Exam Assignment: Forensic Identification of Glass Fragments

Malthe Pabst (mrla@itu.dk), Nicola Clark (niccl@itu.dk), Andy Bao Nguyen (anbn@itu.dk)

January 2, 2022

Machine Learning BSMALEA1KU (BSc DS) Fall 2021, IT University of Copenhagen

Contents

1 Exploratory Data Analysis

1.1 The Data

The data consisted of 214 observations, from six different glass fragment types. The glass fragment features consisted of measurements of the refractive index and the amounts of eight different elements contained in the observations. Overall nine different features for the glass fragments were given:

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

The six different glass types:

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

The data provided, was already divided into a train and test set. The train set contained 149 glass fragments while the test set contained 65 glass fragments.

1.2 The Classes

The classes were not equally distributed, within the train set class 1 *window from building (float processed)* and class 2 *Window from building (non-float processed)* made up the most of the data set, while class 5 *container* and class 6 *tableware* had only under 10 observations each. The third biggest class was class 7 *headlamp*, however this class was significantly less well represented than class 1 *window from building (float processed)* and class 2 *window from building (non-float processed)*, with less than half of the number of observations.

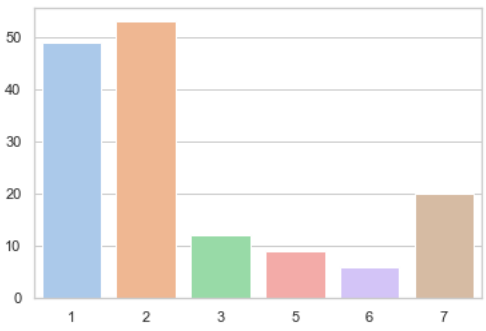

Figure 1: Distribution of the Classes

When looking into the distributions of the values within each feature for each class, it was clear that class 6 *tableware* had only 0.0 values for the features K, Ba and Fe. And only 2% and 9.4% of class 1

window from building (float processed) and class 2 *window from building (non-float processed)* had non 0.0 values for the Ba feature.

Overall it was clear that class 6 *tableware* has fewer observations and less with, non 0.0 values, while the majority of observations were of the class 1 *window from building (float processed)* and class 2 *window from building (non-float processed)*, which had no 0.0 values for any feature. It is important to note that while many of the values were 0.0, there were no missing or NA values, so the 0.0 values still held valuable meaning.

1.3 The Features

A correlation matrix was made for each of the features; a Pearson correlation matrix was used. One of the things that catches the eye was the correlation between Calcium (Ca) and the refractive index (RI). With a Pearson score of 0.82, this was by far the strongest positive correlation of all features. Furthermore the matrix showed that there was a strong negative correlation between Silicon (Si) and the refractive index.

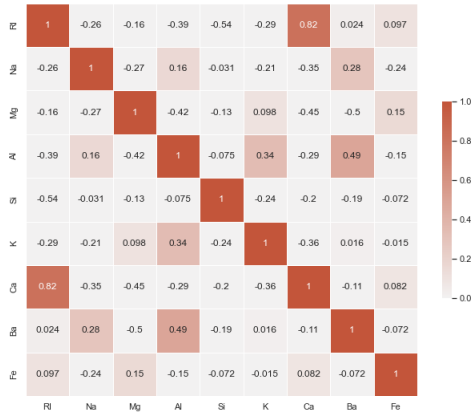

Figure 2: The correlation matrix for each feature

2 Data Preprocessing

Due to the class imbalance the data needed to be balanced before it was used to train models. This was done in order to improve the accuracy of models and to train models that were fairly good at predicting all classes, not just class 1 *window from building (float processed)* and class 2 *window from building (non-float processed)*. Synthetic Minority Oversampling Technique SMOTE was used to create synthetic observations of the underrepresented classes [1].

After oversampling the classes were balanced as can be seen in figure 3.

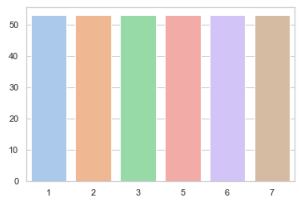

Figure 3: Distribution of Classes After SMOTE

In addition to oversampling, the data was transformed using Scikit-Learn's Standard Scaler. This was important because the scales of many of the features were very different. For example RI values ranged from 1.51 to 1.53, whereas Na values ranged from 11.45 to 15.15 within the test set. Standardising the features allowed for changes in all variables to be on the same scale and thus, viewed as equally important by the models. It is important to note that for simplicity, scaling was used for all models, despite the random forest and decision tree classifiers being insensitive to scale.

Lastly the data was shuffled, despite also shuffling in the classifiers themselves, for example in mini-batch gradient decent in the neural network. Nevertheless shuffling was applied in order to ensure that classes did not appear sequentially within the data, as this could have had an adverse effect on some models.

2.1 Data used for Cross Validation

It is important to note that the data that were used for cross validation for the self made decision tree classifier in this project were not preprocessed. Preprocessing caused overfitting in the cross validation and so the process was cut out.

Cross-validating with oversampled data can lead to non independent train and test sets within each split, as values derived from the test set can, and do, end up in the train set. In theory, the training data on each spit should be over sampled within each fold. This was implemented in all of the library models[2][3][4] using an IMBLearn pipeline[5].

3 Decision Tree

3.1 The Theory

Decision trees are supervised machine learning models, this means that they classify unseen data using information gained from training data. One of the main problems that decision trees are used to solve is that of classification. A decision tree, when used for classification, can be understood as a kind of tree graph with a root node at the start and leaf nodes at the end which each represent a classification. Data flows from the root, through decision nodes which determine the path that is taken through the tree. This path ends in a leaf node, from which the classification can be read.

While training the decision tree, decisions have to be made on where to best split the data. To calculate the most efficient splits on the decision nodes, a few different tools can be used. The first is Gini impurity, which gives a score in the range of 0 to 1 - where 0 represents a completely pure split, and values closer to 1 indicate impurity. The mathematical formula used to calculate this is:

$$g(s) = 1 - \Sigma_{k=1}^{k} p(k)^2$$

Impurity is one minus the sum of probability of class k squared. This is one of the most commonly used methods to split data in a decision tree. The Gini impurity is a measure of how impure split is, where a completely pure split contains data of only one class on each side of the split. To find the best possible split, it's necessary to calculate the Gini impurity for each side, followed by a weighted normalization. The mathematical formula for a split where data is split into two sets a and b is:

$$G(S) = g(a)\frac{|a|}{|S|} + g(b)\frac{|b|}{|S|}$$

Another function that is commonly used to make a split is Entropy. While Gini calculates the pureness of the node given a potential split, Entropy measures the disorder. Entropy gives a value between 0 and 1, here 0 indicates that the node is in order, while 1 indicates that the node is in disorder. The formula for entropy is:

$$h(S) = -\Sigma_{k=1}^{k} p(k) log(p(k))$$

Just like Gini the Entropy has to be measured for each side of the split, and the weighted average is taken.

A decision tree can take hyper parameters which can alter the growth of the tree in such a way as to limit overfitting. One commonly used hyperparameter is the maximum depth of the tree, which determines the maximum decision nodes the data can pass through before ending up in a leaf node. Another common hyperparameter is the minimum samples for a node to make a split during training, thus turning it into a decision node. This ensures that the leaves contain a minimum number of observations; this prevents the tree from overfitting.

One of the benefits of decision trees is the fact that they are a "White-box" method. This means that it is possible to understand the reasoning behind every decision made. One of the disadvantages of decision trees is that they can be easily overfitted to training data, without well implemented regularization.

3.2 The Code

Firstly, the tree structure itself was created. This was done by making two classes, a node and a tree. The node class could create an object, a node, which could contain the following information: node-id, the parent of the node, the right and left child of the node, the depth of the node in the tree, a Gini impurity value, the feature and threshold value for the split, the training data that had been passed to it from its parents, and the most common class in the data.

Following the creation of the node class, the tree class was made. Each tree object contained the following information: a choice of Entropy or Gini impurity, the maximum depth of the tree, the minimum samples required to split, and the tree itself stored in a dictionary for easy inspection.

Following this, a split function was made that calculated the Gini impurity value. This was done by calculating the Gini impurity value of each possible split for each feature, and returning the feature and threshold of the split with the lowest Gini impurity value. Possible splits for a feature were calculated by finding the thresholds between each pair of closest values, when the data was sorted by the feature.

The split function could then be used recursively inside the create tree function to train a tree. This was done by passing data from the root to all of the leaves, while creating the most optimal split at each decision node. After splitting optimally at a decision node, the tree passed the data that was less than the threshold value of the split to the left child of the node, and the data that was greater than the threshold value to the right child. Then the split function was called for each child with its given data until no further splits were possible.

Naturally, this method created a tree that was highly overfitted to the training data it was given. Regularisation in the form of maximum depth and minimum samples per split were added as hyperparameters for the tree class.

Before each split the hyperparameter values were compared to the data in the node to decide whether a split was possible, creating a leaf node in place of a decision node, if this was not the case.

3.3 Choosing Hyperparameters

In order to choose the best hyper parameters for the decision tree, a grid search with k-fold cross validation[6] was used. This was implemented by splitting the training data into different train and validation sets known as folds. The distribution of the classes were taken into account while making the folds. The grid search set the hyperparameters given to it and then ran though the different data sets, returning the mean test score for all k-folds. The scores were then plotted using a heatmap, so that they could be better visualised. From this the best hyperparameters were found to be: a maximum depth of 3, minimum sample split of 2, as can be seen from figure 4.

Figure 4: Heatmap over grid search with k-fold cross validation, x-axis is minimum sample, y-axis is max depth

3.4 Results for the Decision Tree

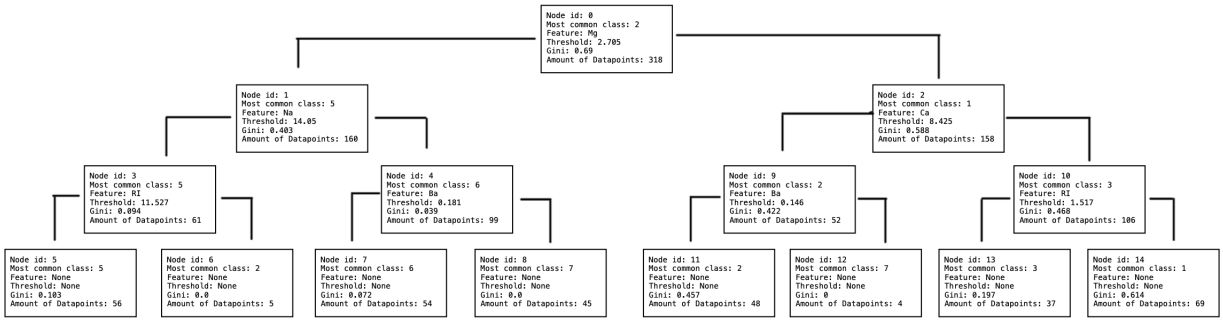

Figure 5: A visualization of the self implemented decision tree model

As figure 5 shows, the decision tree had a max depth of 3, and did not split if the node had 2 or fewer observations.

At the top we have the root, which was the first split in the tree. It split at Mg, with a threshold of 2.705, every observation with a Mg value less than the threshold was passed into the left child of the root (in this case, node 1), and the rest were passed to the right child (node 2). The rest of the tree followed the same pattern, observations smaller than the threshold were always passed left, and observations greater than the threshold were always passed right.

At the bottom of figure 5, the leaf nodes can be observed. These nodes do not split, and they therefore classify the observation to the class that is most represented in the leaf node. In the visualised tree all the all the

classes were represented by at least one leaf node. This was not guaranteed to be the case and most likely, at least in part, is a product of chance.

The self built decision tree classifier gave a macro average accuracy of 64.62%, recall of 61.8% and F1-Score of 58.22%. For comparison the same hyperparameters were given to the sklearn DecisionTreeClassifier [2]. This classifier gave a accuracy of 67.69%, a recall of 63.25% and a F1-Score of 60.34%.

Classifier	Accuracy	Recall	F1-Score
Our classifier	64.62%	61.8%	58.22%
Sklearn classifier	67.69%	63.25%	60.34%

In another library model, where the hyperparameters were correctly optimized, specifically for the sklearn classifier, an accuracy of 63.08%, recall of 64.8% and an F1-score on 59.42% were achieved.

From the confusion matrix in figure 6, it can be seen that the decision tree had problems classifying class 3 *window from vehicle* as none of the observations of class 3 in the test were classified correctly. As expected class 1 *window from building (float processed)* was well classified with an accuracy of 17 out of 21.

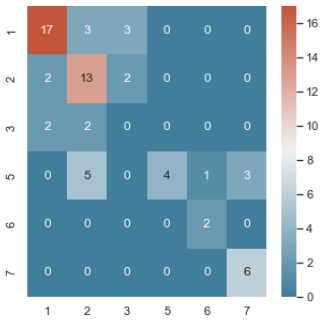

Figure 6: Confusion Matrix for Self Made Decision Tree Classifier, X-axis is real label

4 Neural Network

4.1 The Theory

A feed forward neural network is a supervised machine learning model. A neural network consists of different layers: an input layer, hidden layers, and an output layer. In each layer each neuron is connected to every neuron in the previous and next layer. The neurons take in a linear combination of the neuron values in the previous layer, apply an activation function, and pass the final value to the next layer. This calculation can be simplified such that each layer is represented by a matrix of the weights and biases that define the linear combination of the output vector/matrix X from the previous layer:

$$l(X) = X \cdot W + b$$

Here X is a matrix of data that is moving through the network during a forward pass. W is a matrix of weights. The biases are represented by a vector b but in practice they are appended as a row at the bottom of the weights matrix, a column of 1's is also added to the right side of X to enable multiplication. After this a non linear activation function is applied to each neurons value. This is done to introduce non-linearity, which adds complexity to the network. As, if each neuron were a linear function, the whole network would be a linear function which would not be able to capture non-linear patterns in data.

One such activation function is Sigmoid:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The sigmoid activation function outputs values between 0 and 1. A benefit of using a sigmoid activation function is that is easily differentiable. However it can lead to slow to train models due to weights not being updated much when values are close to 0. No learning occurs if the weight happens to reach 0, this is known as the vanishing gradient problem [7].

Another activation function is Rectified Linear Unit ReLU:

$$R(x) = max(0,x)$$

ReLU is often used for its simplicity and speed. Large weight updates can however cause a ReLU activated neuron to always output 0, which renders the neuron mostly useless for training. This is known as "dying ReLU" and is often tackled using similar functions instead, such as leaky ReLU [8].

In the output layer softmax is a commonly used activation function. Softmax calculates the probability of each class according to the model, and can be used to return the class with the highest probability for classification problems.

$$softmax(x) = \frac{e^{x_n}}{\Sigma_{i=1}^{n} e^{x_i}}$$

In a feedforward neural network, the data moves from the input layer, through the network, to the output layer during a forward pass. Backpropagation is then used in order to optimize the weights. In order to do so the partial derivatives of a loss function are calculated with respect to the weights and biases. These derivatives are then used it to update the weights and biases:

$$W = W - \alpha L(W)$$

Here α is the learning rate or the step size, and $L(W)$ is the gradient of loss function with respect to W. In practice the cycle of forward pass and backpropagation is repeated over many epochs, in order to optimize the weights and biases.

One loss function, commonly used in tandem with a softmax output layer is the cross-entropy loss function.

$$L(\hat{y}, y) = -\Sigma_{k=1}^{K} y \cdot log(\hat{y})$$

The cross-entropy loss function calculates the sum of the log predicted probabilities of the actual class. Where y is the one hot encoded vector and $\hat{y}$ is the probability distribution as predicted by the model, summed over all observations in the current sample K.

4.2 The Code

In practice the neural network which was coded did not work. The possible reasons for this will be discussed in the results.

Nevertheless an adiff, a network, and a layer class, were created for the neural network. The network class was made of multiple layers and the layers were made of multiple weights of the adiff class.

The adiff class was inspired by Rasmusbergpalm's GitHub [9]. It kept track of how the adiff was created in a list of parents which was stored in an instance attribute. From this information, partial derivatives of any adiff could be calculated recursively, by using the backward method on the adiff. Thus all mathematical operations had to be calculated using adiffs, in order enable future differentiation.

Each layer stored the adiff weights in a matrix represented by a list of lists due to numpy operations not being compatible with the adiff class. With the dimensions of this "matrix" depending on the number of neurons in the current and previous layer. The weights and biases were initialised using random values from a normal distribution $N(0, 0.25^2)$. The next layer input, which was calculated by a step between layers in the forward pass, was also stored, to enable easier calculation between layers. As an added benefit it provided storage for the final output in the last layer.

When creating an instance of the network class, the class took the parameter layers_info. This was a list of integers which assigned how many neurons were to be in each hidden layer. Other parameters the network class took during initialisation decided:

- The type of activation function in the hidden layers, ReLU or Sigmoid
- The learning rate, α
- The type of gradient decent GD, Stochastic SGD, Batch, or Mini_Batch
- The batch size if Mini_Batch was used
- The number of learning iterations, Epochs

One a network was initialised, it could be fitted to training data. The fit function created the network and iteratively looped through the following steps "Epochs" times:

1. Created a relevant sample of data depending on the GD type that was chosen. One randomly selected observation for SGD and "batch size" many for Mini_Batch.
2. Fed the sample through the network using a forward pass.
3. Calculated the loss using cross entropy loss.
4. Calculated the partial derivatives of the loss function with respect to all the weights, using the backward method of the adiff class.

5. Changed the weights accordingly $w = w - \alpha \cdot gradient$ for each weight and bias in each layer.

The forward pass was implemented by matrix multiplication between the input matrix and the weights-bias matrix in the next layer. This was followed by passing the resulting matrix through the activation function chosen in the initialisation. The output of this layer was then multiplied with the weights-bias matrix from the next layer and the process was continued until the output layer was calculated. The output layer was hard coded to consist of 6 neurons, corresponding to the 6 classes of glass. A softmax activation function was used to output probabilities on the last layer, as these were needed for the cross entropy loss function [10].

The predict method of the network class took data and classified it using a forward pass through a trained model. The vector output of probabilities was then argmaxed for each observation to find the class label corresponding to the highest probability value.

4.3 Choosing Hyperparameters

Optimal hyperparameters could not be chosen due to a non functional Neural network.

In practice hyperparameter selection would have been implemented in much the same way as it was for the library implementation of the neural network [4] where Grid search cross validation [6] was used with a parameter grid of hyperparameters to search over. The best hyperparameters, as found for the library implementation were: activation tanh, learning rate of 1e-05, and hidden layers of the sizes [30, 35, 30, 30].

4.4 Results from the Library Model

The final, library implemented, model gave an accuracy of 69.23%, recall of 77.71% and a F1-Score of 70.76%.

From figure 7 it can be seen that class 3 *window from vehicle*, class 5 *container*, class 6 *tableware*, and class 7 *headlamp* had high accuracy, however recall on class 3 and 5 was low. This suggests that the model was overly sensitive to predicting class 3 and 5. Class 1 *window from building (float processed)* was miss-classified as class 3 *window from vehicle* 38% of the time.

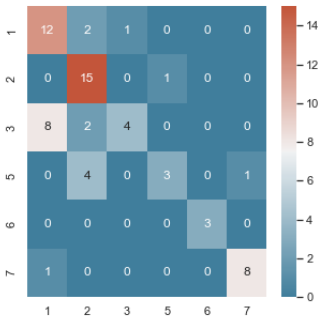

Figure 7: A confusion matrix of the results of the library implementation of the neural network

5 Random Forest

5.1 The Theory

Random Forest is an ensemble machine learning model - that is because the forest consists of many different decision tree models. The trees are generated using different samples of observations and features. To randomize the observations and features bootstrap aggregation can be used. Bootstrap aggregation

or bagging takes in the data points and features. It randomly bootstraps some observations from the data set and some features for each tree. It is important to note that this method uses replacement so that the observations and features can be used for another tree or multiple times in one tree.

A random forest takes the same hyperparameters as a decision tree: minimum sample split, max depth, and split method. But in addition to this the random forest classifier also takes in the number of trees that are desired within the forest. When the random forest is given an unknown data point, each tree in the forest classifies the point. Then an overall prediction can be made, based on a majority vote.

5.2 The Code

To make the random forest, the scikit-learn RandomForestClassifier [3] was used. The RandomForestClassifier takes in different hyperparameters, the ones used were: n_estimators which is the number of trees in the forest, criterion which is the splitting method, max depth, minimum sample per split and bootstrap.

```
from sklearn.ensemble import RandomForestClassifier
RF = RandomForestClassifier(n_estimators=100,
                            criterion = 'gini',
                            max_depth = 5,
                            min_samples_split = 2,
                            bootstrap = True)
```

Figure 8: The code that was used in order to create the RF

5.3 Choosing Hyperparameters

To choose the best hyperparameters the scikit-learn grid-search with cross-validation [6] was used. The hyperparameter grid was made by choosing different possible parameter values, which the function then used in order to find the optimal combination that would give the best results.

```
model = Pipeline([
        ('sampling', SMOTE(random_state = 42)),
        ('classification', RandomForestClassifier(random_state=42))])

parameters = {'classification__criterion': ['gini', 'entropy'],
              'classification__max_depth':[2,3,5,6,8,10,12,14,15],
              'classification__min_samples_split': [2,4,6,8,10,12,14],
              'classification__bootstrap': [True],
              'classification__n_estimators':[10,20,30,40,50,60,70,80,90,100],
              'sampling__sampling_strategy': ['not majority'],
              'sampling__k_neighbors': [1]}

op_dt = GridSearchCV(model, parameters)
trainY, trainX = train['type'], train.loc[:, train.columns != 'type']
op_dt.fit(trainX, trainY)
```

Figure 9: The code that made the grid search cross-validation

The optimized hyperparameters which were found to be: a maximum depth of 10, minimum sample split of 4 and number of trees to 70.

5.4 Results for the Random Forest

The final model had an accuracy of 78.46%, a recall of 76.89% and an F1-Score of 70.53%.

From figure 10 it can be seen that class 5 *container* and 6 *tableware* performed well. Class 1 *window from building (float processed)* and 2 *window from building (non-float processed)* both had a high accuracy too, as expected due to the high representation in the training data.

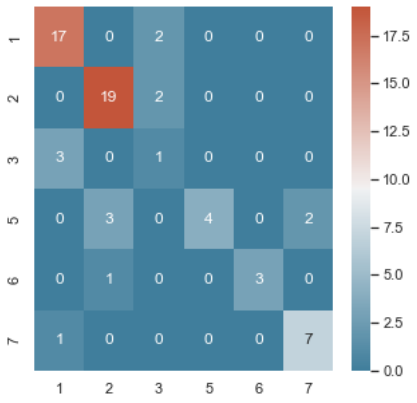

Figure 10: Confusion Matrix of the Random Forest Classifier

6 Discussion

6.1 Decision Tree

A decision tree can easily overfit to the training data, therefore having a high variance towards the training data with a low bias as a consequence. Therefore regularisation methods are an important aspect of training any decision tree. The regularisation used in the final decision tree model included maximum depth and minimum samples per split.

To further make sure the decision tree did not overfit, other regularization methods could have been implemented. Pre-pruning, for example, checks if the split contributes positively by reducing the cross-validation error. If a split does not make a significant change, the pruning undoes the split, and prevents the node from splitting in the final model [11].

The data itself can also impact the performance of the decision tree. Since the classes were imbalanced, namely, there were few data points for classes 3, 5, 6 and 7: *window from vehicle*, *container*, *tableware* and *headlamp*, SMOTE oversampling was used. SMOTE oversampling can cause that the minority class has more influence on the training of the model and therefore may be the cause of the low recall for classes 3 *window from vehicle* and class 5 *container*. On the other hand SMOTE can create synthetic values that may not occur in a real world observation. This may cause the classes to become more overlapped, if the observations of the classes lie close to one another in the feature space.

The accuracy of 64.62% is fairly low. This is most likely due to the data being difficult to separate linearly. Since the decision tree can only make linear splits, linear discriminant analysis LDA could have been implemented to project the data onto a subspace that maximised class separability and reduced dimensionality.

Overall the decision tree could have been improved using some of the above methods, but on the other hand the decision tree is susceptible to high variance and a small change in the data, potentially causing overfitting.

6.2 Neural Network

As previously mentioned, in practice the self implemented neural network did not work. The models trained took a long time and would usually classify all unseen data to the same class. This was due to a number of reasons, most likely mainly due to a slow back propagation function that limited the number of layers, neurons in each layer, and epochs that could be used due to high training time. Additionally there was most likely an error in the differentiation of the loss function as loss is expected to continually decrease and stabilise at a minimum during batch GD, whereas loss on some models would steadily increase even with very small learning rates. This is shown in figure 8.

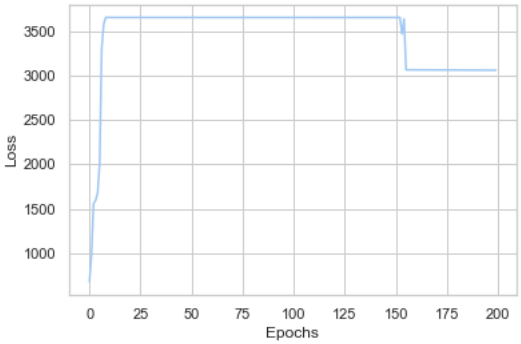

Figure 11: Loss does not Behave as Expected, A Batch GD example

Assuming the neural net had worked as planned, other improvements could have been implemented.

One such improvement would have been using dropout [12]. Dropout is a regularization method that involves temporarily ignoring or "dropping out" a certain percentage or number of neurons in each layer of the network. This leads to a reduction in interdependent learning within the network, making each neuron more robust and less dependent on specific neurons that surround it. Drop out would have been especially important as the number of observations given in the training set was small so overfitting was more likely.

6.3 Random Forest

One of the benefits with the random forest is the fact that each tree is created using bagging. This ensures that classes with few observations like *tableware*, can be classified. On the other hand, since class 1 *window from building (float processed)* and class 2 *window from building (non-float processed)* have many observations the chance of them dominating the classification of the forest is greater than for the classes with less observations. This had partially been prevented by the use of SMOTE, like in the decision tree. The use of SMOTE meant that, although the same amount of observations were present in the training data, the richness and quality of the oversampled data was most likely lower. Another thing that might also have helped is LDA. As described for the decision tree, LDA could help each tree find linear separability, thus improving the splits in the forest.

Another alternative method that could have been used to prevent the forest from being dominated by the majority classes, would have been changing the majority voting system to a weighted voting system. In such a case the classes with fewer observations would have votes that counted for more than the classes with many observations.

6.4 Comparison

From the confusion matrices it is clear that all three models had problems with overclassifing class 5 *container*, since the recall for class 5 was low on all the classifiers. Most commonly class 2 *window from building (non-float processed)* and class 7 *headlamp* were miss-classified as class 5. All of the models had a low recall for class

3 *window from vehicle* and most had low accuracy; class 3 observations were most commonly miss-classified as class 1 *window from building (float processed)*. The general performances for class 1 *window from building (float processed)*, class 2 *window from building (non-float processed)*, class 6 *tableware* and class 7 *headlamp* were good in terms of both recall and accuracy.

The random forest classifier RFC was the best of the implemented classifiers. This can be seen by looking at the results, while having a similar F1 and Recall score to the neural network, the accuracy was almost 10 percentage points higher for the RCF. It is also clear that the RFC was better than the decision tree as it scored over 10 percentage points higher in all measures. In addition to this, it can be seen from the results table below, that the neural network was better than the decision tree, with a 5 percentage points higher accuracy, a 16 percentage points higher recall and a 12 percentage points higher F1 score.

The results for the different classifiers:

Classifier	Accuracy	Recall	F1-Score
Decision Tree	64.62%	61.8%	58.22%
Sklearn NN	69.23%	77.71%	70.76%
Random Forest	78.46%	76.89%	70.53%

A reason that the decision tree was the worst model may have been that the classes were not easily separable. Therefore it was difficult for the tree to separate classes without any help from pre-processing methods like LDA. In addition to this, it seems that the model did not cope well with unbalanced data, despite being trained on SMOTE oversampled data. The oversampled data was most likely not as rich as the original observations, thus not improving the tree.

A random forest is better at capturing the variety of the data. This happens due to the bagging process, each tree in the random forest is diverse and captures different aspects of the observations. This diversity in trees leads to better classification, and thus the model had the highest accuracy, recall and F1 scores out of all of the models trained.

The neural network, although worse-scoring than the random forest, has potential to be a very good classifier. The neural network has the ability to generate the most complexity. Neural networks are prone to overfitting. If more data had been provided the neural network overfitting could have been reduced. Thus, models could have been generated with lower variance but also less bias than would have been seen in low variance, high bias models trained with the low amount of data available for this project.

In conclusion the random forest classifier performed best, followed by the neural network, and lastly the decision tree.

References

[1] Jason Brownlee. Tour of Data Sampling Methods for Imbalanced Classification, 01 2020.

[2] sklearn.tree.DecisionTreeClassifier.

[3] sklearn.ensemble.RandomForestClassifier.

[4] sklearn.neural_network.MLPClassifier.

[5] Pipeline — Version 0.8.1.

[6] sklearn.model_selection.GridSearchCV.

[7] Chi-Feng Wang. The Vanishing Gradient Problem - Towards Data Science, 12 2021.

[8] Kenneth Leung. The Dying ReLU Problem, Clearly Explained - Towards Data Science, 12 2021.

[9] R.P. Palm. nanograd.py, 08 2021.

[10] Jason Brownlee. Loss and Loss Functions for Training Deep Learning Neural Networks, 10 2019.

[11] Jake Hoare. Machine Learning: Pruning Decision Trees, 11 2020.

[12] Jason Brownlee. A Gentle Introduction to Dropout for Regularizing Deep Neural Networks, 08 2019.