IT UNIVERSITY OF COPENHAGEN

Machine Learning

Final Project: Forensic Identification of Glass Fragments

Group AC:

Carl August Wismer (cwis@itu.dk)
Chrisanna Kate Cornish (ccor@itu.dk)
Danielle Marie Dequin (ddeq@itu.dk)

3rd Semester, Autumn 2021
Data Science BSc

BSMALEA1KU

1 Introduction

The investigation of crimes can involve the forensic analysis of glass. Previous studies have shown that the origin of small fragments can be determined based on the elemental composition and refractive index (Evett and Spiehler). This can be evidence that indicates the source of the the glass fragments present on a crime suspect or scene.

This project implements three machine learning (ML) methods and compares their success in accurately classifying glass fragments. Two of the ML methods, feed-forward neural network and decision tree, are implemented from scratch, with results compared against a reference implementation using Python libraries. The final method employed is Gaussian Naive Bayes, using the sci-kit learn library. The strengths and weaknesses to each approach will be discussed and their suitability to this kind of task.

The purpose of this report is to describe the implementation of the different machine learning models used on the data set, and to see how they compare to each other and why. This paper will use and analyze the Neural Network, Decision Tree and Naive Bayes methods.

1.1 The Data

The data is provided by Dua and Graff (Dua and Graff). The data set contains 214 glass fragments and their measurements across nine different features. These features include Refractive Index and the chemical composition of eight different elements. The data are divided in a training set with 149 rows and a test set with 65 rows.

1.2 Data Exploration and Pre-Processing

During exploratory data analysis it became apparent that the features do not all fall within the same range, with variations in mean and standard deviation (SD). For example Iron had a mean of 0.06 and SD of 0.1, whereas Calcium had a mean in the data of 8.92 with a 1.51 SD. To handle this the data was standardized to a mean of 0.0 and SD of 1.0 to bring the features to a common scale without distorting the differences in the range of the values. Standardization used a z-score formula taken from lecture notes at ITU (Grbic). Table 1 shows the descriptive statistics of the original data set.

Table 1: Feature names and descriptive statistics

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe
count	149.00	149.00	149.00	149.00	149.00	149.00	149.00	149.00	149.00
mean	1.52	13.42	2.72	1.43	72.62	0.49	8.92	0.20	0.06
std	0.00	0.86	1.42	0.51	0.78	0.57	1.51	0.55	0.10
min	1.51	10.73	0.00	0.29	69.81	0.00	5.43	0.00	0.00
25%	1.52	12.93	2.28	1.17	72.28	0.13	8.22	0.00	0.00
50%	1.52	13.30	3.49	1.36	72.78	0.55	8.59	0.00	0.00
75%	1.52	13.83	3.61	1.62	73.05	0.61	9.14	0.00	0.11
max	1.53	17.38	3.98	3.50	75.41	6.21	16.19	3.15	0.37

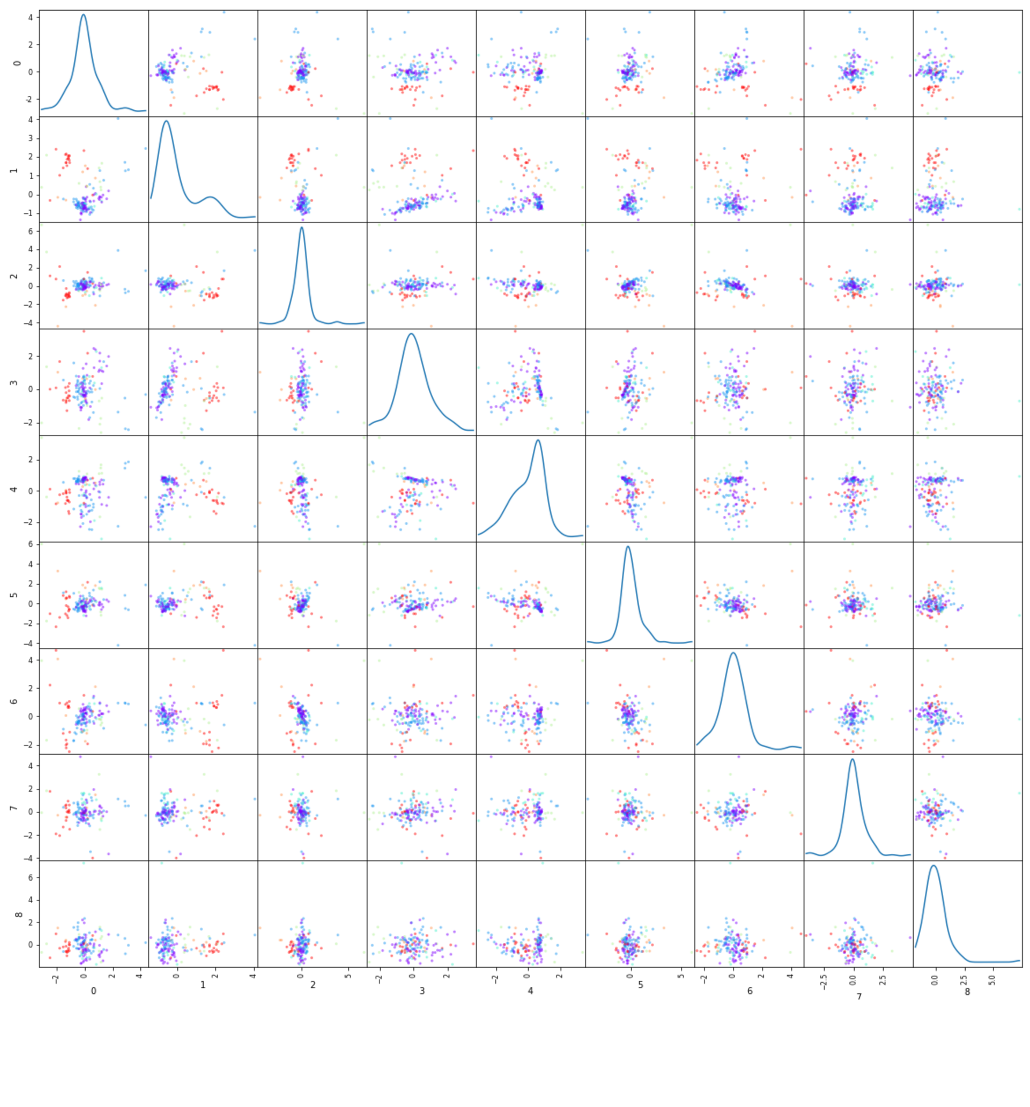
Each data point is labelled with one of 6 class labels. The distribution of these six classes is not balanced, the most often occurring classes being almost ten times as common as the least. For example, 53 samples belong to the class "Window from building (non-float processed)", whereas only 6 samples belong to the class "tableware". In fact, 68% of the data samples in the training set belong to only two of the six classes. The data points had been assigned an integer code to identify it's class. In the original data, these were 1,2,3,5,6,7. For this paper, we reassigned them with the codes 0,1,2,3,4,5 as shown in Table 2.

Table 2: Glass types, codes and counts

Glass Type	Original Code	New Code	Training Set Samples
Window from building (float processed)	1	0	49
Window from building (non-float processed)	2	1	53
Window from vehicle	3	2	12
Container	5	3	9
Tableware	6	4	6
Headlamp	7	5	20

When plotting the attributes in the data set, it was difficult to see any patterns that stood out, correlations, or any attributes that clearly separated the classes. Therefore we decided to perform Principal Component Analysis (PCA). PCA is a popular dimensionality reduction algorithm and that identifies the axis that accounts for the largest amount of variance in the training set (Géron). In this way, PCA can be used to represent a multivariate data set as a smaller set of variables in order to more easily identify trends or clusters. After performing PCA, the scatter plots of the components (Figure 1) were slightly easier to see patterns via visual inspection.

Figure 1: scatter plots of the PCA components



2 Model Descriptions and Implementations

This section will go over a brief explanation of each of the machine learning models used in this project. For the models that were implemented from scratch there will also be a thorough description of that implementation.

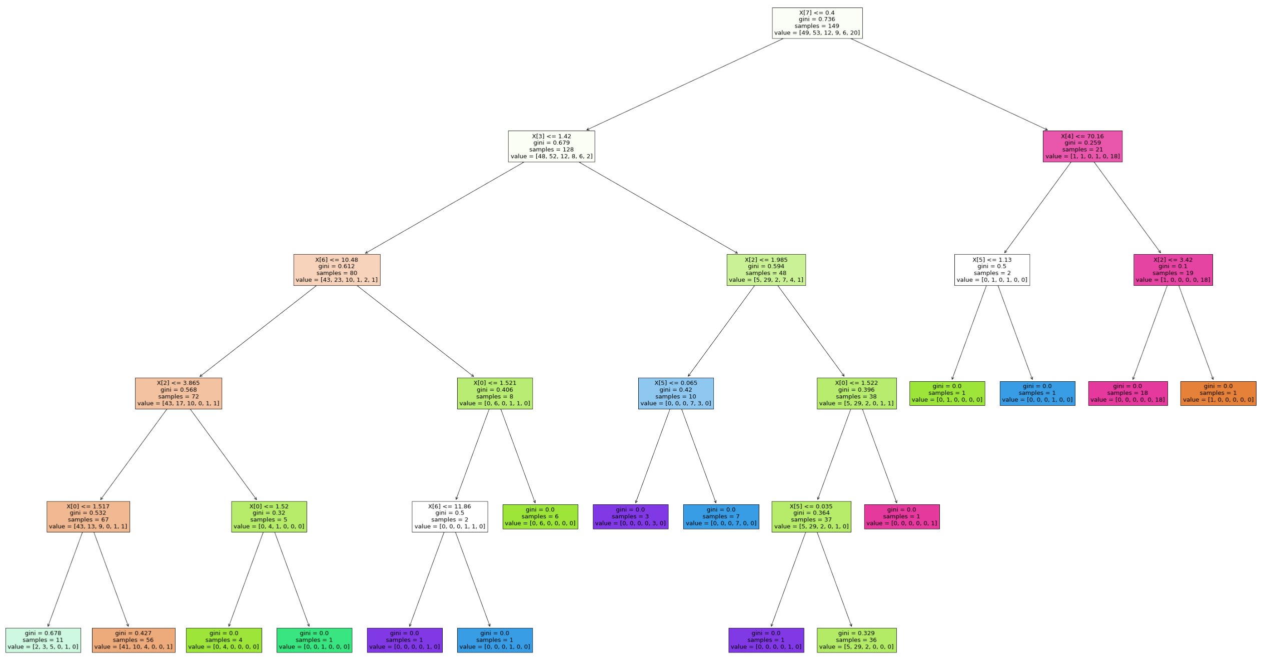
2.1 Decision Tree

A decision trees are a type of supervised machine learning where the data is continuously split according to a certain parameter. A decision tree is composed of two entities; decision nodes and leaf nodes. At each decision node, the data is split into two child nodes based on the defined parameter. The final node is not split, and is referred to as a leaf node.

We used gini impurity, the measure of total variance across the K classes (Gareth et al.), as the parameter to split each node. In training, the process of splitting the data continues until each leaf has the highest possible gini impurity. Once the model is trained, new data points can be put into the model, which will follow down the tree until reaching a final leaf node. The category assigned to that leaf node will be the predicted class for that data point.

See Figure 3 for a visualization of our decision tree using sci-kit learn. Here the tree has a depth of 5. The root node is the node at the top, and the first decision node, which is then split into two child nodes.

Figure 2: Decision Tree to depth 5



X[7] <= 0.4 gini = 0.736 samples = 149 value = [49, 53, 12, 9, 6, 20]
X[3] <= 1.42 gini = 0.679 samples = 128 value = [48, 52, 12, 8, 6, 2]
X[4] <= 70.16 gini = 0.259 samples = 21 value = [1, 1, 0, 1, 0, 18]
X[6] <= 10.48 gini = 0.612 samples = 80 value = [43, 23, 10, 1, 2, 1]
X[2] <= 1.985 gini = 0.594 samples = 48 value = [5, 29, 2, 7, 4, 1]
X[5] <= 1.13 gini = 0.5 samples = 2 value = [0, 1, 0, 1, 0, 0]
X[2] <= 3.42 gini = 0.1 samples = 19 value = [1, 0, 0, 0, 0, 18]
X[2] <= 3.865 gini = 0.568 samples = 72 value = [43, 17, 10, 0, 1, 1]
X[0] <= 1.521 gini = 0.406 samples = 8 value = [0, 6, 0, 1, 1, 0]
X[5] <= 0.065 gini = 0.42 samples = 10 value = [0, 0, 0, 7, 3, 0]
X[0] <= 1.522 gini = 0.396 samples = 38 value = [5, 29, 2, 0, 1, 1]
gini = 0.0 samples = 1 value = [0, 1, 0, 0, 0, 0]
gini = 0.0 samples = 1 value = [0, 0, 0, 1, 0, 0]
gini = 0.0 samples = 18 value = [0, 0, 0, 0, 0, 18]
gini = 0.0 samples = 1 value = [1, 0, 0, 0, 0, 0]
X[0] <= 1.517 gini = 0.532 samples = 67 value = [43, 13, 9, 0, 1, 1]
X[0] <= 1.52 gini = 0.32 samples = 5 value = [0, 4, 1, 0, 0, 0]
X[6] <= 11.86 gini = 0.5 samples = 2 value = [0, 0, 0, 1, 1, 0]
gini = 0.0 samples = 6 value = [0, 6, 0, 0, 0, 0]
gini = 0.0 samples = 3 value = [0, 0, 0, 0, 3, 0]
gini = 0.0 samples = 7 value = [0, 0, 0, 7, 0, 0]
X[5] <= 0.035 gini = 0.364 samples = 37 value = [5, 29, 2, 0, 1, 0]
gini = 0.0 samples = 1 value = [0, 0, 0, 0, 0, 1]
gini = 0.678 samples = 11 value = [2, 3, 5, 0, 1, 0]
gini = 0.427 samples = 56 value = [41, 10, 4, 0, 0, 1]
gini = 0.0 samples = 4 value = [0, 4, 0, 0, 0, 0]
gini = 0.0 samples = 1 value = [0, 0, 1, 0, 0, 0]
gini = 0.0 samples = 1 value = [0, 0, 0, 0, 1, 0]
gini = 0.0 samples = 1 value = [0, 0, 0, 1, 0, 0]
gini = 0.0 samples = 1 value = [0, 0, 0, 0, 1, 0]
gini = 0.329 samples = 36 value = [5, 29, 2, 0, 0, 0]

2.1.1 Decision Tree Implementation

Calculating Gini Impurity

When implementing a decision tree from scratch, we first need to know how to decide the "best" split in the data. This requires calculating the gini impurity value at every possible split. To do this we built a function that takes the data with a given split; divides the data into left and right child nodes dependent on that split.

The gini function goes through both sides and loops through each of the k classes, first calculating the probability of each class as p_k. The gini of each side is the sum of $p_k(1 - p_k)$ for each of the given classes.

$$Gini_{side} = \sum_{k=1}^{k} p_k(1 - p_k)$$

Where:
$p_k = \frac{k}{N}$
k is the number of data points in a given class
N is the number of data points on that side of the split

Finally, the gini scores for the left and right sides are weighted and added together to get the overall gini impurity score of that split. The score for the left side is multiplied by the number of data points on the left side, divided by the total in both sides. The respective happens for the gini score on the right.

$$Gini = G_{left} * \frac{N_{left}}{N} + G_{right} * \frac{N_{right}}{N}$$

The gini impurity function outputs the gini index that is used as the metric that decides the split, as well as the gini of the entire node and the total class counts of the input data. The latter two are used for comparison to the reference implementation model from sci-kit learn.

Finding The Best Split

In order to loop through the data to find the ideal split, we built a function called leaf_hunter. This leaf_hunter takes a data frame as input and outputs the best gini and corresponding attribute and threshold to make a cut, as well as the node gini and class counts.

Since this function will be called repeatedly by our decision tree class until the data is fully split, we check if the data frame of only 1 row. If there is only one row then it cannot be further split, and the function will return with a gini impurity of 0 and the class counts.

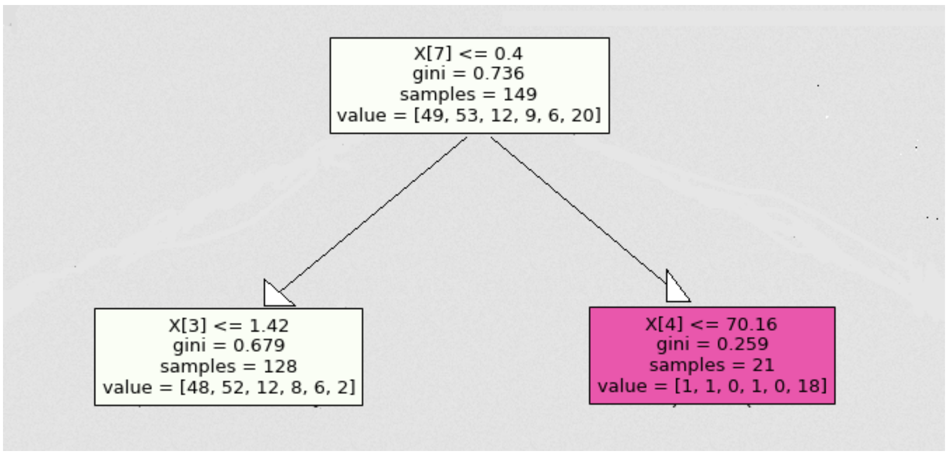
If the input data has more than 1 data point, the function will first loop through all the features. In this case, it will be looping through the 8 elements and the refractive index. Within each feature, the data will be sorted and then looped through, with a threshold chosen between each data point that is not equal. In other words, if there are two data points with the same value, the threshold will shift until there are two non-equal data points next to each other. The threshold will then be set to the halfway point between the two data points.

This chosen threshold, along with the data frame and class labels, will be given to our function to calculate the gini impurity at this specific split. The gini impurity was initialized to 1. If this split gives a better score, the results of the gini function will be saved.

The leaf_hunter function will therefore loop through each attribute, and through each possible split within each attribute, until the threshold is found that gives the best gini impurity score of the given data frame.

A closer look at our tree in Figure 3 shows the root node, where we can see that the model makes the first cut at the feature $X[7]$, which corresponds to the element Barium. The model splits the data into two sets: the glass fragments with a Barium value of ≥ 0.4 , and the ones with a Barium value 0.4. This creates two new nodes. One node has 128 samples, while the other has just 21.

Figure 3: A closer look at the first three nodes of the tree



Decision Tree Class

The decision tree class that we built initializes a node based on the input data frame. It then splits the node using the leaf_hunter function. Each side of the new split is initialized as a new node, which is again split. This process continues until the max depth is reached or only one class is left in each node, whichever happens first.

The decision tree class has a predict method. This method takes a test data frame as input and initializes an empty list to save the class predictions. A copy of the decision tree is made, and the copy is looped through, using the thresholds at each decision node to follow down the tree until a leaf node is reached, or a maximum depth. The class at this end node is then saved as the prediction for the given row in the data frame. Once all rows have been predicted, the method returns the predictions.

Closer Look at the Decision Tree in Action

One can see in Figure 3 that the root node of the tree contains the entire 149 data points from six different classes of glass from the training data. However, after just one cut the right child node has found 18 out of 20 “headlamp” type glass data points.

It appears that just by categorizing the glass fragments into two groups, one group with a given element value below a threshold, and one above the same threshold, we were able to find 90% of all headlamps in the training data set. This suggests that a high value of Barium is most likely a good indicator of headlamp glass. This is furthered by the fact that almost every non-headlamp glass had a Barium value of 0.4 or lower, and as such went to the other child node. Finding homogeneity this early on is a good indicator that the decision tree could indeed be a very useful method of predicting glass fragments.

Reference Implementation

To ensure correctness of our implementation, we compared the results of our premade decision tree to those of our from-scratch implementation. Manual inspection of the full-depth trees allowed us to confirm that the two models made the same splits at every node, and created identical trees.

2.2 FF Neural Network

A Feed Forward (FF) Neural Network is a supervised machine learning model inspired by the networks of biological nodes (Géron). The goal of a FF Neural Network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input x to a category y. A FF network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation (Goodfellow, Bengio, and Courville).

2.2.1 FF Neural Network Implementation

Inspiration for the design of our neural network was taken from a number of online sources (Loy; Kumar; Brownlee).

Layers

A neural network is built from many different components. It consists of several layers, each of which have nodes that connect to the adjacent layers' nodes. The first layer is called the input layer. This layer contains the values of the features from the data set. The number of nodes in this layer is equal to the number of features in the data, which in this case is 9.

After the input layer is another layer of nodes, called the hidden layer. The hidden layer is the layer between the input and the output layers, and there can be more than one hidden layer in a given model. Each layer can vary in the number of nodes it has.

First, we implemented a neural network class. An instance of the class is initialized with a matrix of the features X, the class labels y, and the number of nodes desired for the hidden layer. For simplicity we only implemented the class with a single hidden layer. The default is set to 10 nodes.

Weights and Biases

Next a set of weights and biases is needed, which are the parameters that the model will try to learn and optimize. This optimization occurs during the training part of the process. These parameters represent the importance of each feature in our data.

When an instance of the class is created, the weights and biases are also initialized as numpy matrices using `numpy.random.rand`. The dimensions of the weights for the first layer is the number of attributes *times* the number of nodes, which is 9x10 using the default of 10 nodes. The weights for the output layer is the number of nodes *times* the number of classes, which is 10x6 by default. The biases are initialized randomly, with one bias per node in each layer.

Activation Functions

We also need to choose a set of activation functions. These functions are what allow the model to be non-linear. Having a non-linear activation function allows the model to capture complex non-linearities and interaction effects which are most likely present in our data. (Gareth et al.).

An activation function is applied to every layer except the input layer. There are numerous different activation functions to choose from. For the input layer we implemented a tanh activation function, as it is common to use for FF neural networks. Tanh creates non-linearity by being calculated as follows:

$$Tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Output Layer

For the output layer we implemented a softmax activation function, which is required for multi-class classification. Softmax gives a probability for every possible class for each input value. Softmax is calculated as follows:

$$\sigma(s(x))_k = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$$

Where:
 K is the number of classes
 $s(x)$ is a vector containing the scores of each class for the instance x, and
 $\sigma(s(x))_k$ is the estimated probability that the instance x belongs to class k, given the scores of each class for that instance (Géron).

Computing Loss

Categorical cross-entropy is a loss function used in multi-class classification tasks, and is the one implemented in this project. It is computed as:

$$Loss = -\sum_{i=1}^{K} y_i * log\hat{y}_i$$

Where:
 $\hat{y}_i$ is the i^{th} score for each class in the model output
 y_i is the corresponding ground truth, and
 K is the number of classes.

Neural Network Learning Process

There are two parts of the neural network training process, feed forward and back-propagation.

Feed forward is the process of traversing the neural network's layers from input, through the hidden layer (or layers) and finally through the output layer to class prediction. The loss will then be calculated based on the model's predictions and the true values.

Back-propagation is the part of the process where the network learns. This is done by computing gradients and pushing them back throughout the network and updating the parameters of the model. At each stage, we calculate the gradient of the function and choose the direction of greatest change. This is then multiplied by the learning rate, which controls how large a "step" is taken at each epoch. The values held by the layer are then updated by the resulting amount. This continues until the output layer and all hidden layers are updated. Predictions can then be recalculated using these new values and the new result evaluated. This cycle continues until a predetermined limit is reached, i.e. the number of epochs, or if the loss becomes static indicating the model is no longer learning.

Reference Implementation

For the reference implementation we used keras from tensor flow. While this library implementation has additional options or parameters that can be modified, the model used for this project was built using the same parameters as the home-made version in order to better compare model performance.

Comparing the results of the two neural networks proved more difficult as the results were slightly different. Unlike the decision tree, neural networks are not as easy to manually inspect and compare. This difference in outcome was a bit surprising, as our neural networks should not be stochastic, and therefore should be reproducible. After all, neural networks are a series of matrix multiplications.

One reason for the discrepancy between models is due to the randomly generated initial weights and biases in both neural networks. Setting the seed in numpy for our from-scratch model was simple and the results are reproducible. However, keras has it's own method for seed setting, therefore we were not able to set a matching seed in order to get matching results hence a slight difference in outcome between the two models.

However, the reference and from-scratch models had consistently similar performance, validating that the inner workings of the model we implemented work as expected.

2.3 Gaussian Naive Bayes

Naive Bayes is a family of supervised learning methods based on applying Bayes' theorem, with the single assumption that features are independent within each class. There was no apparent correlation between the features, and many features seem to have a roughly Gaussian distribution. Naive Bayes is suitable for multi-class classification, and with independent features this method seemed appropriate for the task. We specifically chose Gaussian Naive Bayes, as this works with the assumption that features have a normal, or Gaussian distribution.

In addition, Naive Bayes assumption introduces some bias, but reduces variance, leading to a classifier that works quite well in practice as a result of the bias-variance trade-off (Gareth et al.). Since one of our other models was a decision tree, which have a tendency to over-fit and therefore have high variance, it seemed like a good idea to try a model that reduces variance in order to compare the results of these inherent limitations.

3 Hyperparameters and Training

In machine learning, hyperparameters are parameters whose values affect the learning process and determine the values of model parameters that an algorithm learns during training. This section will go over the hyperparameters that were chosen for each model as well as the process chosen for training the models.

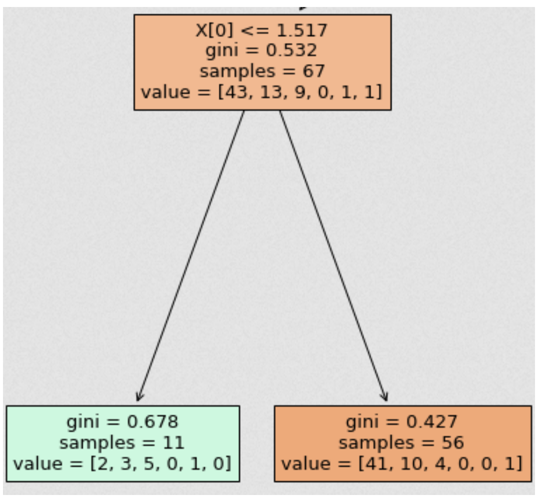
3.1 Decision Tree

The main hyperparameter to consider with a decision tree is the max depth, which is the number of levels down the tree can grow from the root node. As described above, each node will continue to split until there is only one class of data points in the node, or the max depth is reached, whichever comes first. If a decision tree is allowed to continue until all leaves contain only a single data point, the tree will likely be over-fitted to the training data and will not generalize well for the test data.

When setting a max depth of 5 on the training data, we can see that the model does not reach purity in most of the leaf nodes. Figure 4 shows two leaf nodes which contain multiple classes. We need to find out what the optimum depth is to predict as accurately as possible without overfitting.

In order to test a model to see how it performs, the data is often split into a training and validation set. However, this training set is very small and splitting it like this could possibly remove certain classes entirely from the training data. We also want to be able to train our model on as much data as possible, and removing a portion of the training data for validation purposes is likely to have a significant effect. One way to do this is to use k-fold cross validation, and we chose to use the sci-kit learn library to do this. For the decision tree, we ran k-fold several times with 2-5 splits, and trained models in each split to depths from 3 to 10, saving the F1-score. Having multiple test runs we were able to estimate that the best training F1-score occurs at a depth of 6. Therefore we chose a depth of 6 for our final model, which will be built on all the training data.

Figure 4: Example of a leaf node split



We also tested if PCA transformation would have an affect on the model performance. In the end there was little difference in F1-scores using PCA on the decision tree, however the tree needed to run to a deeper depth, so we used the non-transformed data for this model.

3.2 FF Neural Network

For the FF neural network we chose stratified k-fold from the sci-kit learn library in order to preserve the percentage of each class within each fold, and used the default of 5 folds.

The hyperparameters to decide upon are the learning rate, number of nodes in the hidden layer, and number of epochs. A manual grid search was carried out using k-fold cross validation for training in order to find the parameters that performed best. One experiment was done on the standardized data, and another done on the data that been transformed with PCA. The PCA-transformed data seemed to have better performance both in terms of F1-score and needing less epochs and so we opted to transform the test data for the final test. The final parameters chosen for our model were a learning rate of 0.01, 100 nodes in the hidden layer, and 1000 epochs during training.

3.3 Naive Bayes

Hyperparameters for Naive Bayes (NB) were found using manual testing and grid search methods. K-fold cross validation was used with 5 folds. NB uses class prior probabilities as a parameter. As default they are set to the proportion of each class that is in the data that is fitted to the model. First a model was trained with equal class priors, meaning that the prior probabilities are 1/6 for each of the 6 classes. A model was then trained using the prior probabilities based on the class distributions in the entire training set. It was thought that forcing the priors to be equal might help improve the model performance. However, the results in training indicate that the default priors were the best option.

After manually testing, a grid search method was employed to identify the best option for the parameter var_smoothing, using a train/test split to validate. Code for the grid search method was inspired from online sources (Jain). This parameter can be used to account for samples that are further from the distributed mean. With the default being quite low, it seemed as though a higher value could help improve performance of the model. The default value is $1e^{-9}$, the manual optimal value found is $1e^{-1}$, and the best from grid search is 0.231. Using K-fold validation again, the manual value gave better results on the training data, likely because it had more variety in training data.

4 Results

The performance of the three methods of classification will be evaluated on their precision, recall and F1-score. All the methods have been been trained on the same data, yet their results vary greatly when being given the test data.

Precision is defined as the true positives divided by all the positives (true and false). This number tells us how many positives are correctly predicted. It ranges from zero to one, one being the best case, where not a single false positive has been made.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

Recall tells us about the relation between captured positives and actual positives in the data. Recall is considered an important metric in cases where there is a high cost associated with false negatives.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

Lastly, F1-score is a function of the two previous measurements. The F1-score is a harmonic mean between precision and recall, and will therefore usually return a value between the two. Like the two previous metrics, this one will also always return a number between zero and one.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Lastly, an average of all these scores is calculated for each metric. In the case here weighted average is calculated, which takes into account the number of true instances of the class when computing the average. Features with a low sample size will have a corresponding lower weight in the final result.

For each model, a confusion matrix was created. A confusion matrix shows the summary of prediction result on the classification. The number of correct and incorrect predictions are summarized with count values and broken down by each class. The diagonal from bottom-left to top-right shows the correctly identified samples, and all others are incorrectly categorized.

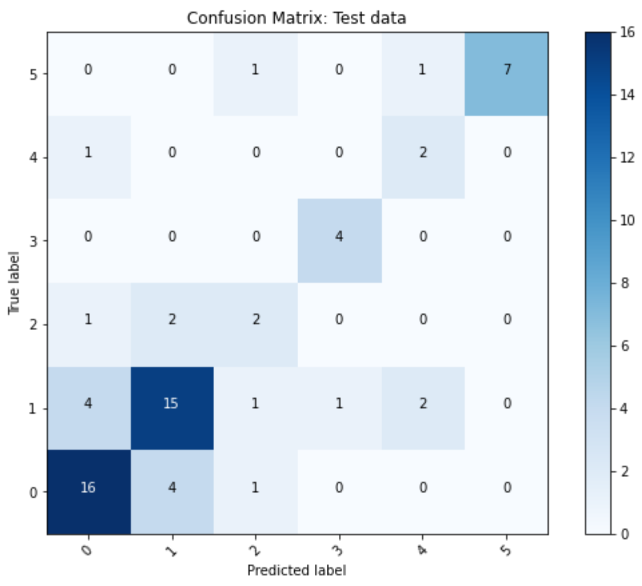
4.1 Decision Tree Results

The confusion matrix in Figure 5 shows the results of running the test data through our trained decision tree. The code for the confusion matrix was borrowed from exercise notes at ITU (Grbic). We can see that the model identifies most of the first two categories correctly. In addition 7 of 9 "Headlamp" fragments were correctly categorized, and no other classes were incorrectly assigned to this class, giving it a perfect precision score.

The recall, precision and F1-scores for all classes are summarised in Table 3 below.

Overall, this model correctly predicts 46 samples, and misclassifies 19. The most common misclassifications fall around classes 0 and 1, which are the two types of window glass, float and non-float processed. However the method actually did relatively well at distinguishing these two types of building window glass, compared to the other methods.

Figure 5: Confusion Matrix of the Decision Tree test data predictions



True \ Predicted	0	1	2	3	4	5
5	0	0	1	0	1	7
4	1	0	0	0	2	0
3	0	0	0	4	0	0
2	1	2	2	0	0	0
1	4	15	1	1	2	0
0	16	4	1	0	0	0

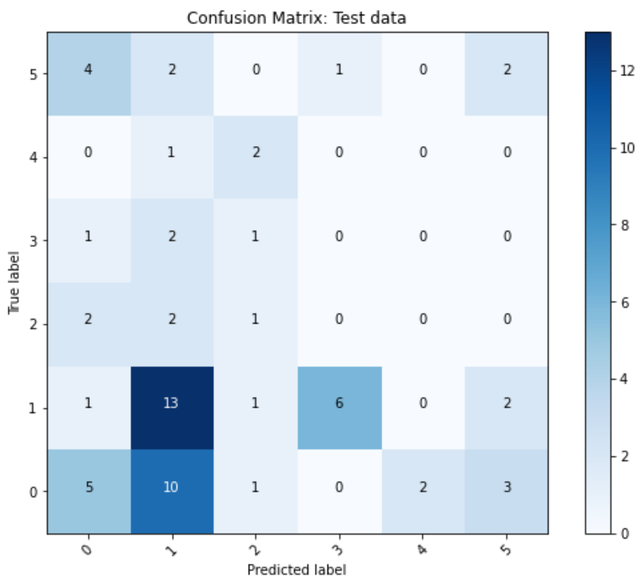
Table 3: Scores for Decision Tree predictions

Glass Class	Recall	Precision	F1
Window from building (float processed)	0.76	0.73	0.74
Window from building (non-float processed)	0.65	0.71	0.68
Window from vehicle	0.40	0.40	0.40
Container	1.0	0.8	0.90
Tableware	0.67	0.40	0.50
Headlamp	0.78	1.0	0.88
Overall	0.71	0.67	0.71

4.2 FF Neural Network Results

The confusion matrix in Figure 6 shows much more mixed results. The model fails to predict correctly any samples of classes 3 and 4, and misclassifies many others.

Figure 6: Confusion Matrix of the scratch FF-NN test data predictions



True \ Predicted	0	1	2	3	4	5
5	4	2	0	1	0	2
4	0	1	2	0	0	0
3	1	2	1	0	0	0
2	2	2	1	0	0	0
1	1	13	1	6	0	2
0	5	10	1	0	2	3

Table 4 shows the recall, precision and F1-scores for this model.

Table 4: Scores for scratch FF-NN predictions

Glass Class	Recall	Precision	F1
Window from building (float processed)	0.38	0.24	0.29
Window from building (non-float processed)	0.43	0.57	0.49
Window from vehicle	0.17	0.20	0.18
Container	0.00	0.00	0.00
Tableware	0.00	0.00	0.00
Headlamp	0.29	0.22	0.25
Overall	0.33	0.32	0.32

The neural network correctly predicts 21 out of 65 samples. This corresponds to a success rate lower than one in three overall, which is far from an impressive result. It is most accurate at predicting class 1, but still only correctly identifies 13 out of 23 samples. It misclassifies 17 further samples as this class though, giving it a precision of 0.57.

4.3 Gaussian Naive Bayes Results

The confusion matrix shown in Figure 7 reveals that the Gaussian Naive Bayes model performs slightly better than the FF-neural network, but still performs fairly poorly. There are a lot of misclassifications and it does not seem to follow a particular pattern, indicating that this model may not be the best method for classification in this situation.

The model is bad at distinguishing the two kinds of windows glass types from each other, predicting the vast majority of the non-float processed as float processed type. It was, however, good at finding glass of the headlamp variety, with only 1 in the category being misclassified. Even this isn't very impressive as this class is quite unique. Overall the model performs well when detecting headlamp glass, but performs poorly otherwise.

Figure 7: Confusion Matrix of the Gaussian Naive Bayes test data predictions

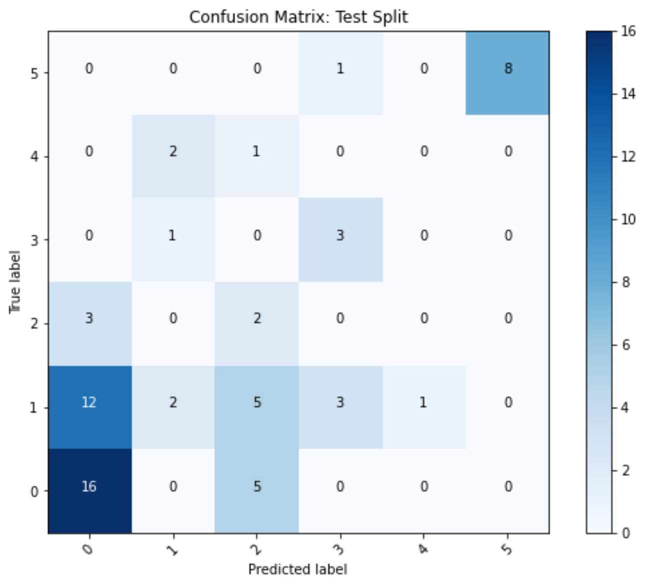


Table 5 shows the recall, precision and F1-scores for this model.

Table 5: Scores for Gaussian Naive Bayes predictions

Glass Class	Recall	Precision	F1
Window from building (float processed)	0.52	0.76	0.62
Window from building (non-float processed)	0.40	0.09	0.14
Window from vehicle	0.15	0.40	0.22
Container	0.43	0.75	0.55
Tableware	0.00	0.00	0.00
Headlamp	1.00	0.89	0.94
Overall	0.48	0.48	0.43

5 Conclusion and Future work

The decision tree model is the best model to use for this data set and classification problem, out of the three models tested in this report. Given the presented scenario of identifying glass fragments found at a crime scene, we would not feel confident in using the predictions from these models as evidence in a criminal case.

While appropriate for multi-class classification tasks, neither a neural network or naive Bayes performed well for this particular data set. It could be that the data set is simply too small or unbalanced in both cases. Future work could attempt to balance out the data set using bootstrapping before training the model, or an ensemble method with bagging to see if that boosts performance. Gathering more samples for training the models would likely improve the models performance.

Gaussian Naive Bayes assumes the data follows a normal distribution. In this data set, most the features appeared to follow a normal distribution after PCA transformation, however this did not seem to make the assumption hold. In general, performing PCA transformations made little difference to the performance of any of the models.

Given a training data set of only 149 records, the decision tree is able to outperform all other models overall. Each decision the tree makes is easy to see and understand, and the performance metrics are overall fairly high.

5.1 Other Models

Given the performance of a single decision tree, future work could look into implementing a random forest to see if performance can be improved. A random forest is a bagging algorithm that operates by creating an ensemble of decision trees. When growing an individual tree each node will consider only a subset of the features, and will then consider the average predictions of all the trees. The bagging aspect could potentially help with the data set being unbalanced, thereby improving the performance of predicting the under-represented classes.

References

Evett, Ian W, and Ernest J Spiehler. "Rule induction in forensic science". In *Knowledge Based Systems*, 152–160. 1989.

Gareth, James, et al. *An introduction to statistical learning: with applications in R.* Spinger, 2013.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

Géron, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019.

Grbic, Djordje. *Lecture notes in Machine Learning*, Oct. 2021.

— . *Lecture notes in Machine Learning*, Nov. 2021.

Brownlee, Jason. "How to Code a Neural Network with Backpropagation In Python (from scratch)". Visited on 2021. `https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/`.

Dua, D., and C. Graff. "UCI machine learning repository." Visited on 2019. `http://archive.ics.uci.edu/ml`.

Jain, Kopal. "How to Improve Naive Bayes?" Visited on 2021. `https://medium.com/analytics-vidhya/how-to-improve-naive-bayes-9fa698e14cba`.

Kumar, Niranjan. "Building a Feedforward Neural Network from Scratch in Python". Visited on 2021. `https://hackernoon.com/building-a-feedforward-neural-network-from-scratch-in-python-d3526457156b`.

Loy, James. "How to build your own Neural Network from scratch in Python". Visited on 2021. `https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6`.