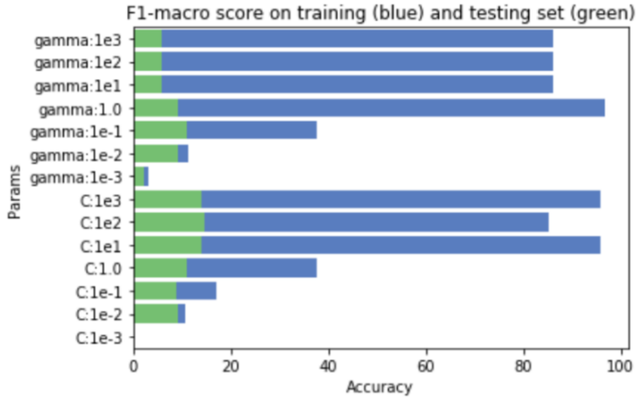


Figure 14

The suspicious of overfitting is supported by the results shown in **Figure 14**. The good performance on the training set is not matched by the performance on the validation set. Gamma controls how much "influence" each single sample has over the final function. With gamma=0.1 the training score was the lowest and the validation score was highest. For smaller values of gamma, both the train and validation scores decrease, as an indication that not enough "influence" from the vectors is being used to model the actual function shape. The good training scores for larger values of C are expected as larger C values tend to better classify the training samples. C=100 is a good tradeoff, with the highest validation score without 100% accuracy on the training set, a common symptom of overfitting.

Learning curve for RBF kernel with class balancing

The best hyper parameters found with model analysis correspond to C=100 and gamma=0.1. In addition, the class's weights are balanced, and the kernel chosen is RBF. **Figure 15** shows the expected trend: validation score grows along with the training size while the training score decreases to account for better generalization. With the training-dataset which is already a large one (25K samples), there was not enough time to see how the graph converges. The Learning curve is not calculated for the testing set, as its size (1M samples) makes it impractical.

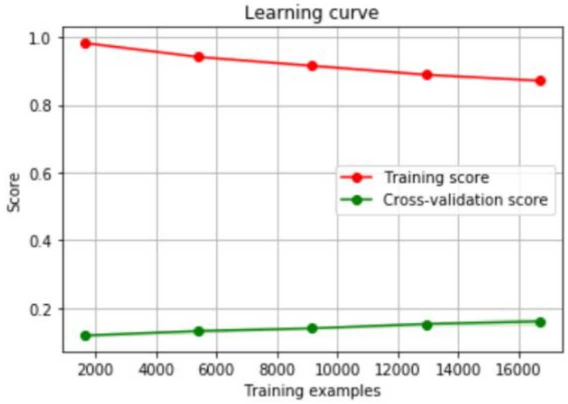
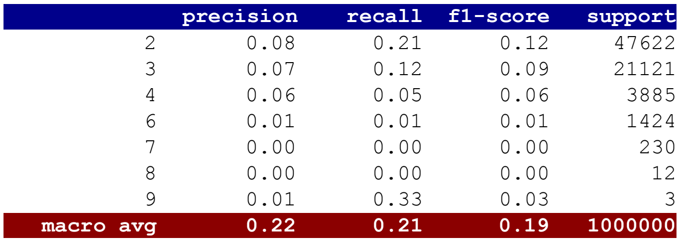


Figure 15

Report against testing set

This is the only classifier for which running time has been measured. Every other classifiers completed in less than 5 minutes (with less than a minute for most cases) when using GPU support in Google Colab. The SVMs are known to be very computationally involved, especially with a large space. Training with 25K instances and testing against 1M instances took 18 minutes to complete. The Classification Report is shown in **Table 5**. Showing the classes with the lowest recall scores.

Table 5



	precision	recall	f1-score	support
2	0.08	0.21	0.12	47622
3	0.07	0.12	0.09	21121
4	0.06	0.05	0.06	3885
6	0.01	0.01	0.01	1424
7	0.00	0.00	0.00	230
8	0.00	0.00	0.00	12
9	0.01	0.33	0.03	3
macro avg	0.22	0.21	0.19	1000000

A terrible **19% F1 macro-avg** was obtained, some classes (including one with over 3K instances) with 0% recall.

Grid Search

A grid search was started for RBF kernel with varying gamma and C values, and a Polynomial kernel with varying degree, gamma and C. In both cases the best result came back with C=1.0 and gamma = 0.1. **The results were still under 20% with many classes left un-classified.** It appears that these classes aren't separable, at least under the linear, SVC and RBF kernels.

General results

During training and doing cross-validation, the results showed high levels of overfitting: 80% in most configurations and **100% F1-score** in some of them. When using the testing-set, it was found that the classes from this dataset are not linearly separable In addition, non-linear kernels were used. **Linear, SVC** and **RBF** kernels were analyzed. The best result obtained was with the RBF kernel with just a 19% F1-macro accuracy. This result confirms that the SVM overfits and is unable to generalize for this particular dataset, due to the difficulty in separating the classes using linear and non-linear kernels.

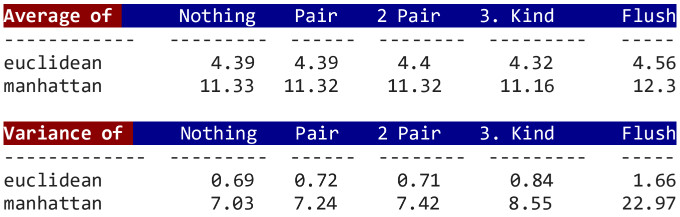
[20] https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

8 K-NEAREST NEIGHBORS (KNN)

Distance (a.k.a. similarity) metric function

Manhattan and Euclidean are the most common distance metrics. Both are meant for real valued-vectors and therefore the data has been scaled using the same method used for **MLP** and **SVM**. Choosing the right function is critical for KNN and therefore both functions are compared. The training set is separated by class, each subset contains all samples of a class. Then the Euclidean and Manhattan distance functions of the pairs formed by the list and its inverse (e.g. first sample vs last, second vs second-to-last) are calculated. The variance and averages are compared in **Table 6**. Only some classes are shown for convenience, but the other classes are alike.

Table 6



```
Average of        Nothing    Pair    2 Pair    3. Kind      Flush
------------     ---------  ------  --------  ---------     -----
euclidean             4.39    4.39       4.4       4.32      4.56
manhattan            11.33   11.32     11.32      11.16      12.3

Variance of       Nothing    Pair    2 Pair    3. Kind      Flush
-------------    ---------  ------  --------  ---------     -----
euclidean             0.69    0.72      0.71       0.84      1.66
manhattan             7.03    7.24      7.42       8.55     22.97
```

At first, Euclidean appears as a better choice as it provides a smaller variance which would in principle help cluster all samples of same class together, making the cluster separation easier. The problem is that the average value of all classes is too similar. E.g. the average Euclidean distance between samples from class “Nothing in hand” is 4.39, the same average is found for samples from class “One pair”. For “Two pairs” the average is 4.4, almost identical, similar result for “Three of a kind”. The same goes for the variance. The variance in the distances from the first two classes is 0.69 and 0.72. This is true for Euclidean and Manhattan, and even for the scaled and not scaled samples. One exception is class “Flush”, with a more significant average value and higher variance. When comparing the cross-class (i.e. compare samples from two different classes) variance for the most-dominant classes (“Nothing in hand” and “One pair”), the variance is almost the same as each class separately: 0.6986. This should cause that the classifier faces difficulty when trying to cluster samples from the same classes together, since the Euclidean and Manhattan distance from arbitrary classes are too similar from the distances of samples from the same class, i.e. it will be difficult to create separate clusters for each class in the dataset.

Algorithm

SciKit-learn provides three algorithms to perform KNN: **Brute Force**, **KD Tree** and **Ballpark**. Each has their pros and cons, e.g. Ballpark is very efficient in high dimensional spaces (> 20D), Brute Force is more efficient for datasets with few samples (e.g. < 30) and KD Tree outperforms Brute Force where the number of samples is larger but Ballpark is still preferred if working on high-dimensional space. Lots of details can be found in the official documentation.[21] Brute Force is not tested because it is designed for datasets with only a few samples.

Model complexity analysis

The weights can be distance or uniform. Intuitively it appears that weights based on distance is more appropriate for the imbalanced dataset, as there will be probably very few (if any) neighbors together of the classes with less than 5 samples, and the rest of the k-nearest neighbors (if choosing higher k) will definitely be from a different class. The following varying values are evaluated. **Algorithms**: Ball tree and KD tree. **Weights**: uniform and distance (the closest neighbors have more weight). **K: 2** (smaller than min. # of samples per class), **5** (minimum number of classes) and **100** (arbitrary large). The results are shown in **Figure 16**.

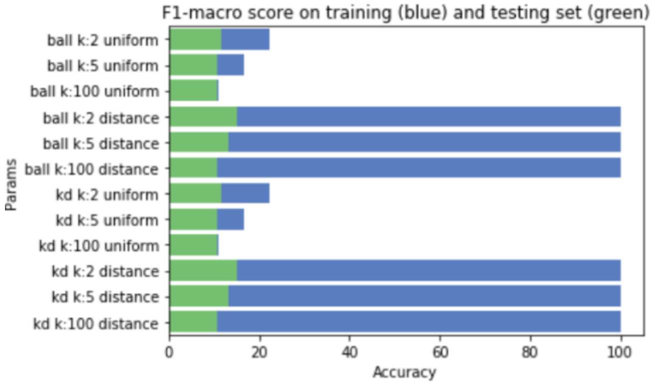


Figure 16

As expected, the weighted distance provided the best results. The perfect score in the training set can be explained in terms of the distance value of each training sample with itself, which is the lowest (i.e. 0.0). Other samples aren’t close enough for their weighted distance to overthrow it, not even when using 100 neighbors (k=100). Another possibility is that the distance function is truly capturing the fact that samples of the same class are clustered together, although this is unlikely due to the problem mentioned in the previous analysis.

Results

Figure 17 shows the Confusion Matrix. The hyper-parameters chosen for this result are 2 neighbors, Euclidean distance metric, with distance weights and using the Ball-tree algorithm. As expected, the classifier’s performance wasn’t good. Most classes are miss-classified as one of the most-dominant classes, but even the samples from the dominant classes get largely miss-classified as some other class. This is due to the problem explained earlier, where the distance metrics, either Euclidean or Manhattan, aren’t able to create separate differentiable clusters for each class. One exception is class 5 (“Flush”). As mentioned earlier as well, this class is significantly different from the other classes, and this allows the classifier to do a much better job for its samples. The Classification Report is shown in **Table 7**. The overall performance of the classifier is pretty poor, achieving a 15% accuracy when using the original data. A comparison with the results achieved when using the transformed data is shown below.

[21] https://scikit-learn.org/stable/modules/neighbors.html#choice-of-nearest-neighbors-algorithm

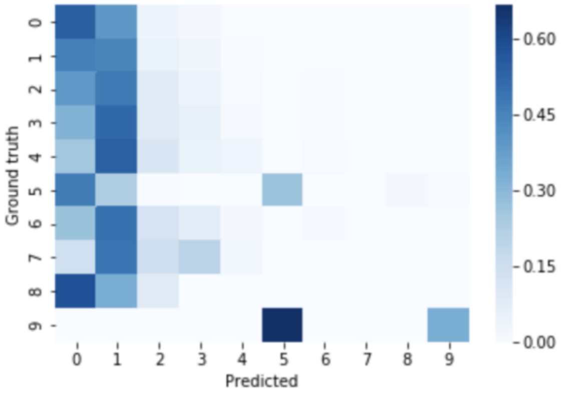


Figure 17

Comparing build vs query time

Figure 18 compares the build (blue) vs query (green) time taken for each model. It appears that we're not plotting the build time, but what's truly happening is that the build times are in the milliseconds range while the query times, since it is querying all of the instances of the training-set at once, are in the seconds range. In average build time for all tests was in between 0.025 and 0.030 seconds. The larger build times of the ball tree are expected due to the underlying trees' structure: "*This makes tree construction more costly than that of the KD tree*"[22]. For all cases, query time is expected to grow as k increases, as the tree needs to be traversed more to find the required neighbors.

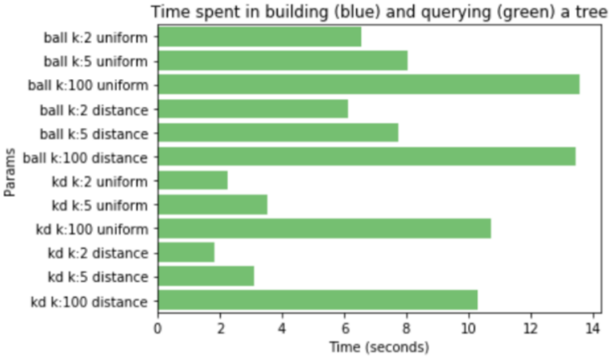
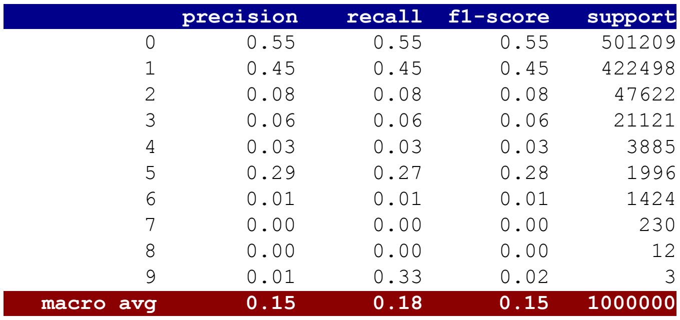


Figure 18

Comparing the results of original data and transformation

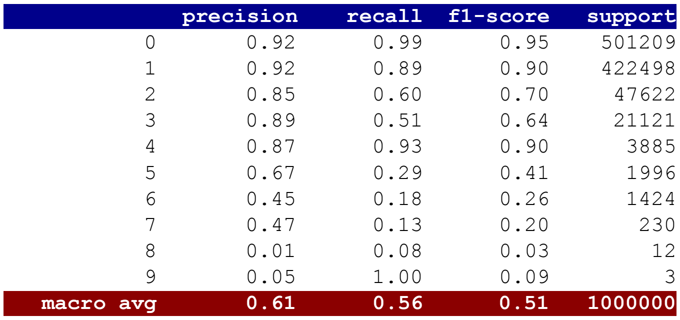
Table 7 and **Table 8** show the Classification Reports from the original and transformed dataset, respectively. The F1-macro scores are 15% for the original and 51% for the transformed dataset. Notice that with the exception of class 5, every other class seems to have been miss-classified as classes 0 or 1 in the original dataset. Class 5 happens to be one of the classes that has an acceptable number of samples (>1000) and the average Euclidean distance metric result is over 0.20 above the averages of class 0 and 1. The other classes, with the exception of class 4, either have a very low representation or are only at 0.10 above/below the averages of class 0 and 1 (4.40). The classification report for the transformation is using the same hyper-parameters. The variance report of the distance metrics for the transformed data is seen in **Table 9**. The average distances for the samples of the same class are clearly easier to distinguish. This is a fundamental requirement for the KNN algorithm, as it depends on being able to cluster together the samples of the same class using one of the distance metrics. **Figure 19** shows the Classification Report. A clear improvement is observed.

Table 7



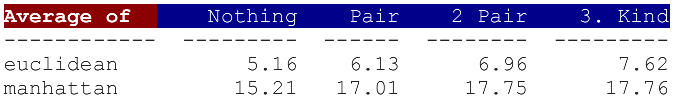
	precision	recall	f1-score	support
0	0.55	0.55	0.55	501209
1	0.45	0.45	0.45	422498
2	0.08	0.08	0.08	47622
3	0.06	0.06	0.06	21121
4	0.03	0.03	0.03	3885
5	0.29	0.27	0.28	1996
6	0.01	0.01	0.01	1424
7	0.00	0.00	0.00	230
8	0.00	0.00	0.00	12
9	0.01	0.33	0.02	3
macro avg	0.15	0.18	0.15	1000000

Table 8



	precision	recall	f1-score	support
0	0.92	0.99	0.95	501209
1	0.92	0.89	0.90	422498
2	0.85	0.60	0.70	47622
3	0.89	0.51	0.64	21121
4	0.87	0.93	0.90	3885
5	0.67	0.29	0.41	1996
6	0.45	0.18	0.26	1424
7	0.47	0.13	0.20	230
8	0.01	0.08	0.03	12
9	0.05	1.00	0.09	3
macro avg	0.61	0.56	0.51	1000000

Table 9



```
Average of      Nothing    Pair   2 Pair   3. Kind
------------  ---------  ------  --------  ---------
euclidean          5.16    6.13      6.96       7.62
manhattan         15.21   17.01     17.75      17.76
```

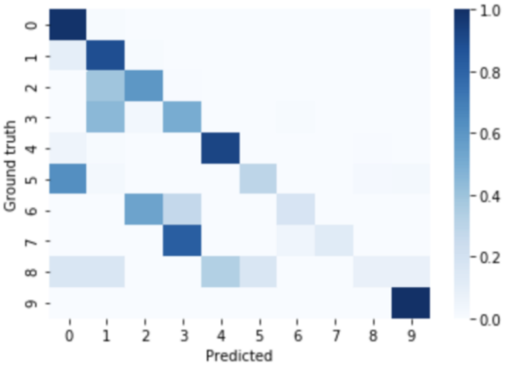


Figure 19

General results

When working with the original dataset, it has been found that the distances (Euclidean or Manhattan) between two samples from the same class, and two samples from arbitrary classes, are nearly identical, i.e. they are all clustered around a Euclidean average of 4.4 with +/-0.20 variance between them. This causes that KNN isn't able to find clearly distinguishable clusters for each class, therefore random samples from all classes are miss-

[22] https://scikit-learn.org/stable/modules/neighbors.html#ball-tree

classified as one other class. The one exception is the "Flush" class, which does have a significantly different distance metric and variance (for samples between its class compared to samples from other classes). This is the only non-dominant class that gets an F1-score of above 20%. All other non-dominant classes are under 8%. Even the dominant classes get largely miss-classified and barely achieve a 50% accuracy score.

With the transformed dataset, the distance metric average and variance between cross-class and in-class samples are significantly easier to distinguish. The average values for in-class average distance is shown in **Table 9**. The result is that the dominant classes now get over 90% accuracy (vs 50% from original data) and only 2 classes get less than 10% accuracy. The result is a significant improvement, but nevertheless a very poor 50% overall accuracy for the KNN classifier.

While it is possible to further massage the dataset or adjust the hyper-parameters to achieve better accuracy, this is not further pursued as the objective of comparing the performance obtained with and without the transformed dataset has been met.

9 CONCLUSIONS

I used the **F1 macro-average score** metric in this paper to measure the performance of the classifiers. I showed that this metric is appropriate for this dataset, given that it is imbalanced. Other metrics can easily give the false impression of success if the dominant classes are classified correctly, while the non-dominant classes are not. For example, the classifier in [5] is correctly classifying 100% of the 90% dominant samples; since the metric used gives weight to the classes proportional to the number of samples, the overall result is over 90% accuracy. In this paper, I showed that when the metric is unweighted, the true accuracy of the classifier is only 70%.

I have shown that the Poker-hand dataset is human-friendly but not Machine Learning-friendly. All classifiers saw a significant accuracy improvement after applying a simple linear transformation to the dataset that made it more appropriate for Machine Learning. For example, I showed that a simpler MLP model provides equivalent results in less computational time if using the transformed dataset. Specifically, I removed a layer of 100 neurons without compromising the performance of the classifier. The results show that the Neural Network accuracy is similar or better than the one achieved by the more complex model while reducing the training time by 25% to 50%.

REFERENCES

[1] R. Cattral and F. Oppacher, "Poker Hand Data Set," Carleton University, Department of Computer Science, 2007. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Poker+Hand.

[2] S. Boriah, C. Chandola and K. Chandola, "Similarity Measures for Categorical Data: A Comparative Evaluation," in *Proceedings of the SIAM International Conference on Data Mining*, Atlanta, Georgia, USA, 2008.

[3] T. Mitchell, "McGraw Hill series in computer science," in *Machine Learning*, New York, McGraw-Hill., 1997, p. 105.

[4] J. Heaton and J. Heaton, "The Number of Hidden Layers," Heaton Research, 2017. [Online]. Available: https://www.heatonresearch.com/2017/06/01/hidden-layers.html.

[5] Brownlee and J. Brownlee, "A Gentle Introduction to Imbalanced Classification," Machine Learning Mastery, 2019. [Online]. Available: https://machinelearningmastery.com/what-is-imbalanced-classification/.

[6] Bhardwaj and A. Bhardwaj, "Poker-Hand Prediction," Medium.com, 2019. [Online]. Available: https://medium.com/@virgoady7/poker-hand-prediction-7a801e254acd.