Machine Learning Project

Maria Sousa[1] and Sabrina Fonseca Pereira[2]

[1]mdom@itu.dk
[2]sabf@itu.dk

January 3, 2022

Course title: Machine Learning
Course code: BSMALEA1KU

1 Introduction

The purpose of this project is to implement and use different machine learning models to classify glass fragments into 6 categories based on 9 different forensic measurements. A neural network and a decision tree were implemented from scratch and compared to the corresponding scikit-learn [1] implementation. Other models provided by the scikit-learn library were also used to explore their efficacy in classifying glass fragments.

2 Exploratory data analysis

The dataset consists of 214 glass fragments, separated in training and test set, with 149 and 65 samples respectively. The glass fragments were labeled as belonging to one of the following classes:

Integer code	**Glass fragment 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 1: Classes of glass fragments in the dataset

The features consist of 9 measurements, including the glass fragments' chemical composition and its refraction index, a standard measurement made for forensic purposes. The chemical compounds were measured in terms of the weight percent for the elements Sodium (Na), Magnesium (Mg), Aluminum (Al), Silicon (Si), Potassium (K), Calcium (Ca), Barium (Ba) and Iron (Fe).

A first look into the distribution of classes in the training dataset (figure 2) shows that this is an unbalanced dataset, which may cause issues for the models when trying to predict underrepresented classes.

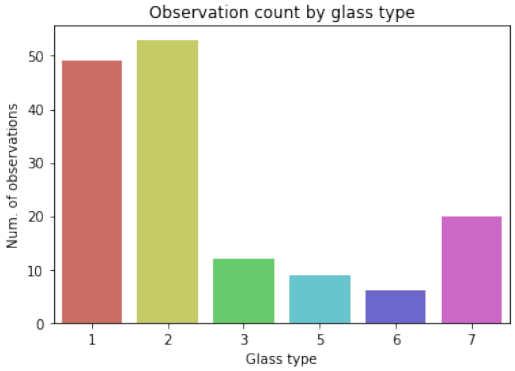


Figure 1: Observation count by glass type

By looking at the distribution of the samples across all features in figure 2, it is also anticipated the models might perform poorly when differentiating between classes 1 and 2 since there is not much variation between the samples. We can also see that class number 7 is easily distinguished by their Barium content for instance, which will be beneficial when trying to predict this class.

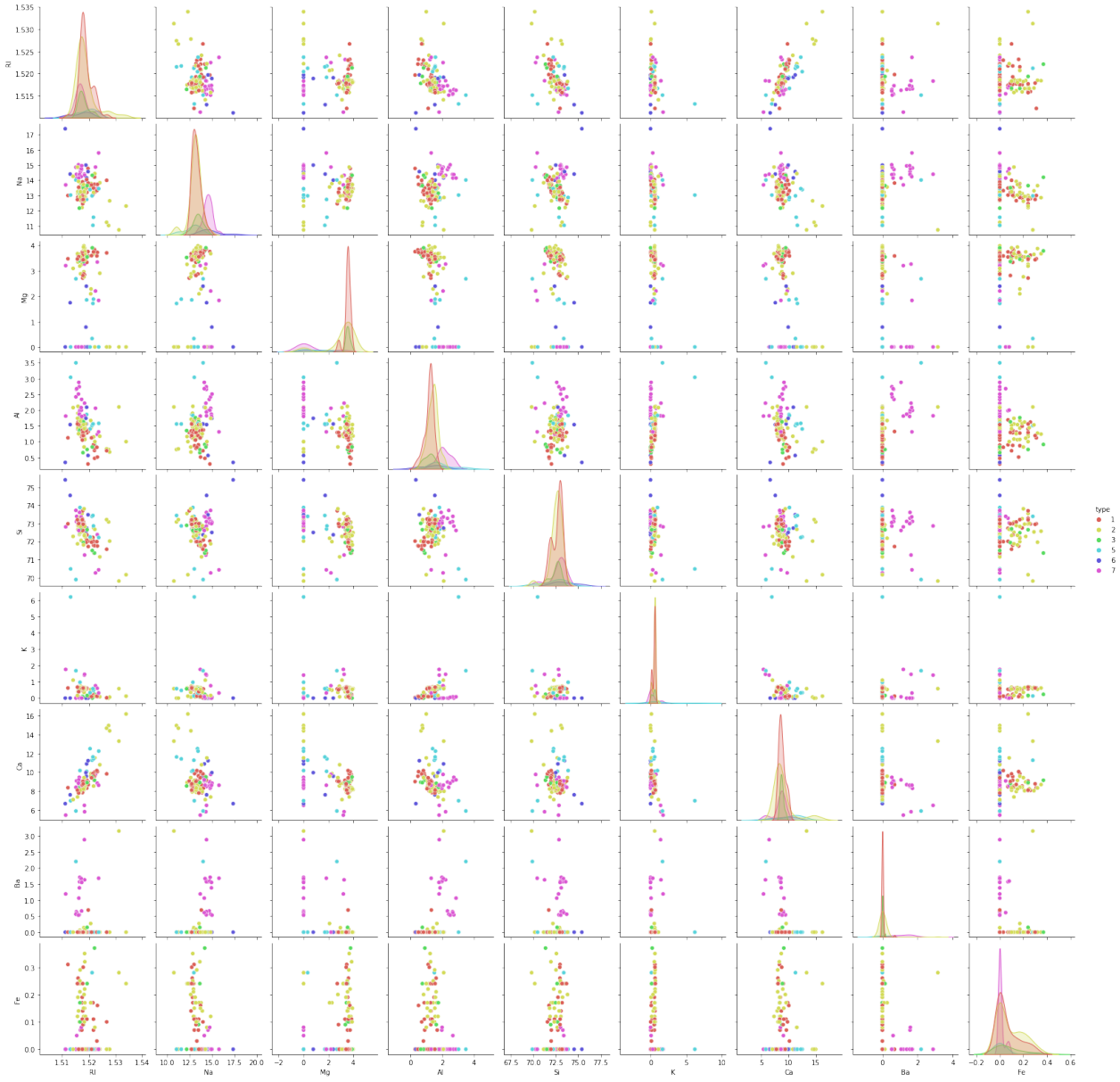


Figure 2: Distributions of samples across all classes

The summary statistics in of the training dataset (table 2) shows differences in scales and standard deviation of samples, so standardising the data before training will be beneficial, especially for the neural network implementation.

3 Decision Tree Model

Decision trees are one of the most powerful classification algorithm commonly used in machine learning [2]. They are easy to interpret providing simple classification rules that can be manually used to

confirm the results, and versatile handling discrete, continuous or mixed input types.

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe
mean	1.52	13.42	2.72	1.43	72.62	0.49	8.92	0.2	0.06
std	0.0	0.86	1.42	0.51	0.78	0.57	1.51	0.55	0.1
min	1.51	10.73	0.0	0.29	69.81	0.0	5.43	0.0	0.0
25%	1.52	12.93	2.28	1.17	72.28	0.13	8.22	0.0	0.0
50%	1.52	13.3	3.49	1.36	72.78	0.55	8.59	0.0	0.0
75%	1.52	13.83	3.61	1.62	73.05	0.61	9.14	0.0	0.11
max	1.53	17.38	3.98	3.5	75.41	6.21	16.19	3.15	0.37

Table 2: Description of training data

The structure of a decision tree starts on a root node which represents the top decision node where the first step of the algorithm selects the best predictor feature. From the root node derives the following decision nodes until reaching the leaf nodes from where there are no other nodes deriving.

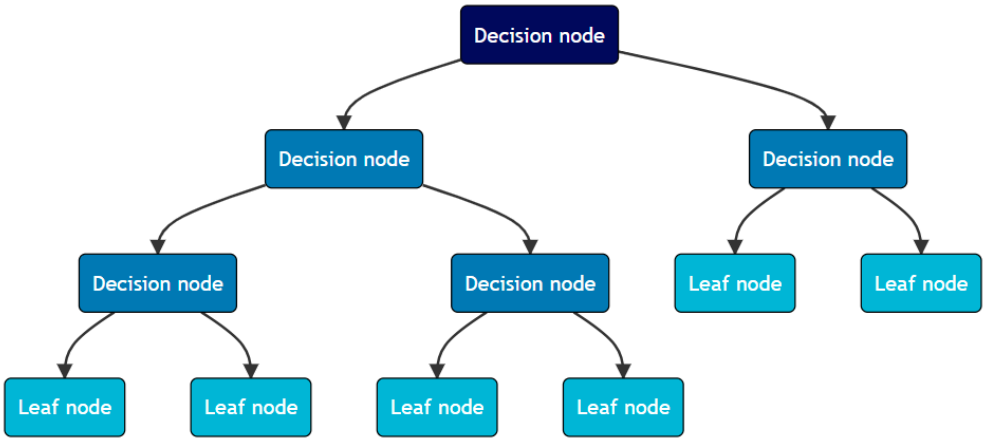


Figure 3: Decision Tree Structure

3.1 Decision Tree Implementation

To implement the decision tree, it was created a collection of functions which can be found in the file `functions.py`:

- `gini` (line 77) calculates the gini impurity of a given node
- `split_node` (line 90) splits a given node based on the given threshold value.
- `find_best_split_feature` (line 106) finds the best split for a specific feature. The best split is considered the one with the lowest gini impurity.
- `find_best_split` (line 141) calls on the `find_best_split_feature` and compares all the best split of each feature to decide what feature (and its best split value) is the best split. Again, the

best split is considered to to be the one with the lowest gini impurity.

- `build_tree` (line 164) a recursive function that builds a dictionary of nodes and its attributes. The best feature to split it on and its threshold, the gini impurity of the node, the values which indicate the count of how many observations are in each class, and the class which indicates what an observation would be classified as if it were a leaf node. The class is decided by counting the values, the value with the highest number of observations will become that node's class.
- `predict` (line 195) a recursive function that uses a tree dictionary to make the prediction for a given observation.
- `print_tree` (line 245) prints a human-friendly visualisation of the dictionary created by the `build_tree` function.

All those functions are brought together by two others, that should be used to train and predict:

- `decision_tree_train` (line 207) returns the decision tree as a dictionary.
- `decision_tree_predict` (line 226) returns a vector of predictions.

3.1.1 Training and optimisation

To train the model it was called the `decision_tree_train` function. The input of this function is the training data, the true labels for the training data and the max depth of the decision tree. This function will first create a list of all observations in the root node which is equivalent to all samples in the training data. Then it will pass on all its inputs and the list to the `build_tree` function.

The `build_tree` function will first create a dictionary mapping each feature to its index, which will then be used by `split_node` once `find_best_split` has found the best split feature and threshold for the current node.

To find the feature and threshold with the best gini score, the `find_best_split` function, takes the training data, the true label data and the list of observations in the current node. It then calls the `find_best_split_feature` for a set number of times, which will be equal to the number of features of the dataset.

To avoid repeated computations due to repeated values in the dataset, the `find_best_split_feature` function first finds the unique observation values for that feature and creates a list. It will iterate through that list, and call `split_node` on each iteration to generated a temporary split to calculate it's weighted gini. To calculate it, it is first needed the gini impurity of each child node of the split which is calculated by `gini` as:

$$G = \sum_{k=1}^{K} p_k(1 - p_k) \tag{1}$$

where k are the classes and p_k is the probability of getting a data point with label k. To find the gini score of the node it was calculated the weighted gini:

$$\mathrm{W} = w_1 g_1 + w_2 g_2 \tag{2}$$

where $w = \dfrac{\text{num. of observations in child node}}{\text{num. of observations in parent node}}$

`find_best_split` keeps track of the best weighted gini and its corresponding feature and split value, updating them only if the new gini score is better than the current best.

With the final decision of the best split, `build_tree` can call `split_node` to create two lists, one with all observations with values smaller or equal to the threshold of the best feature (left node), and another with the ones that are bigger (right node). Now `build_tree` will recursively call itself until reaching one of the two conditions: the gini of the current node is bigger than 0 or it is reached the max depth specified.

`build_tree` will be called twice within itself, one for the observations in the left node and another for the observations in the right:

```
1    # part of build_tree, line 187 functions.py file
2    if curr_step < max_depth and gini_node > 0:
3        node_dict['left'] = build_tree(max_depth, feature_data,
            label_data, left, curr_step+1)
4        node_dict['right'] = build_tree(max_depth, feature_data,
            label_data, right, curr_step+1)
```

This will create a nested dictionary which will be returned by the function so it can be used for making predictions.

3.1.2 Prediction

The function `decision_tree_predict` creates an empty list of predictions and count how many samples it needs to predict, which is the number of times it will run a for loop calling the `predict` function. `predict` makes the prediction for one observation at a time by recursively calling itself as it iterates through the tree dictionary generated by `decision_tree_train`. Once there is a prediction for each sample, the function returns the predictions as a list. The figure 4 illustrates the resulting tree.

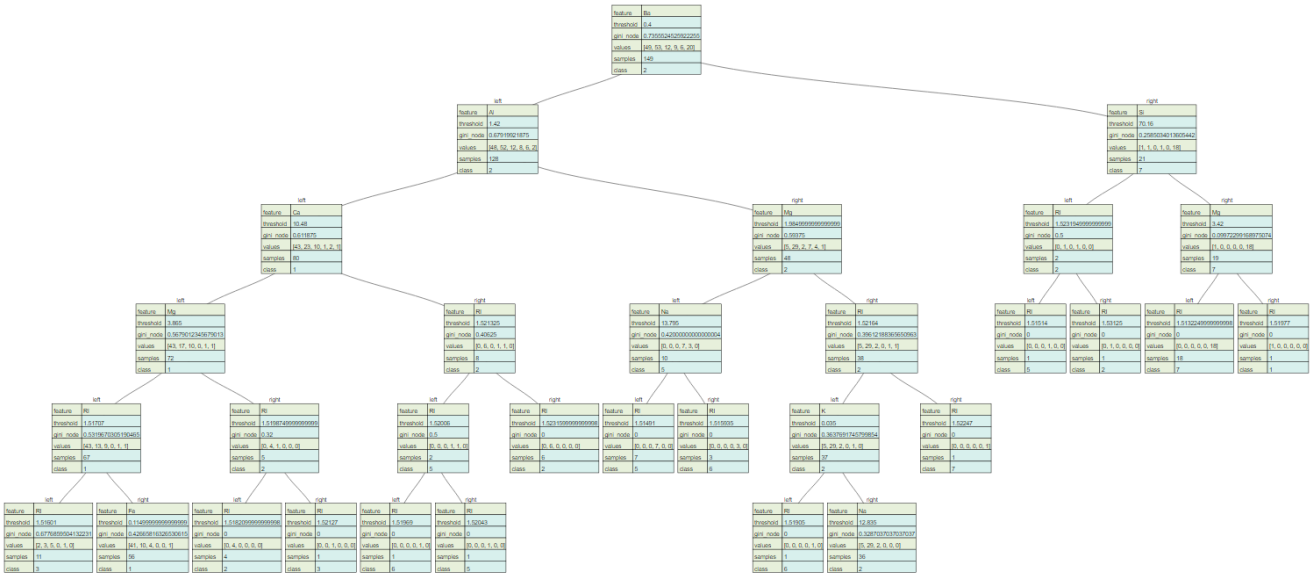


Figure 4: Representation of the tree generated at max depth 5

3.1.3 Results

To test the model, it was used the provided test dataset and measured its accuracy and f1 score. To check for correctness, the model was also compared to the equivalent implementation of the scikit-learn library, where the results were very similar despite of running time difference (table 3). The similar results confirm the considerable performance of the decision tree in the data classification.

Decision Tree	Accuracy	F1 score	Run time
From scratch	0.78	0.79	6.8s
sklearn	0.80	0.80	0.9s

Table 3: Comparing Decision Tree implementations

4 Neural Network Model

The theory behind of neural network algorithm is based on the concept of the biological neurons, which are brain cells organized in layers that receive and process information and transmit it to the connected neurons [3]. In the algorithm, the neurons are represented by nodes and the structure starts with the input layer that can take several inputs, followed by a variable number of hidden layers composed by nodes and ends with the output layer which produces a single or multiple outputs.

In the implemented model, the input layer consists of 9 nodes (one for feature of a glass fragment), a single hidden layer of 9 nodes using the sigmoid activation function, and the output layer with 6 nodes, one for each class in the dataset (figure 5).

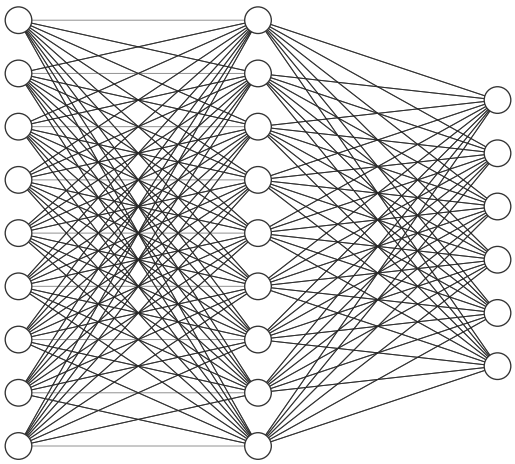


Figure 5: Structure of the implemented neural network

Neural network algorithm has the ability to learn and model non-linear and complex relationships between input and output which is very common on real life problems, and do not impose restriction in the input variables, i.e. how they should be distributed. These proprieties make it a very useful model.

4.1 Mathematics behind the algorithm

To compute the final vector of probabilities, several mathematical formulas were used.

In the dataset, the feature values are in different scales so to ensure that all features contribute equally when training the model, the data was normalized using Min-Max normalisation:

$$\frac{x - x_{min}}{x_{max} - x_{min}} \tag{3}$$

where x is an original value.

The first step of the neural network algorithms is to multiply the feature values by its corresponding weights, then add it to the bias constant (formula bellow). The weight value reflects the importance of each input while bias inserts flexibility and better generalisation to the neural network [4].

$$\sum x_i w_i + b \tag{4}$$

where x_i is the feature value, w_i is the weight and b is the bias.

The weighted sum of inputs is the input of the activation function in the hidden layer. The activation function purpose is to introduce non-linearity to the model which is necessary as a simple linear function (1) has very limited capability when trying to classify complex data [5]. This implementation

uses the sigmoid function which is defined as:

$$\frac{1}{1+e^{-x}} \tag{5}$$

where x is the input value.

In the output layer, the output of the sigmoid function is multiplied by a different set of weights summed to a different sets of biases. As this is a multi-class classification problem, the result is then insert through the Softmax activation function in other to get the probability vector for each data point. Each entry in the vector will correspond to the probability of it belonging to one of the classes. The Softmax function is defined as:

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{6}$$

where z_i/z_j is each element of the input vector

During back propagation, for each set of predicted values the cross entropy loss was calculated to determine the difference between the prediction and the true values. The cross entropy loss function defined as:

$$-\sum y_i \log \hat{y}_i \tag{7}$$

where y_i is the actual probability and $\hat{y}_i$ is the predicted probability.

4.2 Neural Network Implementation

The feed-forward neural network was implemented from scratch with one hidden layer using the sigmoid function as its activation function. As this is a multi-class classification problem, one-hot encoding was used in conjunction with the softmax function in the output layer.

4.2.1 Training and optimisation

To train the model, the function `neural_network_train` was implemented. It takes as inputs: a matrix of observations, a vector of true labels, and the desired learning rate and epochs as hyperparameters. This function outputs the optimised weights and biases which will be used for predictions, and a list of the cross-entropy loss for each iteration, which is used to analyse the backpropagation process.

The weights were initialised as random values and the biases as the zero vector. The weighted outputs were then put through the sigmoid function in the hidden layer. The output of this computation is then sent through to the output layer. The values are again multiplied by weights and biases and then put through the Softmax function, which in turn outputs a vector of probabilities for each one of the observations.

To optimise the model, the cross entropy loss is calculated for each iteration. The results are added to a list which is returned by the `neural_network_train` function so the optimisation process can be visualised.

The function uses backpropagation to minimise the loss by optimising the weights and biases. The directional derivatives for each layer are computed for each epoch. This process starts with the last layer, as its input depends on the results from the hidden layer that comes before it. The resulting derivatives are then multiplied by the learning rate to slowly shift the weights and biases to optimise the model and increase the accuracy of the model.

The partial derivative of the loss in respect to the weights and bias vary depending on the activation function used in the different layers since the chain rule is applied to solve it. For the output layer (Softmax), the partial derivative in respect to the weights is given by:

$$\frac{\partial\ L}{\partial\ wo} = \frac{\partial\ L}{\partial\ o_{soft}} \cdot \frac{\partial\ o_{soft}}{\partial\ i_{soft}} \cdot \frac{\partial\ i_{soft}}{\partial\ wo}$$

$$\frac{\partial\ L}{\partial\ o_{soft}} \cdot \frac{\partial\ o_{soft}}{\partial\ i_{soft}} = o_{soft} - y$$

$$\frac{\partial\ i_{soft}}{\partial\ wo} = i_{soft}$$

where L = loss, w = weight of the output layer, o_{soft} = softmax output, i_{soft} = softmax input, y = true labels (hot encoded).

The partial derivative in respect to the bias is given by the same formula differing just in the last part:

$$\frac{\partial\ i_{soft}}{\partial\ bo} = 1$$

where bo = bias of the output layer.

For the hidden layer (Sigmoid), the partial derivatives in respect to weights and bias follows the same principle were the last fraction is equal to 1 for the bias. The formulas are presented bellow:

$$\frac{\partial\, L}{\partial\, wh} = \frac{\partial\, L}{\partial\, o_{sigm}} \cdot \frac{\partial\, o_{sigm}}{\partial\, i_{sigm}} \cdot \frac{\partial\, i_{sigm}}{\partial\, wh}$$

$$\frac{\partial\, L}{\partial\, o_{sigm}} = o_{soft} - y * wo$$

$$\frac{\partial\, o_{sigm}}{\partial\, i_{sigm}} = o_{sigm} \cdot (1 - o_{sigm})$$

$$\frac{\partial\, i_{sigm}}{\partial\, wh} = x$$

where L = loss, wh = hidden layer weights, o_{sigm} = sigmoid output, i_{sigm} = sigmoid input,y = true labels (hot encoded), wo = output layer weights, x = input feature values.

4.2.2 Prediction

The prediction function `neural_net_predict` takes the weights and biases found by `neural_network_train` and uses them to predict the classes for all observations in a given dataset. As the output of the softmax function is a vector of probabilities and not a single value, the index for the highest value is used to identify the class predicted. As the index values do not match the class values in the dataset, a dictionary was used (line 26) to map the indexes to the correct class integer code.

4.3 Results

We tested our models' prediction using raw dataset, and preprocessing it in two different ways, min max normalisation and PCA. The

The results are presented in the table below:

Neural network input	**Accuracy**	**F1 score**
Not normalised	0.35	0.18
Normalised	0.66	0.63
PCA	0.74	0.74

Table 4: Comparing neural network performance with different types of input

As expected, using non-normalised data fared poorly, in fact it only predicted one class. The normalised data had significantly better results, but still could not predict class 3. Which makes sense, as it is one of the most under-represented class, and when looking at figure 2 it is visible that it has similar chemical composition to classes 1 and 2, as they are all windows, and that would make it even more difficult to separate them.

To improve model's performance, the data was also transformed using scikit-learn's PCA method. When using the transformed data to predict and test the model a 8% increase in accuracy and a 11% increase in the F1 score was achieved.

The similarity of results when comparing the accuracy score of normalized data with the correspondent scikit-learn implementation was used to assert the models correctness.

5 Random Forest

The random forest model works by training several decision trees made up of random subset of samples from the original training data, it will obtain the predictions from those individual trees, then predict the class that got most votes.

As the decision tree performed well, and often we can get better predictions with ensemble methods than with the best individual predictor [3] we decided to test the random forest algorithm from the scikit-learn library as our third model to see if we could improve our prediction accuracy on the glass dataset.

The model was trained with 250 estimators and a max depth of 8. The hyperparamaters values were found by testing different values using `GridSearchCV`. Predicting glass fragment classes with 86% accuracy and 86% weighted f1 score, it was the best model we tested.

6 Interpretation and discussion of the results

Analysing the scores of all models (table 5), random forest model is the one with the highest scores which was expected, as it is a combination of several decision trees and the decision tree model, which has only one decision tree, had already performed better than the Neural Network.

	Neural Network (PCA)	**Decision Tree**	**Random Forest**
Accuracy	0.74	0.78	0.84
F1 score	0.74	0.79	0.84

Table 5: Comparison between different models

The shortcomings of the neural network might be explained by its complexity, while it makes for a very flexible model with many hyperparameters, it also makes it difficult to find ideal ones. The neural network implemented in this project has only two hyperparameters, the learning rate and number of epochs for optimisation. Using a different activation function and the addition of more layers could have been beneficial to its predictive capability. Another issue leading to lower scores could be the nature of the problem itself, as there are 6 different classes and the results are given as a probability vector the different classes can have similar probability values which leads uncertainty and misclassification.

On the other hand, tree based models are simpler algorithms that use the input values directly, which in this case, yields better results. This could be explained by the fact that the chemical composition of different types of glass is inherent to it, makes them what they are. So classifying them based on these features and the values that represent them is a better approach then using neural networks to try and find patterns in the data.

When comparing the results of the 3 different models by looking at their confusion matrix in figure 6, the most glaring difference is how the neural network had more difficulties differentiating between classes 1 and 2 than the tree based models. Underrepresented samples were generally difficult to classify, not only due to the imbalance in the data but mostly because of similar chemical compositions which makes the data harder to separate.

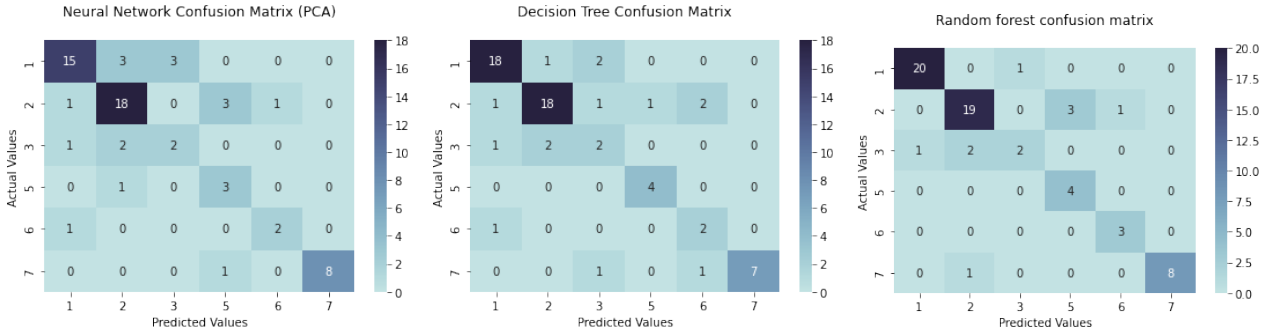
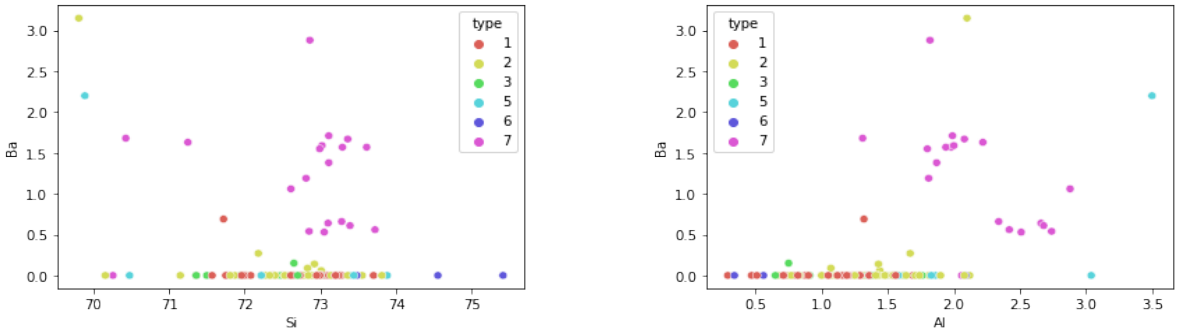


Figure 6: Confusion matrices for all models

The exception is class 7, as seen in figure 1, samples that belong to it still make up a smaller percentage of the training set in comparison to classes 1 and 2. But when looking at a feature like Barium (Ba), the class 7 samples are much easier to distinguish than any other.



(a) Barium and Silicon

(b) Barium and Aluminum

Figure 7: Relationships between chemical compounds

When looking at the first split of the decision tree in figure 8, it can also be seen that Barium is the feature chosen for the first split. Followed by Silicon and Aluminum as the best features for the second split.

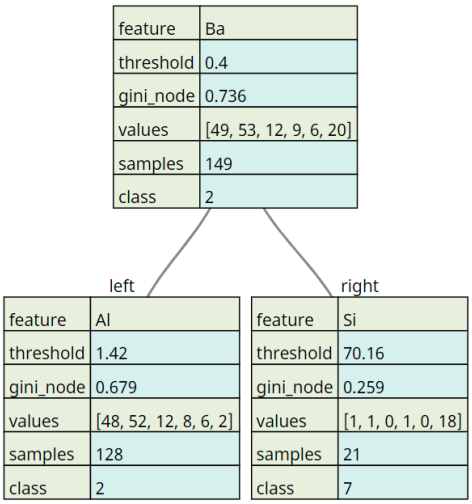


Figure 8: First split of decision tree

7 Conclusion and future work

Machine learning models can be used to predict glass fragments for forensic purposes - with a fair bit of certainty. The most evident hurdle for all models was differentiating between classes 1 and 2, that despite being the most representative classes, they have similar very similar chemical composition.

The scikit-learn implementation of random forest was the best performing model, followed by the decision tree and lastly the neural network. Both the implementation from scratch and scikit-learn's implementations did not show great results, even with normalised data. But we were able to get much better predictions when projecting the feature values using PCA.

The neural network implementation could see improved performance by the addition of more hyperparamater options like the number of hidden layers, number of node in the hidden layers and different activation functions such as ReLU or Tanh. In case of the decision tree, implementing pruning techniques to reduce the risk of overfitting could also be beneficial.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[2] Bahzad Charbuty and Adnan Abdulazeez. Classification based on decision tree algorithm for machine learning. *Journal of Applied Science and Technology Trends*, 2(01):20–28, 2021.

[3] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Inc., Sep 2019.

[4] Farhad Malik. Neural Networks Bias And Weights - FinTechExplained - Medium. *Medium*, Dec 2021.

[5] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 04:310–316, 05 2020.