Forensic Identification of Glass Fragments

Course: Machine Learning (BSMALEA1KU)

IT University of Copenhagen

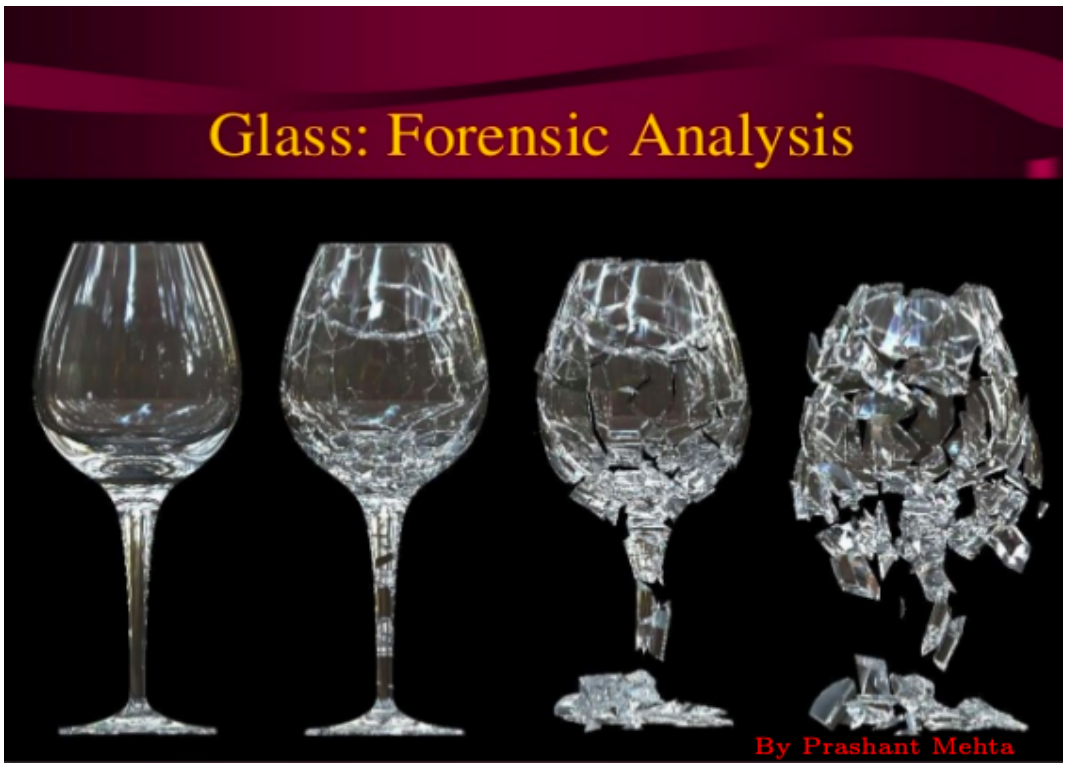
January 2022

Group A

Eva Christelsdóttir, evac@itu.dk

Jonas Toft-Jensen, joto@itu.dk

Julie Skoven Hinge, juhi@itu.dk



1 Introduction

In crime scenes, every small detail may be the piece of evidence leading to catch the perpetrators. Be it fingerprints, a droplet of blood, or even a footprint. To prepare for every detail, police must have techniques to process the data they gather on site. One of the types of data is the source of objects found on a scene, such as glass shards. This could help distinguish between the broken window in the shop broken into, the headlamp from the perpetrator or the wine glass from the restaurant.

In this study the viability of using machine learning to trace the origin of glass is investigated. The study will use 5 different classification methods to predict the source of different types of glass. Each method used will be introduced in each section along with its results.

2 Our Approach

This data was gathered from a study carried out by Evett and Spiehler(1987) and was part of *the UK forensic science service* [5]. The data set consists of 214 observations in total that belong to 6 different classes. For each of these observations there are 9 features - one describing the refractive index (RI) and the rest describing the chemical composition of the glass fragment. All 9 features are shown in figure 1:

RI	Na	Mg	Al	Si	K	Ca	Ba	Fe
refractive index	Sodium	Magnesium	Aluminum	Silicon	Potassium	Calcium	Barium	Iron

Figure 1: Attributes of the glass identification data set

Out of all the observations, 149 were used for training data, and the remaining 65 were used in our testing process. For the training of our models, we decided to use both a training, validation and testing data set, which means that the 149 training observations were further split into a training set consisting of 111 samples (75%) and a validation set consisting of 38 samples (25%).

For the exploratory data analysis, we plotted the distribution of the classes with glass types for our training data and found that glass type 2 (*Window from building (non-float processed)*) was the most common class with 53 observations while glass type 6 (*Tableware*) was the least common class with just 6 observations. This class distribution can be seen in figure 2.

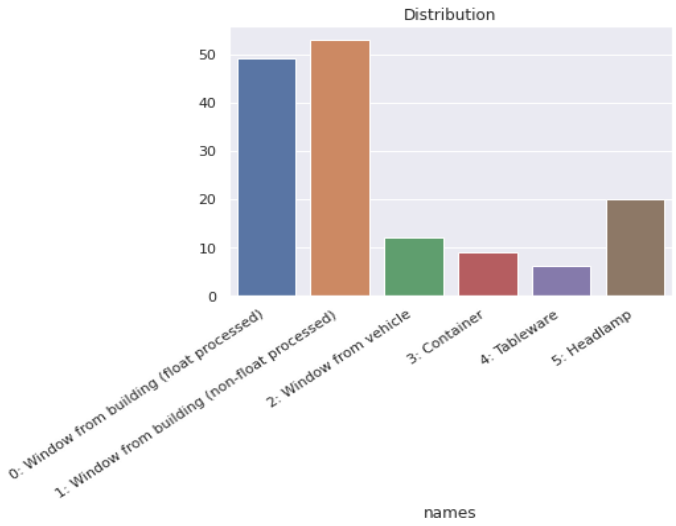


Figure 2: Distribution among classes

For each classification method we will calculate its accuracy score and F1-score. The distribution of classes gives us somewhat an expectation to the findings of our classification methods. The dense glass types with the most observations can be expected to get classified the best - glass type 2, then glass type 1 and glass type 7.

However, we do expect some misclassification for all classes. As seen in figure 3, there exists a large overlap between the different classes, especially between classes 0, 1 and 2. This makes sense, since float processed windows from buildings (class 0) and non-float processed windows from buildings (class 1) only differ in their production, and vehicle windows (class 2) is arguably similar to class 0 and 1. This overlap is expected to have an effect on our results in terms of misclassification. Furthermore, glass from headlamps is too scarce and spread out in the first two dimensions to meaningfully distinguish them from the other classes.

Figure 3: Scatterplot of PCA transformed data

3 Pre-Processing and Data Manipulation

The data was loaded as a NumPy array and split into the observations and true classes. As the true classes did not contain any glass type 0 or 4, the class labels were shifted down to the range 0-5. We also made sure to check for missing data, which there were none of, and so we were able to proceed without any further data manipulation.

Following this, we normalized our data to the interval [0, 1], which allowed us to have a common scale across all measurements for our classes. The normalized version performed best in all cases, and so the result will reflect the models' performances using the normalized data rather than the un-normalized. Furthermore, we made a separate variable to hold a version of our normalized data that has been transformed using Principal Component Analysis (PCA), which we used to test the effects on the accuracy score. Each section for each method will include details on whether the PCA-transformed data was preferred.

4 Decision Tree - M1

4.1 Introduction of the method

Decision trees work by constructing a 'tree' of if statements. If the data fulfils the if statement requirement, it will follow the branch to one side, while if it does not, it will follow the branch to the other. The data goes down through the children branches until it ends in a leaf node. The prediction from a leaf node will depend on the fraction of the training data that were assigned to each class.

Decision trees are built via a greedy algorithm. This means that while all the splits are locally the most optimal, the tree is not ensured to get the optimal decision regions.

4.2 Application of the method

In our implementation, two classes were made. One was a tree class which served as the storage for variables and the interface. The other was a leaf class which works as the specific implementation of the tree and is created by the tree class when using the fit method. With the fit method, a leaf was made root. This leaf would then split and create two new leaves, that would split recursively until the stop condition was met.

The split was found by inspecting each column by itself. Each column was sorted, and then the midpoint between each adjacent value saved. For each midpoint, the potential Gini impurity with this split for the data in the leaf was found.

The Gini impurity for each leaf was found by first finding the fraction each class represented of the leaf. These fractions were squared, and all summed together. This measure of purity is then subtracted

from 1:

$$i(Leaf) = 1 - \sum_{k=K} (p_k)^2$$

The combined Gini impurity for at split is then found as the impurity of each of the two leaves, multiplied by the fraction of the data that will be in the leaf:

$$split = \sum i(leaf) * p(leaf)$$

If the Gini impurity was lower than the previous highest Gini impurity, the cutoff-point and column to split on would be saved. When all possible splits have been considered, there will be a best split with the Gini impurity as measurement for the impurity. If the impurity is lower than a threshold, it will recursively create two children leaves with the data that belongs to each. These leaves will then again find the best possible split and create two new leaves, until a stop condition is met.

When a stop condition is met, the node will be marked as a leaf. It will then calculate the fraction that each class represents in it and mark these fractions as the return-values when predicting new observations.

These stop conditions exist as different hyper-parameters. Max-splits determines the maximal depth of the tree. Min-gini determines the minimal Gini-impurity that a split needs to have for it to make the children. If it has a higher impurity, it will be made a leaf. Min-leaf-size determines the minimal size of a leaf, and a tree must have more observations than two leaves to make a split. Lastly, min-gini-improvement, which is a number between 0 and 1, that determines how much the new split should reduce the gini-impurity for the split to be accepted.

4.3 Results

We used the *DecisionTreeClassifier* library from Scikit-learn to make a reference implementation and compared this with our decision tree model. We also tried using the PCA-transformed data but found that it worsened our result and decided against using it.

With the use of our validation data set, we found the results via a search using the hyper-parameters of depth, minimal number of nodes in a leaf and a minimal gini improvement. The model that had the highest F1 micro-averaged score on the validation data was saved. Both of the models with the highest score were then applied on the test data.

The accuracy of the Scikit-learn implementation was 76.9%, and 61.5% for our model. The F1-score micro-averaged was 75.2% for the Sci-Kit learn implementation and 52.2% for our implementation.

One of the differences between the two models is how the size of leaves and splits are handled. In our implementation, there is no difference between the minimum size of the split and the size of a leaf. In the Scikit-learn implementation, the minimal size of a split and the minimal size of leaves are two different

parameters. The split size is the size a node will minimally have for it to split. The leaf size is the minimal size a new leaf needs to have when created. This distinction allows for smaller and potentially purer leaves while still maintaining enough data points when splitting to find good decision boundaries. The risk with this approach is that this more easily overfits the data, however, the benefit is that it can also avoid underfitting, especially when dealing with classes containing small amounts of data.

Another difference is the use of randomness when constructing the tree. In the implementation from Scikit-learn, only a random subset of the features are considered when making a split, while all features are considered in our model. While this makes the construction of the tree faster, it should theoretically make the tree less accurate, as it might miss some optimal decision boundaries. As the amount of data is very limited with only 214 observations for training, this speed optimization was not required, and thus not implemented in our project.

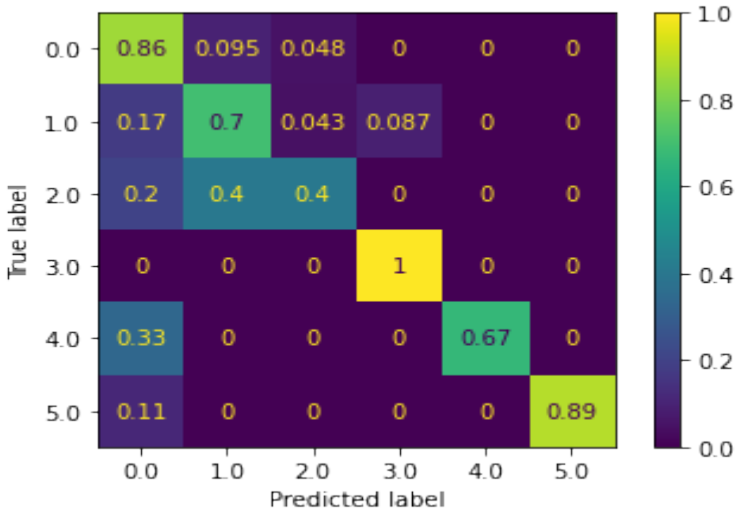


Figure 4: Confusion matrix of Decision Tree (row-normalized)

When looking at figure 4 of the confusion matrix of Scikit-learn's decision tree, we see some misclassifications, especially for class 2. The 40% accuracy follows very closely the distribution in the training data, where only 42% of the training data is in leaves classified as class 2. Class 3 has a high recall of 1, similarly, classes 0 and 5 have a close to 90% recall.

The discrepancy in the F1-score micro-average between our handmade model and the Scikit-learn implementation is likely a result of underfitting for our hand-made model for some of the smaller classes. Especially class 4, which in the Scikit-learn's model gets an area to be predicted by having small nodes with just 1 or 2 of class 4 nodes. These are not present in our hand-made tree, as they are too dispersed to form a majority in any leaf and will thus not be predicted in any leaf.

4.4 Visualization of the Decision Tree

Figure 5 shows the reference implemented decision tree from Scikit-learn, along with the splits and purity of each split. As it can be seen in the figure, impure leaf node was almost exclusively impure with class 0, 1 and 2. Furthermore, class 5 could be almost exclusively separated from all other classes, thus it makes sense that class 5 had a high recall of almost 90%.

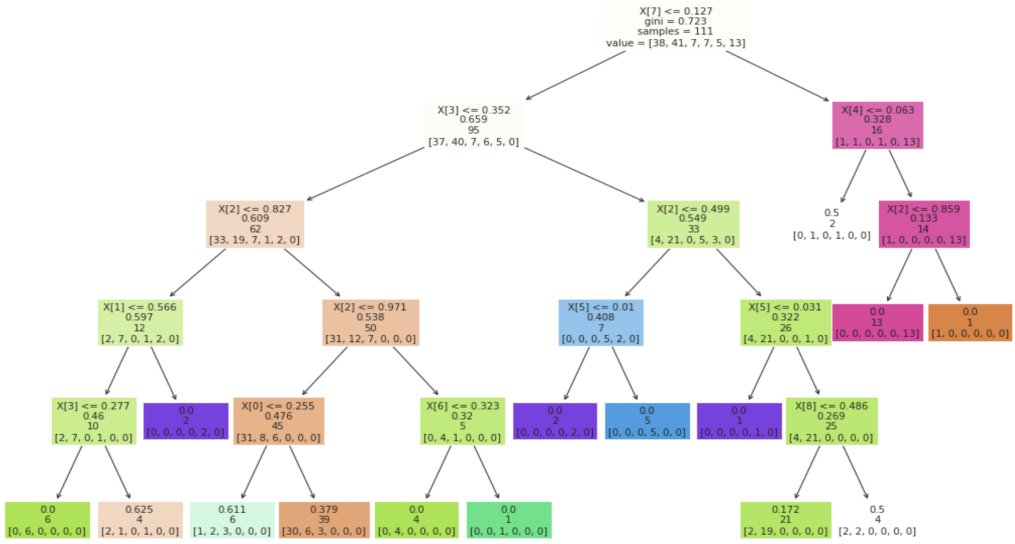


Figure 5: Visualization of Decision Tree when trained

5 Feed-forward Neural Network - M2

5.1 Introduction of the method

A feed-forward neural network, also known as the multilayer perceptrons (MLPs), is a supervised deep learning algorithm consisting of several layers and neurons used to make class predictions.

5.2 Application of the method

In our case, the first layer in the neural network contains 9 neurons - the number of features, while the last layer contains 6 neurons - the number of classes. This means that the input of the first layer is the features for a certain observation and the output of the last layer is a guess on which class that observation belongs to.

All the layers in-between the input and output layers are called hidden layers and contain neurons as well. These neurons each contain a weight, bias and an activation, and the sum of these individual weights multiplied with the corresponding activation plus a bias are used to determine the activation of

the next layer. We can therefore say that the output layer is made up of complex transformations of the input, X. Before this occurs though, the activation multiplied with the weights plus the bias is put through an activation function which ensures non-linearity.

We decided to use the Rectified Linear Unit (ReLU) activation function, which mathematically is defined as $y = max(0, x)$. ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.[3] ReLU was chosen since it can be used in multi-layer networks, and it is known to allow models to perform better and faster. We also decided to use the Softmax activation function on the final layer to get a probability on each class from which the highest probability result will be our prediction.

So, the transition of deciding each activation with the previous layer's nodes and putting that through an activation function can mathematically be described as:

$$a^{(1)} = \sigma(Wa^{(0)} + b)$$

In the first round of the neural network, all weights and biases will be initialized randomly. Because they are randomly initialized, the neural network will perform very badly in making the classifications. To fix this, we must calculate how wrong the guesses were by using a cost function. We decided to use the cross-entropy loss function since we are working with a classification problem and since it outputs a probability between 0 and 1 for each class. For us to perform cross-entropy we first had to one-hot-encode all our response labels, meaning that there had to be a 1 where the class is true and otherwise 0's. After doing this we can perform cross-entropy loss using the formula: [1]

$$-\sum_{n=0}^{n}\sum_{m=0}^{6} y_{im} log(f_m(x_i))$$

According to this, we are looking at the sum of the loss of all our training observations. For our neural network to perform well, we want to minimize this loss. This is done by performing gradient descent which updates the weights and biases of the neurons by taking steps in the direction that will minimize the cost function.

The direction that minimizes the cost function the most is found by finding the gradient of every layer relative to the cost function.

We do this in a "backwards" manner known as backpropagation, which in short is a technique to efficiently compute the gradient of the cost function automatically. This backpropagation algorithm computes the neural network's error with regard to every model parameter, meaning that it will find how much the weight and bias should be modified to reduce the error. We start from the final layer, meaning that these error vectors are computed backwards, since the cost is a function of outputs from the neural network. We need to recognize how the cost differs with earlier weights and biases, and thus

we repeatedly apply the chain rule until the neural network obtains functional expressions and converges to a solution.

5.3 Results

When implementing our own hand-made feed-forward neural network, we drew inspiration from a neural network made with NumPy, which we found on GitHub.[4] After training on our validation data and then testing on our test data, we got an accuracy score of 60.0% and an F1-score of 54.0% with the use of the normalized data.

For our reference implementation of the neural network we chose to use Scikit-learn's *MLPClassifier* library. We used Scikit-learn's *Grid Search CV* to find optimal hyper-parameters using our validation data. It was found that the optimal activation function was 'ReLU', optimal solver was 'lbfgs', alpha should be 0.05 and lastly, that the learning rate should be 'constant'. Furthermore, it was discovered that using the PCA-transformed data provided us with the better results. When running our test data, we got an accuracy score of 70.8% and an F1-score of 58.1%. To further explore, the *Sequential* library from Tensorflow-Keras was attempted, but its results were worse than the previous mentioned.

6 Random Forest - M3

6.1 Introduction of the method

Random forest is an ensemble of decision trees. The method works by creating a multitude of decision trees that are separately trained. The final prediction of the random forest will then be the most often predicted class by the individual trees.

For each tree created, a random subset of the columns will be chosen. Furthermore, the data for each tree is randomly sampled with returned, which is called *bagging*. The performance of the tree is then measured by how well it classifies the data not included in the *bagging*.

As decision trees are likely to overfit their training data, the use of many trees is a way to avoid overfitting. This comes from the fact, that every tree gets different *baggings* which results in overfitting, differently, and independently of other trees. While the bagging makes the individual trees less accurate, the overfitting mistakes are more likely to be different for each tree. The majority of the trees will then tend to avoid that specific overfitting, and thus equalize some of the mistakes. As the overfittings are then voted out by the majority, the ensemble tends to generalize better. This method uses the "wisdom of the crowd" to make its predictions.

6.2 Application of the method

For this classification method we used the *Random Forest Classifier* library from Scikit-learn. We tried using the PCA-transformed data but found that it worsened our result and decided to do it just using the normalized data. Additionally, *Grid Search CV* from Scikit-learn was used with our validation data to find optimal hyper-parameters and achieve the best result.

6.3 Results

When running our test data, our random forest model gave us an accuracy score of 75.4% and an F1-score of 66.6%. Meaning that it yields better results than our hand-made decision tree, which was expected, since random forest is an ensemble of several trees. However, this is slightly lower than the Scikit-learn's decision tree, which has an accuracy score of 76.9%. There could be two reasons for the lower accuracy of the random forest. One is that since there are fewer columns to train on for all the specific trees, several of the trees will be worse than the big decision tree, and so it generalizes badly. Another is that since some of the classes are very small and spread out, they risk being outgeneralized by the larger classes.

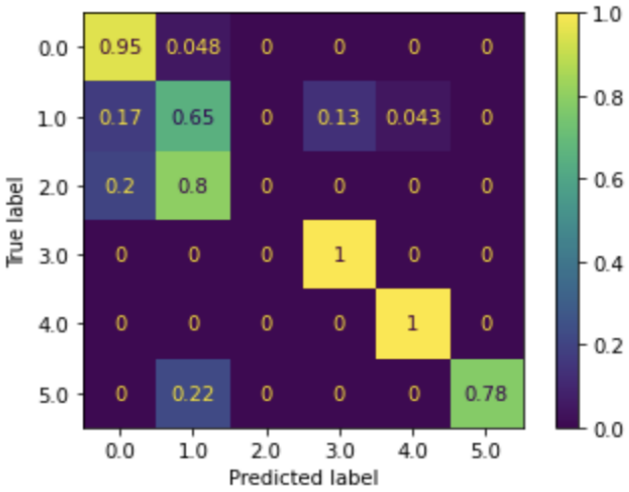


Figure 6: Confusion matrix of Random Forest (row-normalized)

Figure 6 shows the row-normalized confusion matrix resulting from our random forest model. It has a high recall for all the classes, especially class 3, 4 and 0. It does however misclassify all of class 2, and it is thus unable to distinguish vehicle windows. This was expected, since class 2 has considerate overlap with class 0 and class 1, as discussed in section 2, and thus could be expected to get misclassified, as in our case. Furthermore, figure 3 showed that class 0 and 1 were somewhat overlapping and this has proven to make it difficult for the model to make a distinct split between these two classes - e.g., 4.8% of class 1 has been classified as class 0.

7 Support Vector Machine - M4

7.1 Introduction of the method

Support Vector Machine (SVM) is a supervised machine learning algorithm. The objective of SVM is to find a hyperplane in an N-dimensional space that distinctly classifies the data points.[6] The goal is to find the plane with the maximum distance between data points, e.g. maximum margin.

To implement SVM's we focus on the points that lie on the edges of each data cluster and we try to find a hyperplane that lie in the middle between each cluster. The points that lie on the edges help make up support vectors which are vectors from the origin to the points.

To make the best classifications we must use the midline (decision boundary) between these support vectors as a threshold. The shortest distance between the midline and each support vector is called a margin, and when we are making classifications, the objective is to find a maximum margin such that the decision boundary is in the middle between each support vectors.
The widest margin is calculated by minimizing

$$\frac{1}{2}||W||^2$$

which is subject to the constraint

$$y_i(\vec{w} * \vec{x_i} + b) - 1 \geq -1$$

Where $y_i = 1$ if an observation belongs to one class and $y_i = -1$ if it belongs to the other.
This constraint further explains that all points are correctly classified and are lying on the margins or further (However, this is an assumption). It is easier to minimize this function since it is convex and differentiable as opposed to maximizing half the width.
Now we can predict new observations by using the formula,

$$h(x) = \sum_{i \in S} a_i y_i (x_i \times x) + b$$

SVM's work by plotting two features against each other and since our task is a multi-classification problem this means that the model must implement a "one-versus-one" approach resulting in $\frac{n_classes*(n_classes-1)}{2}$ constructed classifiers in total.

When using SVM's we can also either decide to use a soft or a hard margin. Using a soft margin means that we allow some misclassifications and vice versa.

7.2 Application of the method

We used the Scikit-learn *Support Vector Machine* library when implementing this algorithm.

In our implementation we tried specifying the kernel as a Gaussian, a polynomial, or a Sigmoid kernel. We also used a penalty, meaning that we penalized values that lie with-in the margin since we have some overlap in our data. A lower penalty means that we would have a soft margin and therefore allow more misclassifications but at the same time make the model more generalized.

7.3 Results

After trying different hyper-parameters on our validation data, it was discovered that using PCA-transformed data with a Gaussian kernel with a penalty of 2 gave us the best result. With these parameters we obtained an accuracy score of 69.2% and an F1-score of 48.8% on our test data.

8 K-Nearest Neighbors - M5

8.1 Introduction of the method

K-Nearest Neighbors (KNN) is a nonparametric method and a supervised machine learning algorithm, which in our case is used to solve a classification problem. KNN classifies new data points based on K nearest other data points.

When classifying a new data point x, we look at the sphere centered on x containing precisely K points. To minimize the probability of misclassification, we assign the test point x to the class/sphere with the largest value of K_k/K - meaning the highest posterior probability. We obtain the posterior probability of class membership using Bayes' theorem [2]:

$$p(C_k|\mathbf{x})\frac{p(\mathbf{x}|C_k)p(C_k)}{p(\mathbf{x})} = \frac{K_k}{K}$$

8.2 Application of the method

We wanted to find the K which reduces the misclassifications we encounter while maintaining the ability of the algorithm to accurately make predictions when it is given new data points. For this we chose to use Scikit-learn's *KNeighborsClassifier* library.

Our approach was to run the algorithm several times and to do a hyper-parameter selection with different K values in the search of the best suited K. As PCA just rotates the data, all points will be in equal distance from each other in both normalized and PCA space. Since the results would be the same, the model was not trained on the PCA-transformed data.

8.3 Results

After exploring hyper-parameters with our validation data it was found that K should be specified to 1 resulting in an accuracy score of 67.7% and an F1-score of 62.4%. KNN is a simple classifier and has proven to be affected by the complexity of the given data set and its results reflect this. The result can be seen on the validation plot in figure 7. Based on the plot we can confirm that setting k=1 would be the best option. Furthermore, this value is proven to be the best for both the training data and the testing data.

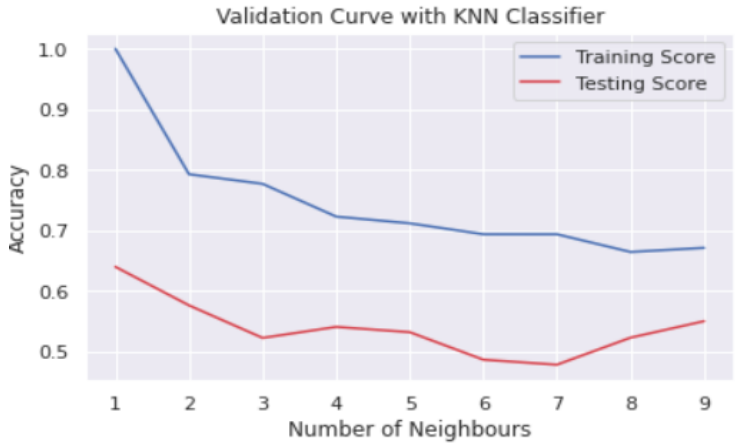


Figure 7: Validation curve of K-Nearest Neighbors

9 Discussion

Our expectation made from the class distribution was to some extent in accordance with our results, e.g., class 2, which was arguably similar in categorization with classes 0 and 1, and thus expected to get misclassified, was the worst classified class in both the confusion matrix of our decision tree and in the confusion matrix of our random forest.

A core challenge was that we had a very small amount of data to start with, which then became even smaller after splitting it into a train, validation, and test set. The small amount of data thus made it very difficult to obtain good results.

Other challenges we faced was that the classes were somewhat overlapping, and some classes were very sparse, e.g. glass type 6 containing only 6 observations. This particularly became a challenge in the support vector machine model and the K-nearest neighbors model since they are especially in need of being able to make a ”clean” cut between classes, as opposed to the decision tree model and random forest model, where the decision boundary is able to ”carve out” the classes more clearly and thereby not depend on a linear separation of the data like i.e SVM does.

From the classifiers in this study, it is viable to trace some glass shards to their origins. Glass from windows, either with or without float processing, were possible to distinguish from most other types of glass, although not each other very well. Glass from containers were also able to get recalled with 100% recall in multiple of the models. Windows from vehicles was less possible to be classified correctly, and only had 40% recall in the best classifier. This is because there was quite a large overlap of the data between the different classes.

9.1 Method comparison

Accuracy scores and F1-scores (macro-average) for the 5 methods		
Model	Accuracy score	F1-score
Decision Tree(Sklearn implementation)	76.9%	75.2%
Random Forest	75.4%	66.6%
Neural Network(Sklearn implementation)	70.8%	58.1%
Support Vector Machine	69.2%	48.8%
K-Nearest Neighbors	67.7%	62.4%
Decision Tree(our implementation)	61.5%	52.2%
Neural Network(our implementation)	60.0%	54.0%

As seen from the table above, the decision tree made with Scikit-learn's library has the highest accuracy score of 76.9% and the highest F1-score, in part because it had a recall of at least 40% for all classes. The random forest performed almost as good as the decision tree, but the decision tree's ability to correctly classify glass from class 2 distinguishes it from the random forest, which misclassified all glass from class 2. The random forest however does have a higher recall on class 0 and class 4, so if high recall for these classes is needed, the random forest could arguably be the optimal classifier.

10 Conclusion

The aim of this project was to investigate which classification techniques were suitable for determining the origin of glass fragment using elemental composition and refractive index. We chose to use 5 different classification methods, which all had various strengths and weaknesses.

Overall, the decision tree proved to have the highest accuracy, as well as being the most stable across different classes. This study suggests that classification can give the police a suggestion of where a shard of glass came from, but that classification is not precise enough to be used as proof in court. Further research could arguably look into if other variables for glass have a higher predictive power, as well as

how more data would improve the models.

References

[1] Gareth James Et al. *An introduction to statistical learning*. 2013, pp. 403–458.

[2] Christopher M. Bishop. *Pattern Recognition And Machine Learning*. 2006, p. 126.

[3] "How do ReLU Neural Networks approximate any continuous function?" In: (2021). URL: https://towardsdatascience.com/how-do-relu-neural-networks-approximate-any-continuous-function-f59ca3cf2c39.

[4] "Neural-Network-Numpy". In: (2020). URL: https://github.com/mlpotter/Neural-Network-Numpy/blob/master/NeuralNetworks.ipynb.

[5] "Rule Induction in forensic science". In: (1987). URL: http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/evett_1987_rifs.pdf.

[6] "Support Vector Machine — Introduction to Machine Learning Algorithms". In: (2018). URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.