IT UNIVERSITY OF COPENHAGEN

Machine Learning

Forensic Identification of Glass Fragments

Authors:
Jonas-Mika Senghaas (jsen@itu.dk)
Ludek Cizinsky (luci@itu.dk)
Lukas Rasocha (lukr@itu.dk)

BSMALEA1KU

1 Introduction

Glass is a material that is prominently part of criminal investigation processes. When a suspect is apprehended for a crime involving shattered glass, it is a standard procedure to submit particles of his clothing to a forensic science laboratory, in order to determine whether or not evidentiary material is present. However, even in the case where glass particles were detected, it often remains unclear whether those particles are connected to the crime. On this basis, the goal of this project was to build a machine learning model, that is able to classify different types of glass fragments based on their elemental composition and refractive index (RI).

2 Data Description and Preprocessing

2.1 Fundamental Data Description

The data used within this project was obtained from a study carried out as part of a research program for the *UK Forensic Science Service* [1]. The data set contained a total of 214 glass fragments that were obtained in a pre-split of 149 training and 65 test samples. For each glass fragment, a total of nine features were recorded, including a measure of the refractive index (RI), which describes how fast light travels through a material. It is a standard measure in glass analysis as it varies significantly for different types of glass. The remaining eight measured features described the chemical composition of the glass fragment in percent. Table 1 lists all measured features.

RI	**Na**	**Mg**	**Al**	**Si**	**K**	**Ca**	**Ba**	**Fe**
Refractive Index	Sodium	Magnesium	Aluminium	Silicon	Potassium	Calcium	Barium	Iron

Table 1: Observed Features for Glass Fragments

Each glass fragment in the data set belonged to one of six classes. In the data the classes were integer-encoded and mapped to the respective glass types depicted in Table 2.

Integer Code	Glass Type
1	Window from building (float processed)
2	Window from building (non-float processed)
3	Window from vehicle
5	Container
6	Tableware
7	Headlamp

Table 2: Glass Types and their Encoding

2.2 Data Cleaning

All data integrity checks carried out, such as checking for missing values, checking for the ranges of each feature, and adding up the percentages in the chemical composition in all glass fragments, did not report any major inconsistencies. Thus, no further data cleaning was necessary.

3 Exploratory Data Analysis

3.1 Class Distribution

Figure 1 shows the class distribution in the training and test split. The distribution of classes were roughly the same amongst the two splits. This was important to notice, since evaluating a classifier that was trained on data with a class distribution highly different from the testing set, would result in an inaccurate estimations for the out-of-sample performance. Apart from that, a strong imbalance in the six observed classes became obvious. While the two majority classes 1 and 2 made up more than 65% of the entire data, the underrepresented classes 3, 5 and 6 all combined accounted for less than 25% of all data points. This imbalance might pose challenges when it comes to building a classifier correctly classifying the minority classes.

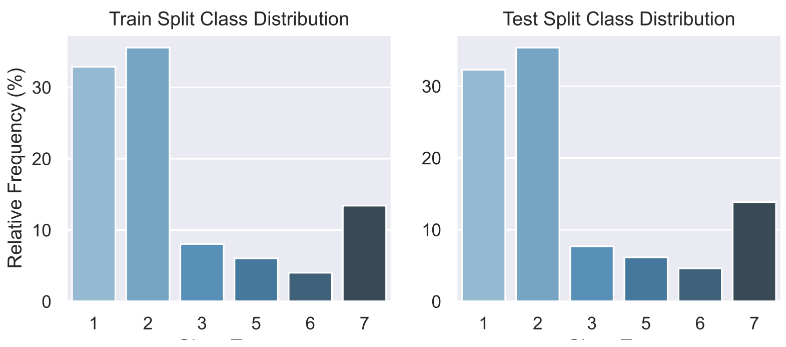

Figure 1: Class Distribution in (a) Training Split and (b) Test Split

3.2 Distribution of Features For Each Class

Figure 2 displays the distribution of each feature in each class, which gives an intuition of how well each feature separates the classes.

The nine plots show that the different features are differently well-suited for classifying the glass fragments. Features such as *silicone* do not seem to perform well, as there is little variation in the distribution of the feature in the six glass types. On the opposite, features such as *calcium*, *barium*, *magnesium* or the *RI*, seem to be better suited, as they separate subsets of types of glass fragments. However, it can be summarised, that there does not exist a single feature, that can be used to perfectly separate the classes. It is therefore likely that the classifier will need to be trained on a combination of features.

3.3 Principal Component Analysis

Figure 3 visualises the first two principle components of the data set. The visualisation provides an understanding of the difficulty of the classification problem.

It is evident, that the classification problem is not trivial. The first two principle components reveal, that glass types for similar real-world purposes, ie. class 1, 2 and 3 all being windows, have related feature distributions and are thus overlapping in the first two principle components. This is likely to pose a challenge to the classifier. In contrast, class 7 (headlamp) is well-separated and is therefore expected to be easier to classify correctly.

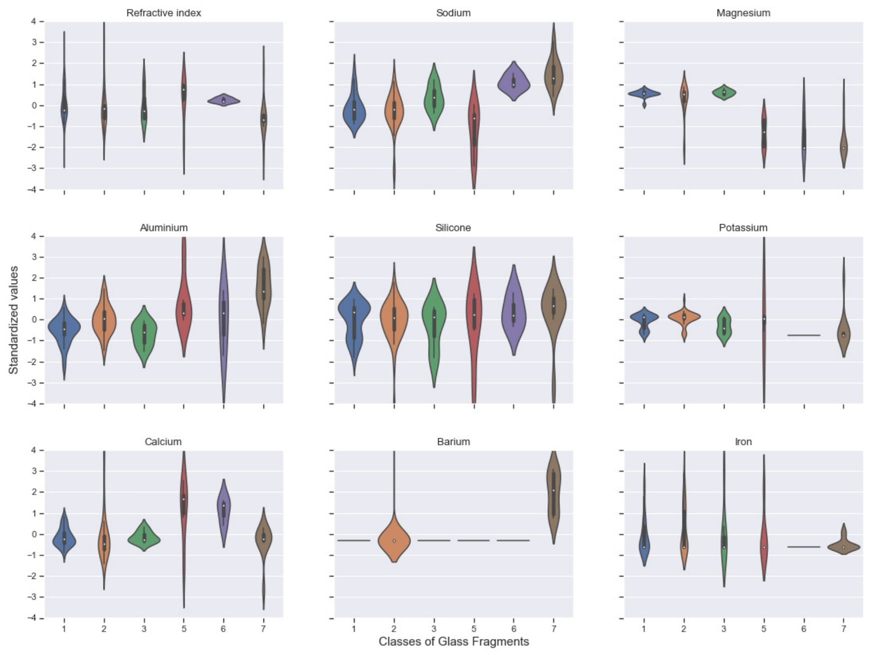

Figure 2: Standardised Distribution of Features for each Class

Another challenge might be the under-representation of classes 3, 5 and 6. It will be difficult for the model to learn the patterns in the data for these classes, especially in the case of class 3 being in the middle of the point clouds of the majority classes 1 and 2.

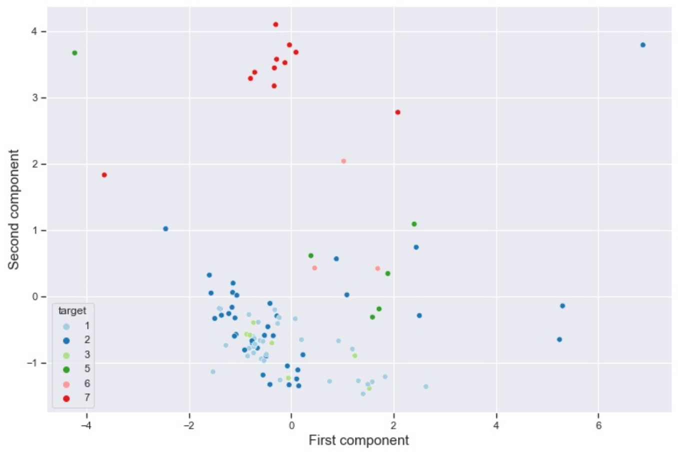

Figure 3: Scatter Plot of first two Principal Components of Training Split

4 Implementation of Classifiers

4.1 Decision Tree

4.1.1 Details on Implementation

The decision tree classifier was implemented using three different classes:

1. `Node()` - Low level data-structure used to represent nodes in a decision tree
2. `DecisionTree()` - Decision tree parent class which implements the building process of a tree (fitting)
3. `DecisionTreeClassifier()` - Client-facing constructor for setting up the model

The `Node()` class is a data structure holding information about a single split at any level in a decision tree. There are three types of `Node` objects. A node can be of type (`self.type`) `root`, `internal` or `leaf`. Root and internal nodes are constituted in the same way. They hold the indices of the subset of the original training data that are considered for splitting at the node (`self.values`) and information about which feature index (`self.p`) provides the best cut point s (`self.val`) to partition the subset into two nodes. These two nodes are then pointed to in the `self.left` and `self.right` attributes. Using this linked structure, the decision tree can be traversed until a leaf node is reached for classifying data points. Leaf nodes do not contain any pointers to other nodes and thus also do not define a best-split. Instead, they save information about the decision being made in that leaf (decision region), which is saved in the `self.predict_proba` and `self.predict` attributes.

The `DecisionTree()` is the core class as it implements the construction (fitting) of the decision tree as well as prediction of classes for the given data. A call to `.fit()` with a feature matrix X and a target vector y representing categorical classes initialises the root (`self.root`) to a `Node` instance with all indices in the training data and then recursively splits the node until full purity or a stop-criterion is reached. The recursive call to `.split()` first finds the best possible split through the private, helper function `._best_split()` for the data in the node to be split. The function returns the index of the feature and the corresponding value for which the weighted loss (impurity measure) was minimal, alongside the weighted loss itself. All are stored in the current node to be split. Finally, the indices of the data points in the left and right child (found from the best split) are computed and initialised as `Node` objects. If the resulting child is pure (ie. only contains data points of the same class) or a stopping criterion is reached, the recursion ends and the node is turned into a `leaf` node. Otherwise, the `.split()` function is recursively called on the children of the former node. On a call to `.predict()`, the fitted tree is traversed until a leaf node is reached. The decision tree predicts to the majority class in the leaf node (`self.predict` attribute of leaf).

4.1.2 Correctness

In theory, a decision tree classifier is able to overfit any unique training data to 100% training accuracy. It was therefore tested, whether the custom implementation has this property. Figure 4 shows the training and test accuracy on a random 70-30 split of the *Iris Dataset* with deleted duplicate data points for the custom `DecisionTreeClassifier()` trained on the first two features in the data and stopped at increasing depths (from 1 to 20). As expected, the training accuracy continuously increases, since for each added depth of training the purity of the leaves increases. A decision tree with maximum depth of 11 is enough to entirely overfit the data and result in 100% training accuracy. Overall, the implementation behaves exactly as expected.

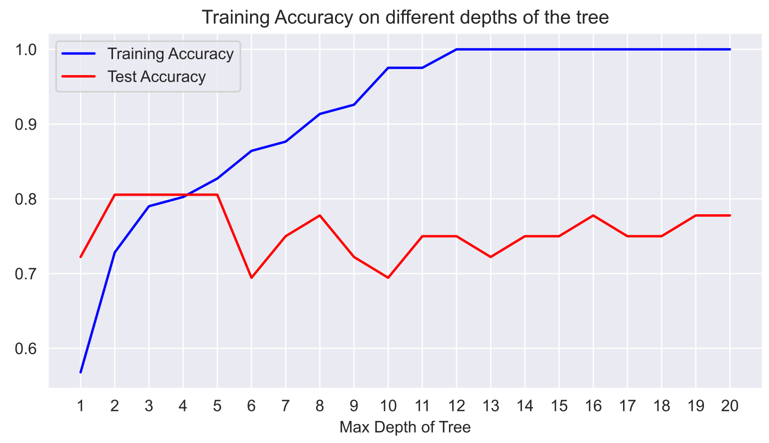

Figure 4: Training and Test Accuracy on Iris Data (Feature 1 and 2) for increasing *Max Depth* Hyperparameter

Figure 5 depicts the 2d-decision regions constructed for decision trees trained to different maximum depths on the *Iris* and *Circles Data sets* in row 1 and 2 respectively. The left-most plots show the decision regions for a tree trained to depth 1. This means, only a single split is allowed. With larger maximum depths, the models learn more and more of the variation (and noise) in the training data and finally overfit the training data to 100% training accuracy. Again, the decision regions constructed are sensible and conform with a reference implementation of *sci-kit learn*.

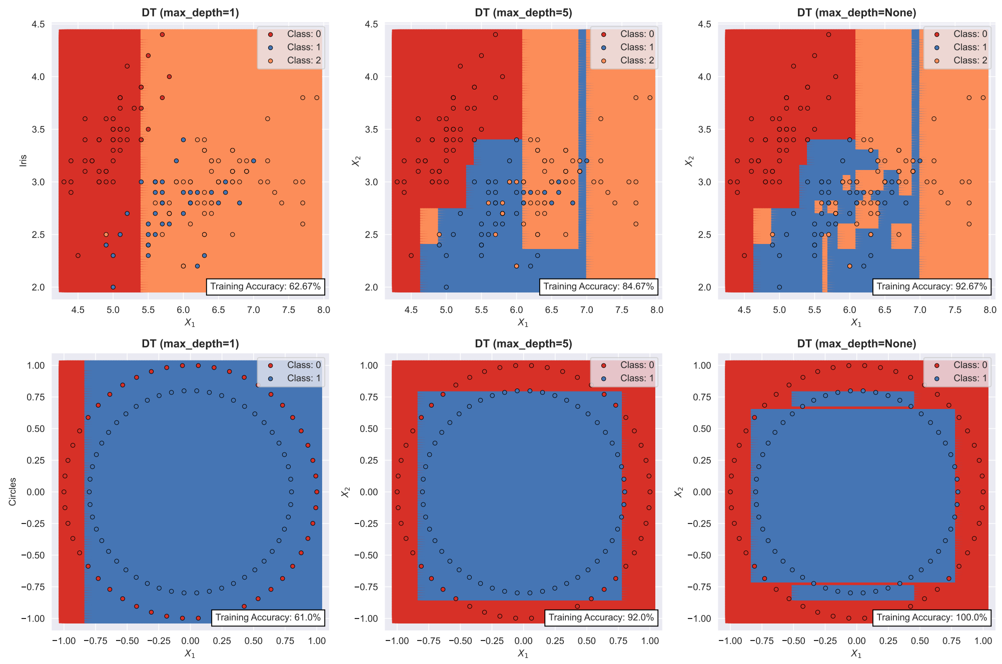

Figure 5: Performance of Decision Tree on Iris (Row 1) and Circles (Row 2)

4.2 Neural Network

4.2.1 Details on Implementation

The NeuralNetworkClassifier() was implemented using three different classes:

1. Var() - Low-level numeric data type with auto-differentiation (adapted from *Nanograd* [2])
2. DenseLayer() - Fully-connected layer within neural network
3. NeuralNetworkClassifier() - Client-facing class for constructing, training and predicting from neural network

The Var() class is low-level numeric data type. It allows for basic numeric operations (such as addition and multiplication) and auto-differentiation. This means, that through storing information about how a Var instance was created via its parents attribute (self.parents), the gradient of any Var instance can be computed.

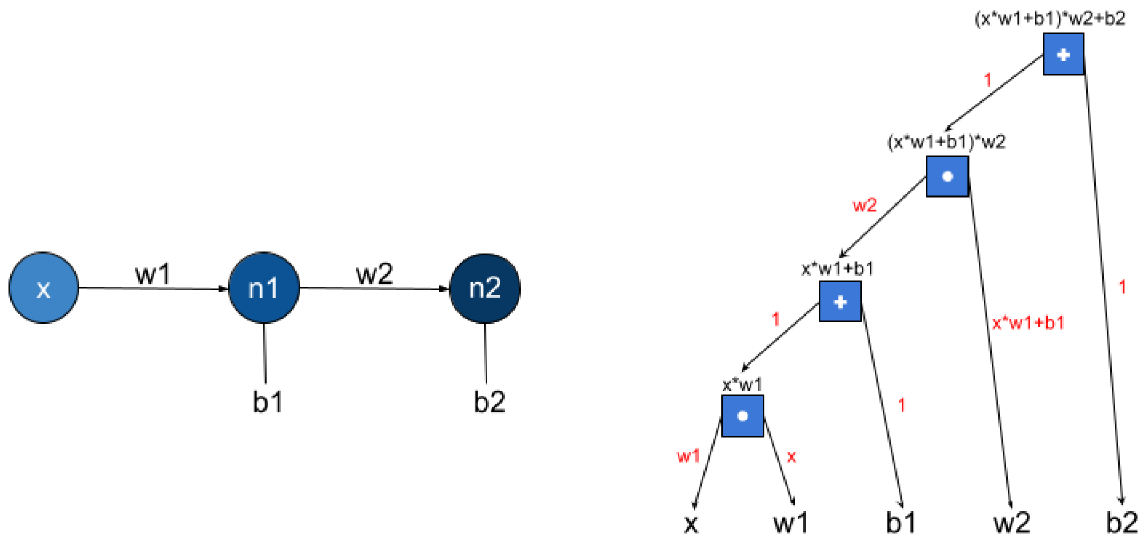

Figure 6: A simple network with one feature and one hidden layer (no activation functions) along with a computational dependency tree

Figure 6 demonstrates the Var's utility of a forward pass through a highly simplified neural network, where the output n_2 is computed through forward passing x as $n_2 = (xw_1 + b_1)w_2 + b_2$. The right side of the figure shows a dependency tree of this computation, where each node is Var instance. Red values denote the partial derivative of a given child with respect to its parent. Finally, to compute partial derivatives of any node with respect to any of its descendants, the tree is traversed reversely from the node to the descendant and the product of edge weights is the final partial derivative.

This algorithm is a translation of the chain rule. For instance, the partial derivative of n_2 with respect to w_1 is computed from the Var tree as:

$$\frac{\partial n_2}{\partial w_1} = 1 \cdot w_2 \cdot 1 \cdot x$$

The Var() then implements a method self.backward(), which traverses the dependency tree by computing the partial derivative for each node. The partial derivatives are then assigned to the attribute self.gradient in the respective nodes.

The `DenseLayer()` class serves as a building block for the `NeuralNetworkClassifier()`. The class is being initialised with an input and output dimension, as well as an activation function. On initialisation, a weight matrix (`self.weights`) is built based on the provided number of input and output dimensions along with a vector of biases (`self.biases`). The weights and biases are initialized from a uniform distribution $[0, 1)$ and each element is converted into a `Var` instance for the back-propagation to work. The core functionality of the `DenseLayer()` class is to feed forward a set of data points through the layer. The `self.forward` method takes a matrix X of `Var` instances. This matrix corresponds either to the output of the previous layer within `NeuralNetwork()` or to the feature matrix if the dense layer is the first layer. Finally, it computes the forward pass (using `self.weights`, `self.biases` and `self.activation`) and returns a matrix of `Var` instances - the activation of neurons in that layer. The operation is vectorised for performance reasons.

The `NeuralNetworkClassifier()` serves as a wrapper which puts together the preceding two classes and is used to fit the parameters (weights and biases) of each layer. The class is initialised by inputting a list of layers (`self.layers`), which are all instances of the `DenseLayer()` class, and the type of a loss function to be used for training the model (`self.loss`). To train the network, the method `self.fit()` with a feature matrix X and target vector y needs to be called. In addition, it is possible to specify training hyper-parameters including the *number of batches*, the *number of epochs* and the *learning rate*. In the training loop, for each epoch all batches are fed forward through the network and their respective losses are propagated back to update the weights in the network. Once the neural network is fitted, the `self.predict()` method can be used to feed forward data points and get the corresponding predictions from the neural network.

4.2.2 Correctness

The Universal Approximation Theorem [3] states that a neural network with a single hidden layer is able to approximate any continuous function. Figure 7 shows the loss and training accuracy history of the custom neural network implementation trained for 100 training epochs on the Iris Dataset (duplicate data points were removed). The neural network used contained a single hidden, ReLu activated layer of 20 neurons and trained for ten batches per epoch with a learning rate of 0.01. The visualisation of the training history confirms the model's implementation. Due to batch learning, the loss history is slightly choppy, nonetheless the loss is continuously decreasing and maximising the training accuracy to 99% training accuracy. Overall, the implementation behaves as expected.

Figure 8 depicts the 2d-decision regions constructed for simple neural networks trained for different amounts of epochs on the *Iris* and *Circles Data sets* in row 1 and 2 respectively. The same network architecture was used for both data sets. It is evident, that the neural network is able to learn patterns from the training data.

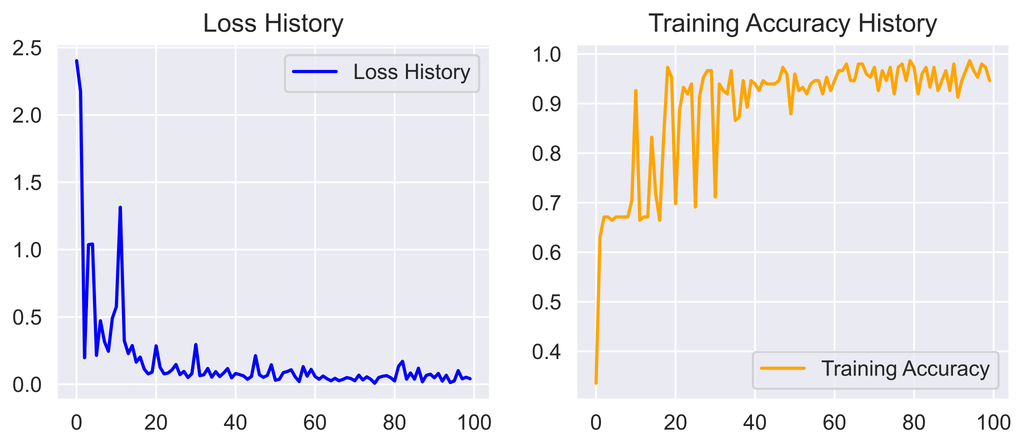

Figure 7: Overfitting Iris Dataset with Custom Neural Network Implementation

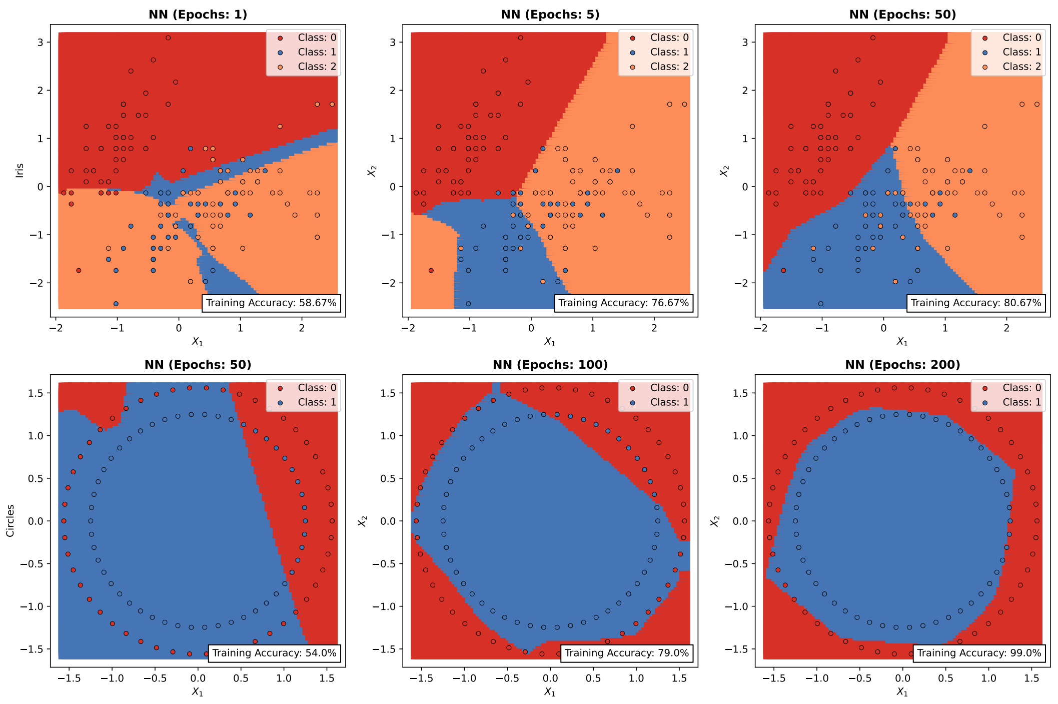

Figure 8: Performance of Neural Network on Iris (Row 1) and Circles (Row 2)

5 Details on Machine Learning Methods

5.1 Decision Tree

5.1.1 Introduction

Decision trees are a statistical learning method used both in the classification and regression setting. They are non-parametric models, meaning that they don't approximate a function $\hat{f}$ to describe the relationship between the features and target, but are classifying data by inferring a hierarchical decision structure (the decision tree) [4].

Decision trees are based on the idea of segmenting the feature space into a number of simple regions (rectangles in 2-dimensional space, cubes in 3-dimensional space, etc.). The challenge of tree-based learning methods is to identify the regions that best separate the classes. Mathematically, a loss function, ie. cross-entropy, gini-impurity or 0-1 loss, should be minimised for each data point in the training split, st.

$$\min \left\{ \sum_{j=1}^{J} \sum_{i \in R_J} \text{loss}(y_i, \hat{y}_{R_j}) \right\}$$

However, it is computationally infeasible to consider every possible partition of the feature space into J regions and evaluate the loss. For this reason, a top-down, greedy approach, known as recursive binary splitting is used to construct the decision tree in a computationally light fashion.

Recursive binary splitting is a recursive algorithm that splits a given partition of the training set (node) into the two (hence the name *binary* splitting) partitions that best separate the classes in the original partition. It thus finds the cut point s for a feature X_i that forms two sets $\{X|X_i < s\}$ and $\{X|X_i \geq s\}$ that minimise the weighted loss of both regions. The decision tree is then built by recursively applying this algorithm, until some stopping criterion is reached, or all regions are pure (ie. all training samples in a decision region are of a single class). The final prediction is based on identifying which region a new data point belongs to and then predicting the majority class in that region.

5.1.2 Decision Tree Model Selection

The final decision tree classifier is a pipeline of a `StandardScaler` instance, that was fitted on the training split and a `DecisionTreeClassifier` instance. For the scope of this project a combination of the hyper parameters `criterion`, `splitter`, `max_features` and `max_depth` of the tree were checked within a grid search.

The grid search ran over all possible combinations of hyper parameters defined. For each setting it fitted five models on four (randomised) folds and validated the performance of the model on the remaining split. It then computed the averaged accuracy score on the validation splits that occurred within the training and returned the model configuration that maximised the performance of the model on the validation splits. The best generalising decision tree was found to use *entropy-impurity* as a loss-function, is trained to maximum depth of five and considers eight features for each split.

5.1.3 Decision Tree Performance Evaluation

The best generalising model achieved an averaged accuracy score in the cross-validation splits of 70%, which is the estimated performance on unseen data. Because of that, it is surprising to see a test accuracy of only 62%. The variation in the two metrics is likely occurs, because both the training and test split are rather small, making it prone to random variation. From the classification report it becomes clear that the decision tree struggles with separating classes 1, 2 and 3 from each other. In contrast, the model does rather well on the more separated classes 6 and 7 with an F1-Score of 80% each.

Evaluation Metric	Score
Training Accuracy	87%
Validation Accuracy (during CV)	70%
Test Accuracy	62%

(a) Decision Tree Accuracy Scores on Training, Validation and Test Split

Class	Precision	Recall	F-Score	Support
1	0.68	0.62	0.65	21
2	0.64	0.61	0.62	23
3	0.25	0.20	0.22	5
5	0.33	1.00	0.50	4
6	1.00	0.67	0.80	3
7	1.00	0.67	0.80	9
accuracy			0.62	65
macro avg	0.65	0.63	0.60	65
weighted avg	0.67	0.62	0.63	65

(b) Decision Tree Classification Report on Test Split

Table 3: Decision Tree Performance Evaluation

5.1.4 Visualization of Decision Tree

Figure 9 shows a visualisation of the constructed decision tree. At each level of the tree the node carries information about the decision rule applied at the specific node, the loss-value (ie. gini impurity), the number of samples considered and the majority class in the node.

One interesting observation is that the feature *barium* is used at depth 2 to separate a pure leaf with class 7 (headlamp) off. This was expected from inspecting the feature distribution in each class in Section 3, since the feature distribution of class 7 was distinct for the feature *batrium*.

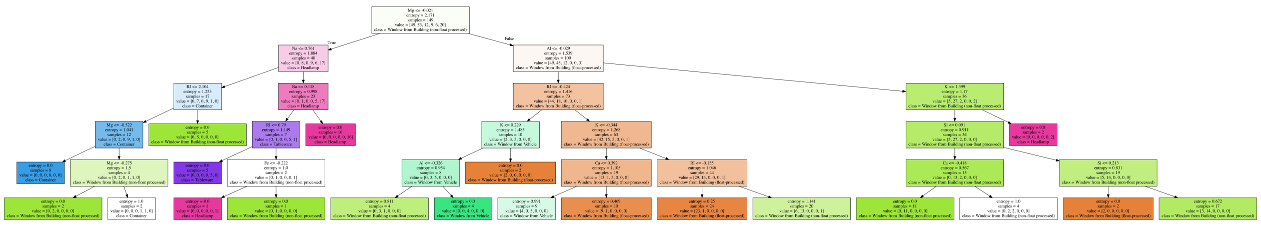

Figure 9: Visualised Decision Tree

5.2 Neural Network

5.2.1 Introduction

A feed-forward neural network is a machine learning algorithm that constructs a nonlinear (decision) function $\hat{f}(X)$ that tries to approximate the true relationship between some feature matrix X and a target vector y [4]. In each layer of the neural network, an activation function is applied to ensure the non-linearity of the decision function $\hat{f}$.

To train a neural network a loss function (e.g. cross entropy) is minimised using the technique of back-propagation. The algorithm computes the gradient of the loss with respect to all the weights and biases. This gradient is then used to tweak the parameters to reduce the error using a gradient descent step. The process is repeated until the neural network converges to the desired solution [5].

When dealing with a classification problem the response y is a vector produced by the last layer (the output layer) with a soft-max function as its activation, ensuring that the sum of elements is one $(\sum_{i=1}^{i=k} y_i = 1)$ and each respective element is in the range of 0 to 1 $(0 \geq y_i \geq 1 \;\; y_i \forall y)$. Therefore the individual scalars of the vector y can be interpreted as probabilities that a data point that was fed forward through the network belongs to the different classes. The prediction is then made to the class with the largest probability.

5.2.2 Neural Network Model Selection

Since the custom `NeuralNetworkClassifier` is not able to reproduce the exact results of other deep learning libraries, such as *Pytorch* or *Keras*, due randomness in the initialisation of weights, different optimisation algorithms and a different technique of back-propagation, this section will be split into model selection for the custom implementation and using the Python deep learning framework *Keras* that is built on top of *Tensorflow*.

Custom Implementation. Since the training process for the custom `NeuralNetworkClassifier()` implementation is relatively slow due to the recursive back-propagation, the model selection process was limited. Too complex network architectures as well as too high number of training iterations were not feasible, as well as an exhaustive grid search to tune hyper parameters. The model selection was therefore primarily based on heuristics and trial-and-error. Since the custom implementation does not allow to evaluate the model on a validation split at each epoch to stop the training process, the correct number of epochs to prevent overfitting was inferred from experience.

The final configuration of the model is a neural network with a single, ReLu-activated, hidden layer of 20 nodes that is trained for a total of 100 epochs using 10 batches, a learning rate of 0.01 and computes loss using cross-entropy.

The training history of loss and training accuracy is depicted in Figure 10. It can be seen that the loss is steadily decreasing and maximising the training accuracy.

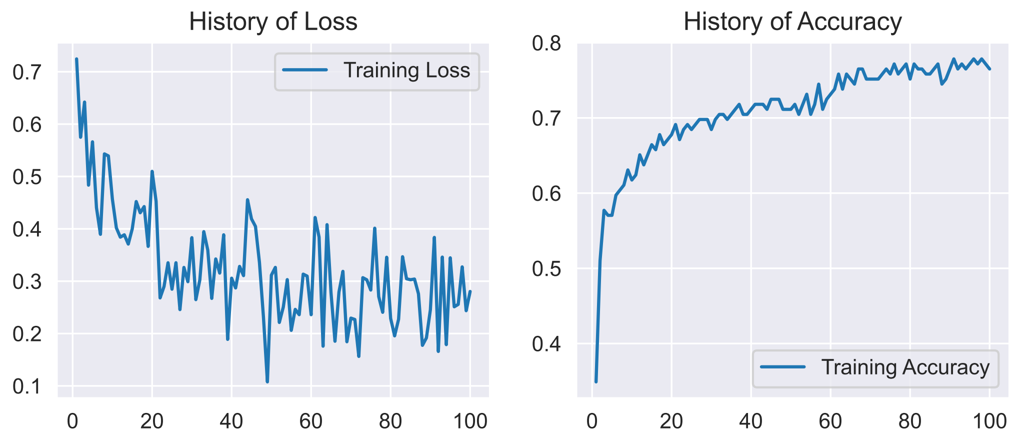

Figure 10: Training History of Custom Neural Network Implementation

5.2.3 Custom Neural Network Performance Evaluation

The fitted model achieves a training accuracy of 72% and a test accuracy of 74% (4). The model is best at predicting class 1, 2 and 7 correct, which is logical since these are the majority classes. Except for the minority class 3, that is very similar to class 1 and 2 (as seen from the EDA), the model does an overall good job of classifying.

Evaluation Metric	Score
Training Accuracy	72%
Test Accuracy	74%

(a) Custom Neural Network Accuracy Scores on Training and Test Split

Class	Precision	Recall	F-Score	Support
1	0.70	0.90	0.79	21
2	0.73	0.70	0.71	23
3	0.00	0.00	0.00	5
5	0.60	0.75	0.67	4
6	0.67	0.67	0.67	3
7	1.00	0.89	0.94	9
accuracy			0.74	65
macro avg	0.62	0.65	0.63	65
weighted avg	0.69	0.74	0.71	65

(b) Custom Neural Network Classification Report on Test Split

Table 4: Custom Neural Network Performance Evaluation

Keras Implementation. Similarly, the neural network architecture for the Keras neural network was mostly derived through trial-and-error. A final configuration of a simple, neural network with a single hidden, ReLu-activated layer of 50 neurons, with an `Adam()` optimiser set to a learning rate of 0.005 and optimising for cross-entropy loss on the entire batch (batch gradient descent) for each epoch was found to give good results. Using the `EarlyStopping()` callback, the training process was monitored, such that a continuous increase of the validation loss over 2 epochs would stop the fitting of the model. This prevented overfitting.

With this configuration, the model callback was reached after ~ 80 epochs. The training history is depicted in Figure 11 and reveals that the loss continuously decreased for the training split, and was stopped just as the validation loss was beginning to increase again.

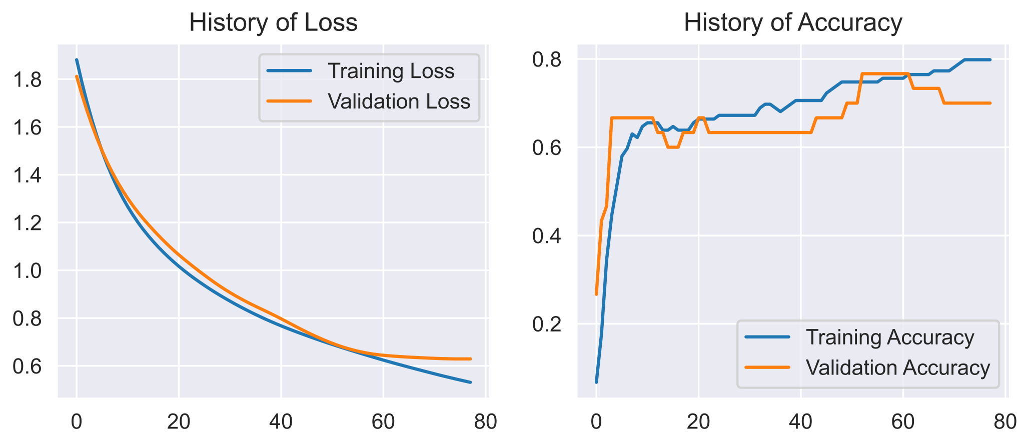

Figure 11: Training History of Keras Neural Network Implementation

5.2.4 Keras Neural Network Performance Evaluation

As can be seen from the history of accuracies, the final trained neural network achieves a training and validation accuracy of roughly 80%. The validation score for this model is a good estimate for the test accuracy, which is 75% (Figure 5). From the classification report (Figure 5) it becomes clear, that the model does well on the better separated classes 6 and 7. It has more difficulty separating the closely related classes 1, 2 and 3 and, in fact, never predicts the minority class 3. Nevertheless, the overall classification result is good.

Evaluation Metric	Score
Training Accuracy	78%
Test Accuracy	75%

(a) Keras Neural Network Accuracy Scores on Training and Test Split

Class	Precision	Recall	F-Score	Support
1	0.72	0.86	0.78	21
2	0.77	0.74	0.76	23
3	0.00	0.00	0.00	5
5	0.75	0.75	0.75	4
6	0.75	1.00	0.86	3
7	1.00	0.89	0.94	9
accuracy			0.75	65
macro avg	0.67	0.71	0.68	65
weighted avg	0.73	0.75	0.74	65

(b) Keras Neural Network Classification Report on Test Split

Table 5: Keras Neural Network Performance Evaluation

5.3 Random Forest

5.3.1 Introduction

The statistical observation of the wisdom-of-the-crowd motivated the emergence of so-called ensemble methods in the cosmos of machine learning. Ensemble methods are groups of predictors, that collectively construct the ensemble's decision [4].

One of the most popular ensemble-methods is the so-called `RandomForestClassifier()`. It refers to a series of individual decision trees independently trained on random sub-samples and randomly selected sets of features of the input data to generate non-correlated classifiers. The final-decision is then attributed to the class being predicted by the largest number of individual trees (hard voting-classifier) [4].

5.3.2 Random Forest Model Selection

The model-selection-pipeline for the `RandomForestClassifier()` was similar to that of the single decision tree. Again, a pipeline that involved scaling the features was used, and grid searched for a set of hyper parameters in order to find the configuration of hyper parameters that best generalise. Within training, the hyper parameters `n_estimators`, `criterion`, `bootstrap` (sampling from original data with or without replacement) and `max_depth` were searched.

5.3.3 Random Forest Performance Evaluation

The grid search returned that the best-generalising random forest classifier uses 100 predictors, Gini-impurity as a loss-function, bootstrapped random sub-samples of the training data and trains each tree to a maximal depth of 8. This fitted model has a training accuracy of 100%, while still having a high mean validation accuracy of 77% encountered during the 5-fold cross validation. The accuracy on the final test set is 83%.

Evaluation Metric	Score
Training Accuracy	100%
Validation Accuracy	77%
Test Accuracy	83%

(a) Random Forest Accuracy Scores on Training, Validation and Test Split

Class	Precision	Recall	F-Score	Support
1	0.95	0.95	0.95	21
2	0.83	0.83	0.83	23
3	0.00	0.00	0.00	5
5	0.50	1.00	0.67	4
6	0.75	1.00	0.86	3
7	1.00	0.89	0.94	9
accuracy			0.83	65
macro avg	0.67	0.78	0.71	65
weighted avg	0.80	0.83	0.81	65

(b) Random Forest Classification Report on Test Split

Table 6: Random Forest Performance Evaluation

6 Interpretation and Discussion of the Results

6.1 Classification Challenges and the Ideal Classifier

Building a good model to accurately predict glass fragments from a data set of 214 data points turned out to be demanding. The core challenges were:

1. Little amount of data for 6-class classification problem
2. Skewed class distribution
3. Class overlap

In the light of the model being used in criminal investigation processes, an ideal model would be transparent in a way that the decision-making is comprehensible for humans. Furthermore, the model should be well-suited to tackle the challenges mentioned.

6.2 Comparison of Model Performances

The three models evaluated within this project can be ranked by their performance (measured by the expected out-of-sample accuracy) as follows:

1. `RandomForestClassifier()`
2. `NeuralNetworkClassifier()`
3. `DecisionTreeClassifier()`

Looking at the individual classification reports, all models were able to almost perfectly predict class 7 which goes hand in hand with the findings from the EDA and Figure 3. In contrast, the prediction of the overlapping classes 1, 2 and 3 turned out to be generally challenging. In order to classify these classes correctly, the model needed to establish a complex decision boundary, that at the same time generalises well. The `RandomForestClassifier()` did the best job at separating the classes 1 and 2 from each other (Table 6), giving one indication of its overall good performance. The `DecisionTreeClassifier()` struggled the most at separating the overlapping classes (Table 3). Another noticeable, common pattern from the classification reports is that the minority classes were generally more difficult to predict. That is reasonable, since a lack of training examples makes it difficult to learn class-specific properties.

6.3 Interpretation of Model Performances

The `RandomForestClassifier()`, being an ensemble of a large number of, uncorrelated decision trees, as well as the `NeuralNetworkClassifier()` are complex models, with longer training times, but more accurate predictions. This explains, why both models outperform the simpler `DecisionTreeClassifier()`. The fact that the `RandomForestClassifier()` gives better results than the `NeuralNetworkClassifier()` is probably due to the fact that neural networks usually need larger amounts of training data to give good results. In fact, neural networks perform best on complex classification problems with large amount of data to learn from. Since this classification problem is quite the opposite of that, the neural network is not performing as well as the `RandomForestClassifier()`.

6.4 Conclusion

The analysis has shown that a model chosen solely on the basis of best performance (`RandomForestClassifier()`) has an expected out-of-sample performance of over 80%, which is a solid result given the challenges of the problem at hand. However, in the light of the model being used in assisting criminal investigations, transparency in the way decision are being made might be relevant, i.e. in court trials. Thus, if interpretability of the model is a core requirement for the final model, a `DecisionTreeClassifier()` should be considered.

References

[1] I. W. Evett and E. J. Spiehler. “Rule induction in forensic science”. In: (1987).

[2] Rasmus Berg Palm. *Nanograd github*. `https://github.com/rasmusbergpalm/nanograd`.

[3] *Universal approximation theorem*. Nov. 2021.

[4] James G. Witten D. Hastie T. Tibshirani R. *An Introduction to Statistical Learning - with Applications in R — Gareth James — Springer*. 2013.

[5] A.Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. 2019.