Machine Learning BSMALEA1KU - Forensic Identification of Glass Fragments

Nicolai Reimer & Oliver Hellum

nrei@itu.dk, olihe@itu.dk

1 Data

1.1 Introduction

The data includes information about glass fragments from 6 different types of glass. That being :

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 dataset is limited and includes a total of 214 observations and 9 features. It holds information about refractive index (RI) and the chemical composition of each glass fragment. The features can be seen here:

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

In this analysis the data will be seperated into a training set consisting of 149 observations and a test set including 69 observations. The training set is considered small, which can be a problem for machine learning methods, since it might be fitting to random noise. This problem can be minimized; the steps taken towards this will be discussed in the next sections. First it is relevant to look at a summarization of the number of observations in each class in the training set. This can be seen in the table below:

Class	1	2	3	5	6	7
Numbers of observations	49	53	12	9	6	20

The table clearly shows that class 3,5 and 6 only include an insignificant number of observations. This unbalance can be a problem, since it might prevent the methods to generalize well. In order to minimize this effect we are going to use K-fold validation to prevent overfitting to single datapoints in different classes.

Furthermore for dimensionality reduction, we are looking at two different methods, namely Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA). This will be used to project the data into a smaller subspace with fewer dimensions by creating new features and removing insignifant ones to help prevent overfitting.

1.2 PCA vs. LDA

In PCA, new axes are created in the direction of biggest variance. This means that most of the variance is captured in the first principle component, then the second principle component and so forth. This way, it is possible to reduce the feature space by throwing away the last principle components that do not account for much variance in the data.

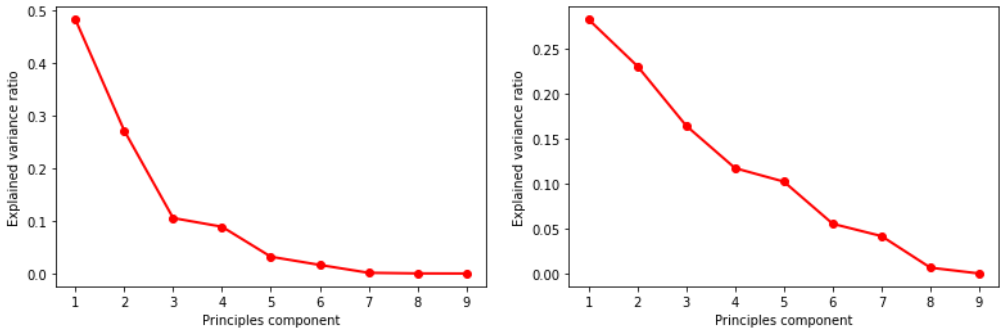
A downside of PCA is that it is an unsupervised learning method, meaning it cannot differentiate

between classes. So though the first principle components account for the most variance, they do not attempt or maximize (or even preserve) separability of classes. LDA, on the other hand, attempts to do just that.
LDA maximizes between-class variance while minimizing within-class variance, thus maximizing class-separability. Similarly to PCA, the first discriminants account for most class-separability while the last discriminants account for the least.

However, LDA is more likely to overfit the data compared to PCA. This is because the covariance of features are estimated for all the classes before creating an estimate of a pooled covariance-matrix. So LDA is more likely to capture random fluctuations in data in individual classes. This especially holds true for classes with few observations, like "Tableware", "Container", or "Window from vehicle". We therefore prefer using PCA over LDA due to our limited number of observations. Another possiblity would also be to first use PCA and then LDA, to reduce the risk of overfitting the LDA by first reducing the feature space. However, the problem with too few observations in some classes still persists, so we choose to rely solely on PCA for dimensionality reduction.

1.3 Standardized vs. Non-standardized

In theory, the data should be standardized so that the PCA does not weight features on a bigger scale higher. Standardizing ensures that each feature is on a similar scale. This way, PCA creates principle components in directions of most variance with respect to the different scales of the features. However, we noticed that we sometimes got better results without standardizing the data first. Therefore, we include performance for every model with and without standardization. Below are scree-plots of the explained variance ratio of different principle components. The left scree-plot is PCA without standardizing:



As we can see, we lose much more of the explained variance when throwing away principle components when standardizing. When standardizing, it appears that the only PC we can throw away "for free", is the last one, while when not standardizing, we can throw away at least the last three.

1.4 Hyperparameter-tuning

When tuning hyperparameters, we want some data to validate the performance on. However, since we do not have much data from the beginning, we find it more prudent to use K-fold validation. This way, all our training data is used for training and validation. This is done using scikit-learn's GridSearchCV or RandomizedSearchCV, depending on how long it will take to fit the models. We use

5-fold cross validation. In principle, it would be better to use more folds due to the few observations in the training data. However, we found that using more folds did not result in choosing hyperparameters with substantially better performance. Thus, we decided to go with the faster 5-fold cross validation.

For tuning hyperparameters we decide to choose the model with the highest validation F1 macro score. The F1 macro score lets us try to maximize a combination of precision and recall. We chose F1 macro over F1 weighted, since we wanted every class to have equal weighting, even if their frequencies differ. It could be argued that since this is to be used in case work, weighing precision higher than recall might be a good idea. However, we do not believe we have enough domain knowledge to properly weigh between the two. Therefore, we are instead using the simple F1 macro as evaluation score.

For methods with many possible values for hyperparameters, we try to start with wide intervals (for numerical hyperparameters) and use RandomizedSearchCV with verbose=2. The parameter random_state is always set to the seed 42. This way, we can see the hyperparameters tried and which combination performed best. Afterwards, we would do a new search, limiting hyperparameters to be around the value we got from the former search. Depending on the algorithm, this step would use either another RandomizedSearchCV or GridSearchCV. This approach means that we are probably not finding the global optimum, but a local one. Included in the report is the parameter grid for the final search yielding our results to improve reproduceability.

2 Methods

2.1 M1: Linear Discriminant Analysis

2.1.1 Theory of Linear Discriminant Analysis

LDA is a generative model for linear classification, meaning the resulting decision boundaries are linear. It is the method that follows from the following assumptions:

- The class conditionals $P(X = x|Y = k)$ are Gaussian with individual class means (multivariate normality)
- The class conditionals share the same covariance matrix (homoscedasticity)
- Observations are independent of each other

From decision theory, we know that we need to estimate the class-posteriors $P(Y = k|X = x)$. Let π_k be the prior probability of class k, and let $f_k(x)$ be the probability density function for the class conditional for class k. Then Bayes theorem gives us:

$$P(Y = k|X = x) = \frac{p(x, y)}{P(X = x)} = \frac{f_k(x)\pi_k}{\sum_{i \in K} f_i(x)\pi_i}$$

where K is the set of classes. We see that we makee assumptions about the joint probability $p(x, y)$ to obtain the posterior probabilities to then later create discriminant functions for choosing a prediction.

This is what makes LDA a probabilistic generative model. If the assumptions of multivariate normality and homoscedascticity holds, the class density is

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)'\mathbf{\Sigma}^{-1}(x-\mu_k)\right)$$

where p is the number of features (i.e., the dimension of the vector x).
Say we just want to make a prediction. Since the denominator $\sum_{i\in K} f_i(x)\pi_i$ is the same for every class, it does not make a difference for our prediction:

$$P(Y = k|X = x) \propto p(x, y) = f_k(x)\pi_k$$

And since we then only care about, what posterior probability is the largest, we can take log (since it's a strictly monotonically increasing transformation):

$$\log p(x, y) = \log \pi_k - \frac{1}{2}(x-\mu_k)'\mathbf{\Sigma}^{-1}(x-\mu_k) + \log\left(\frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}|^{1/2}}\right)$$

Since the last term does not differ between classes, we can define the linear discriminant functions:

$$g_k(x) = 2\log \pi_k - (x-\mu_k)'\mathbf{\Sigma}^{-1}(x-\mu_k)$$

For Quadratic Discriminant Analysis, we relax the assumption of homoscedasticity, resulting in the class densities. This gives us the following discriminant functions:

$$g_k(x) = 2\log \pi_k - \log|\mathbf{\Sigma}| - (x-\mu_k)'\mathbf{\Sigma}_k^{-1}(x-\mu_k)$$

In LDA, we only need to estimate π_k, μ_k and $\mathbf{\Sigma}$. For QDA, on the other hand, we need to estimate π_k, μ_k and $\mathbf{\Sigma}_k$ for every $k \in K$. This means that QDA has a higher risk of overfitting. This is especially true for classes with few observations, like "Tableware", "Container", or "Window from vehicle". On the other hand, QDA is more flexible, since it does not have the assumption of homoscedasticity. This means that LDA can suffer from high bias, while QDA suffers from high variance. However, due to the few observations in certain classes, we have decided to go with LDA, even though the differences in covariance matrices can be substantial. We calculated the largest differences in emperical class covariance matrices and sorted them:

```
[5.31597803 5.0404375  4.23681417 4.11273026 3.93859741 3.66305688
 3.61690869 3.22526803 2.9497275  2.73534964 2.02202026 1.80914335
 1.58666286 1.55121585 0.27554053]
```

Here, we do see some fairly big differences between elements of the emperical class covariance matrices. It should also be noted, that the assumption of homoscedasticity in LDA does not mean it will necessarily perform worse if this assumption does not hold true. This can especially hold true, when transforming the data through PCA beforehand.

2.1.2 Implementation

Since we do not know the parameters of the Gaussian distributions, we need to estimate them using out data:

$$\hat{\pi}_k = N_k/N$$

$$\hat{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{k,i}$$

$$\hat{\mathbf{\Sigma}} = \frac{1}{N-K} \sum_{k \in K} (X_k - \mu_k)(X_k - \mu_k)'$$

where N_k is the number of observations for class k, $x_{k,i}$ is the $i'th$ observation in subset of X where the class is k called X_k. The reason for the factor $\frac{1}{N-K}$ is to make the pooled covariance estimator unbiased. The denominator is the degrees of freedom (because we have K estimated means). However, in scikit-learn, the following estimator is instead used:

$$\hat{\mathbf{\Sigma}} = \frac{1}{N} \sum_{k \in K} (X_k - \mu_k)(X_k - \mu_k)'$$

This can be seen on line 92 of scikit-learn.covariance._emperical_covariance.py. So for simplicity of tests of our implementation, we use the same estimator.
Using those three estimators, we calculate the value of the discriminant function of each class and choose the class whose discriminant function gives the highest value, i.e $\hat{y} = \arg\max_k g_k(x)$.

2.1.3 Tests

We test our implementation by making sure we estimate the same priors, class-means and covariance-matrix, as well as produce the same predictions as the implementation in scikit-learn. Many discriminant functions can be used that will give different values, meaning it does not make sense to test if we get the same as with scikit-learn. The easiest way to verify that we get the same priors, means and covariance, is to compute the difference between our estimations and scikit-learns and see, if every entry is 0. It should be noted that our implementation is equivalent to scikit-learns LinearDiscriminantAnalysis using the 'eigen' solver. We tested using the whole training data and here are the results:

```
Difference in priors:
[0. 0. 0. 0. 0. 0.]

Difference in means:
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]

Biggest absolute difference in covariances:
2.220446049250313e-16
```

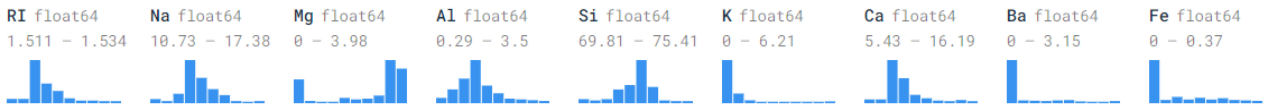
```
Difference in predictions:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0]

Difference in predictions on test data:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

As we can see, our implementation get the same results as the one from scikit-learn, when the parameter "solver" is set to "eigen". Due to the size of the covariance matrix, only the biggest absolute difference between the covariance matrices is included in the screenshot for convenience. The difference between the two covariance matrices is calculated in the attached Jupyter-notebook file.

2.1.4 Hyperparameter-tuning & Results

LDA does not itself have any hyperparameters. However, we can choose the number of principle components from PCA to feed into LDA, making it a hyperparameter. To tune this, we used scikit-learn's Pipeline to pack StandardScaler (for standardizing), PCA and our LDA classifier. This pipeline is used in GridSearchCV with F1 macro as evaluation metric. For values to evaluate for with "n_components"

in PCA, we try all possible values, i.e. $\{1, ..., 9\}$. We expect LDA to perform fairly, since it usually generalizes very well; something that works well with little data. However, we do not expect it to perform exceptionally due to the emperical covariance matrices not being similar. Additionally, the features appear not to follow a Gaussian distribution (we used DeepNote for these histograms):



Our results are as follows:

	PC's	Best score
Standardized	9	0.516
Non-standardized	6	0.531

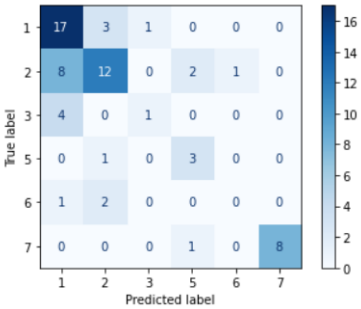
When testing on the test data, we get the following performance for the non-standardized model:

'F1 Macro': 0.49435393607760236, 'Accuracy': 0.6461538461538462, 'Precision Macro': 0.4681337181337182, 'Recall Macro': 0.542558086036346

And for the standardized model, we get:

['F1 Macro': 0.513153879437954, 'Accuracy': 0.6307692307692307, 'Precision Macro': 0.5388888888888889, 'Recall Macro': 0.5283586381412467]

We see that standardizing the data before using PCA and LDA gives us the best F1 macro and precision macro scores, making it the best of the two models under our evaluation criteria. Looking at the confusion matrix:



we see that the model has issues with predicting samples from class 6 and class 3. It makes sense for it to have some trouble with class 6, given the small frequency of the class resulting in small priors. The confusion matrix also shows us that class 3 seem to not be linearly separable from the other classes. We can therefore expect that other classifiers will probably also have some trouble with class 3.

2.2 M2: Decision Tree

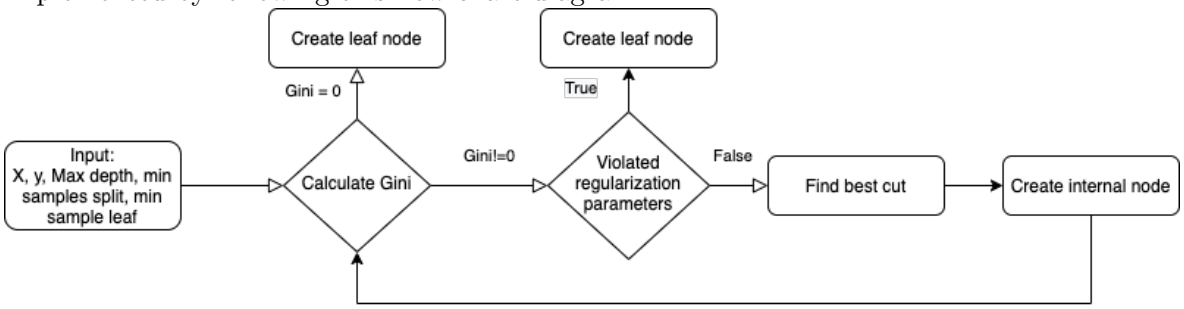
A decision tree is a supervised machine learning algorithm. It has many different implementations and can be used as a classifier and a regression algorithm. This report focuses on the binary decision tree as a n-way classifier. In its simplest form this tree is a set of continues true/false questions, that will lead to a result given an input. Therefore the decision boundary will consist of only vertical or horizontal lines. The decision tree itself consist of three different parts: a root node, internal nodes, and leaf nodes. The root node and the internal nodes both hold a question, that can be either true or false. The difference is that the root node is the first question asked, while the internal nodes are the rest of the questions. These nodes will always point to another node. The leaf nodes hold the result, and they are the last nodes of the tree.

When a binary tree is made, every internal node cuts the dataset into 2 smaller pieces (Children). This is a very important step of the algorithm, since it is crucial that the cut is made in the best way possible. In order to measure the quality of a cut, the gini impurity will be used. The algorithm will run through all the possible split in the dataset taking the features into account, and for every split the gini impurity will be calculated. From the split with the highest gini impurity a new node will be made. This can be a root node, if it's the first node, an internal node, if it's not, and a leaf node if a child is pure. The algorithm will repeat itself recursively on every subset until every datapoint is separated (Unless it is impossible to separate 2 datapoints).

Decision trees are very prone to overfitting, since the algorithm will continue until every datapoint is separated. Because of that, it is important to add regularization parameters for this method. In this implementation the users can regularize the max depth of the tree, the minimum samples in a leaf and the minimum samples in a split.

2.2.1 Implementation of Decisions Trees

The algorithm to make decision trees will be implemented by hand, and in this section the key points will be presentated. The code itself can be found in the attach file to the report. If the reader is only interested in the results of the report, we suggest skipping this section. Firstly, our algorithm can be implemented by following this flow-chart diagram.[1]



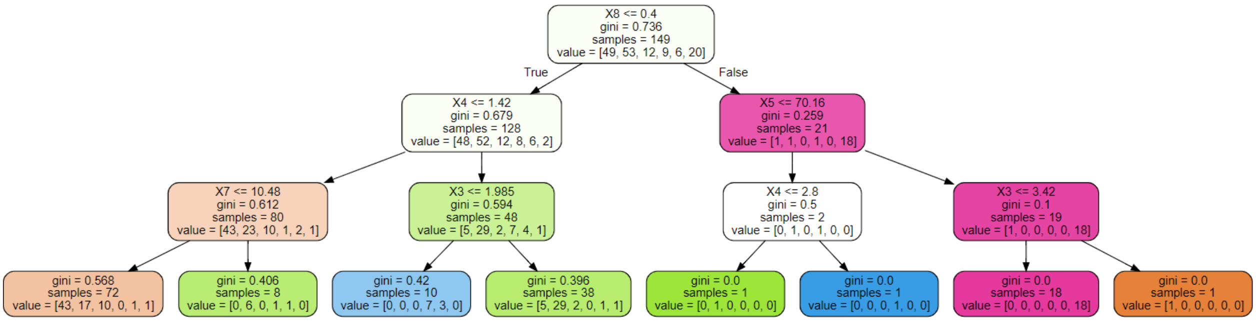
The important points to notice in this diagram are that the algorithm is recursive, and it's going to call it self untill it has two leafs on every branch. The algorithm can be implemented without the regularization parameters Max depth, min samples leaf and min samples split, but it will then be overfitting the training data. The datastructure of the tree can be implemented different ways. We have decided to create the leaf nodes and internal nodes as classes, because it creates a intuitive tree like datastructure.

2.2.2 Test

To verify our implementation of decision trees classifies correctly, it will be testes against a decision tree classifier implemented by scikit-learn. This will be done by making different trees with different regularization parameters, and check whether the two implementations produce the same trees and the same predictions. We tested for a each of the 3 hyperparameters, as well as the combination with max depth = 3, min samples split = 8, min samples leaf = 2. The following are screenshots of

[1] Inspiration for data structure and visualization of tree is taking from https://towardsdatascience.com/algorithms-from-scratch-decision-tree-1898d37b02e0

visualizations of a tree with max_depth=3 and default for thee other two:




```
X8 <= 0.4
True:
  X4 <= 1.42
  True:
    X7 <= 10.48
    True:
      Predict: 1
    False:
      Predict: 2
  False:
    X3 <= 1.9849999999999999
    True:
      Predict: 5
    False:
      Predict: 2
False:
  X5 <= 70.16
  True:
    X1 <= 1.5231949999999999
    True:
      Predict: 5
    False:
      Predict: 2
  False:
    X3 <= 3.42
    True:
      Predict: 7
    False:
      Predict: 1
```

As can be seen, the trees only diverge when it comes to an internal node who's children are pure leaf nodes We subtract the predictions from each other and get

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

This shows clearly that two implementations predicts the same in every case. If the tree is change to the parameters max depth = 10, min samples split = 20, min samples leaf = 2 or max depth = 10, min samples split = 2, min samples leaf = 10, the same conclusion is drawn. If we compare the prediction made by our implementation and the one made by scikit-learn on the test set, we get:

```
array([0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

It is worth noticing that the two models differ in prediction 4. This can be explained by the randomness in the model, when a leaf node is made, since some cases datapoints are able to be completely seperated by different features. This fact can lead to slightly different trees, which again might lead to slightly different predictions such as seen here.

2.2.3 Hyperparameters and results

In this machine learning method we are going to set the hyperparameters Max depth, min samples leaf and min samples split. We expect the model to perform better than LDA, since the classes are not linear separable, and the decision tree is able to better catch some of these patterns. Using gridsearch

	W standardize and PCA	W/O Standardize	W/O PCA and standardize
max_depth	{4,...,9}	{2,...,9}	{3,...,8}
min_samples_leaf	{1,...,2}	{1,...,2}	{1,...,4}
min_samples_split	{2,...,11}	{3,...,11}	{2,..., 9}
PC's	{5,...,9}	{5,...,9}	N/A

we get the results

Decision Tree	W/O standard scaling	W standard scaling	W/O scaling and PCA
Best score	0.448	0.479	0.506
max_depth	7	8	5
min_samples_leaf	1	1	1
min_samples_split	9	3	6
n_components	5	5	N/A

The search shows that the best model gives a F1 macro score of 0.506. All of the combinations of hyperparameters will be applied to the model, and the performance will be tested on the test set. For the non-standardized we get:

```
{'F1 Macro': 0.5321375372393248, 'Accuracy': 0.6, 'Precision Macro': 0.538938492063492, 'Recall Macro': 0.5731251437773176}
```

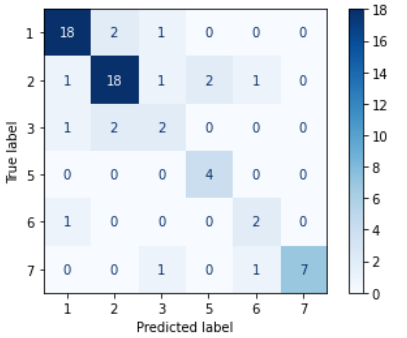
And the standardized

```
{'F1 Macro': 0.6340568808653915, 'Accuracy': 0.7538461538461538, 'Precision Macro': 0.6299603174603174, 'Recall Macro': 0.6701173222912353}
```

Finally without PCA and standardization

```
{'F1 Macro': 0.7172619047619048, 'Accuracy': 0.7846153846153846, 'Precision Macro': 0.7069985569985571, 'Recall Macro': 0.7473659995399126}
```

It is seen that the performance increases on the test set if we do not use PCA and scaling before training our model. This might be a consequence of a "bad rotation" of our data with PCA, since the decision tree is only able to make vertical og horizontal decision boundries. If we look at the results in a confusion matrix we get:



From the confusion matrix we can see that the classifier in generel has a high performance. It has a F1 score on 71.7 % and it is classifying correct 78% of the time. It does not look like there is a particular problem in the fit, since the errors are spread out on different classes.

2.3 M3: Support Vector Machine

2.3.1 Theory

The idea behind Support Vector Machines (SVM) is to choose the linear decision boundary that maximizes the distance between the decision boundary and closest point from each class, i.e. maximizing the "street". When no point is allowed on the street, this is called hard margin classification. This, however, probably will not generalize very well for classes close together, and it will not work at all for overlapping classes. Therefore, soft margin classification is used, where the goal is to keep the street as large as possible while limiting margin violations. In scikit-learn, this is done using the C hyperparameter. A larger C means heavier punishment for margin violations, thus a more narrow street.

This is a constrained convex optimization problem (in particular, it is a quadratic programming problem with linear inequality constraints). It turns out that the solution to the dual problem solely depends on the dot product between the observations. Right now, our SVM model can only produce linear decision boundaries. However, if we transform our features and add them to our feature space, we can create a higher dimensional feature space. Producing linear decision boundaries in the higher dimensional space results in a non-linear decision boundary when going back to our original space. However, since we only need the dot product between observations in the transformed space, we can instead use a kernel function to calculate the dot product directly without ever transforming the space. This is called the kernel trick. Using scikit-learn, we'll be looking at linear, polynomial and rbf kernels:

- Linear: $K(x_i, x_j) = \langle x_i, x_j, \rangle$
- Polynomial: $K(x_i, x_j) = (\gamma\langle x_i, x_j\rangle + r)^d$, where γ is set by the parameter "gamma", r is set by "coef0" and d is set by degree.
- RBF: $K(x_i, x_j) = \exp\left(-\gamma||x_i - x_j||^2\right)$, where γ is set by "gamma"

2.3.2 Hyperparameter-tuning & Results

For tuning parameters, we used scikit-learn's Pipeline to apply standardization and PCA. We go through each of the three kernels and tune hyperparameters for that particular kernel, recording the results. A fit takes some times to calculate, especially if the kernel is not linear. Furthermore, we also have many hyperparameters to tune, so for polynomial and RBF kernels, we use a randomized search instead of a grid search. For the linear kernel a grid search is more than fast enough.

Due to SVM being more flexible than LDA with its potential for non-linear decision boundaries (lower bias), and it's use of regularization (potentially not that much higher variance), we expect it to outperform LDA. For the last randomized search, we had the following options for the hyperparameters (with and without standardizing), when applicable:

svc__kernel	{'linear', 'poly', 'rbf'}
pca__n_components	{1,...,10}
svc__C	{10, 11,..., 25}
svc__class_weight	{None, 'balanced'}
svc__degree	{1,...,10}
svc__coef0	{1,...,10}
svc__gamma	{'scale', 'auto'}

We find that the optimal hyperparameters are:

	PC's	Kernel	Gamma	Degree	Coef0	C	Class weight	Best score
Non-standardized	4	Polynomial	Scale	2	1	17	None	0.671
Standardized	9	RBF	Auto	N/A	N/A	20	None	0.678

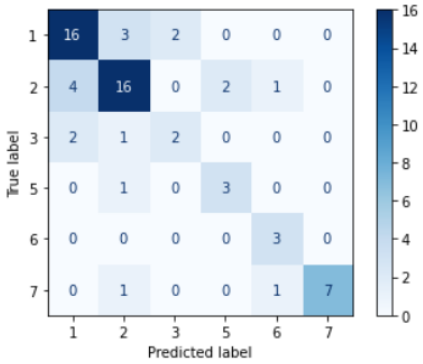
This gives us the following performance on the test set for the non-standardized:

'F1 Macro': 0.6605769230769231, 'Accuracy': 0.7538461538461538, 'Precision Macro': 0.6415546594982079, 'Recall Macro': 0.7303887738670348

And for the standardized:

'F1 Macro': 0.698568044788975, 'Accuracy': 0.7230769230769231, 'Precision Macro': 0.6924242424242424, 'Recall Macro': 0.7308891189325971

Again, we see the standardized model out-perform when it comes to F1 macro and precision macro scores. Looking at the confusion matrix for the standardized:



we see it still having trouble with classifying class 3. However, it performs significantly better on observations from class 6 and 2 when compared to LDA. On the other hand, it is not as strong on classes 1 and 2 when compared to the Decision Tree.

2.4 M4: K-Nearest Neighbour

K-Nearest neighbor is a supervised machine learning algoritm. In this section we will be looking at K-nearest neighbor as a classifier due to our glass classification problem. It is a "model-free" method, that only take the k nearest points into account, when making a classification, which gives it very flexible boundries. The main idea behind the method is that datapoints within a class tend to look alike and cluster together. Therefor by looking at the k nearest neighbors you can make a qualifednguess on different data inputs classes. Since the algorithm is looking at distance between points, when finding the nearest neighbors, it is important to standardise the data and find an approprotate metric. Otherwise you will find that wrong features might be more influential than others when making a classification, which can lead to poor performance.

2.4.1 Hyperparameters and results

To tune this machine learning method we are going to set the hyperparameter numbers of neighbors, k, and n, the numbers of dimension in PCA. Furthermore we are going to do a LDA transformation on the PCA components, where we are going to collapse the data into 2 dimension, with fischers LDA. The collapse into 2 dimension will be done to visualize the boundries optained by the method. The PCA-LDA procedure is made to maximize the information passed on into the 2 last dimensions. The performance is maximazied by scikit-learns gridssearch where K will run between 1-10 and numbers of components in PCA will run between 1-9. We also made a grid search for the non-standardized model without the cap on discriminants. We expect the method to generally work well due to its trait of being "model-free" probably working fairly well little data. Furthermore, it is capable of understanding some non-linear patterns in the data:

K-nearest neighbor	W/O standard scaling	W standard scaling	Non-standardized with free discriminants	Standardized, free discriminants
Best score	0.556	0.519	0.625	0.643
n_neighbors	5	3	1	1
PCA_components	5	4	4	4
LDA_components	2	2	4	4

The grid search shows that the best model gives a F1 macro score of 0.556 for 2 features. These hyperparameters will be applied to the model, and the performance will be tested on the test set. For the non-standardized we get:

{'F1 Macro': 0.5009751773049645, 'Accuracy': 0.6, 'Precision Macro': 0.5156954156954157, 'Recall Macro': 0.5019036116862203}

And for the standardized

{'F1 Macro': 0.5191859066859067, 'Accuracy': 0.5692307692307692, 'Precision Macro': 0.550595238095238, 'Recall Macro': 0.4992293535771796}

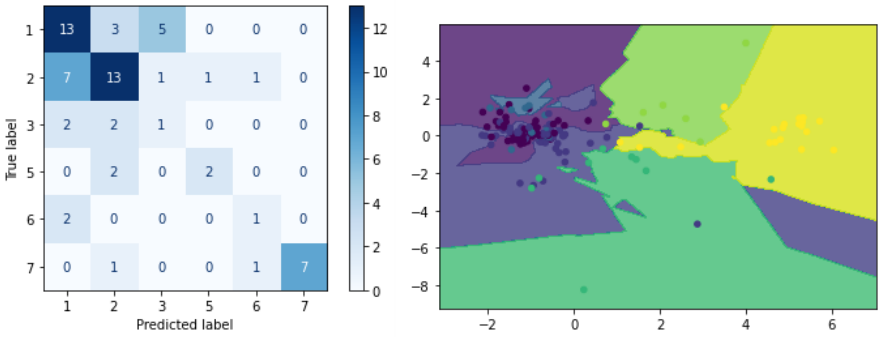
For the full model with standardizing we get:

{'F1 Macro': 0.7147727272727273, 'Accuracy': 0.7538461538461538, 'Precision Macro': 0.6927318295739348, 'Recall Macro': 0.7798021624108581}

And without standardizing:

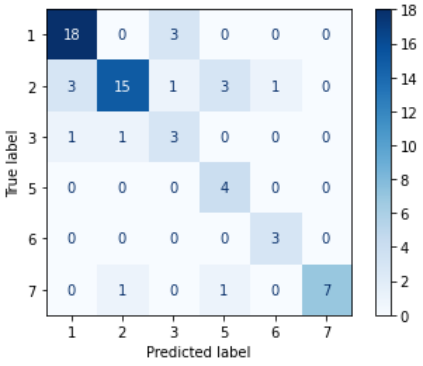
{'F1 Macro': 0.747669804355851, 'Accuracy': 0.7692307692307693, 'Precision Macro': 0.7298510313216195, 'Recall Macro': 0.8145157579940189}

It is seen that the performance increases on the test set if the data is scaled, if we decide to stricly have two discriminants. Otherwise, standardizing actually hurs the model. Also it is worth noticing, that the F1 macro and the validation set and the test set is close, which shows us, that the model is not overfitting. Therefore we are going to take a look at the model, where the data is scaled. We start by plotting can plot the confusion matrix and the decision boundries.



From the confusion we can see that the classifier, has problems classifying class 1 and 2, whereas it is easier to classify class 7. By looking at the obtained decision boundries, this can problem can easily be seen, because the 2 classes are clustered together in the purple area. The rest of the classes, are more easyly distinguished from each other. Nevertheless the results, are quite satisfying taking into account, that it seperation is done by only two features.

If we look at the confusion matrix for the full model without standardizing:



we see that it does not have classes it has particular problems with, although it seems to be predicting class 3 a little too often. It should also be noted that given more data, it is likely that a model with a higher n_neighbor will come out on top instead.

2.5 M5: Random Forest

A random forest classifier is a supervised machine learning algorithm. It is an expansion of a the decision tree classifier explained in M2. The method works by creating n random trees that works like an ensemble. Each individual tree has its own class-prediction on an input, and the class with the most predictions in total will be the guess of the random forrest. It is important that there is a low correlation between these trees, since this will prevent a single tree from gaining a substantial weight in the counting. By expanding the decision tree classifier to a random forest classifier, it is possible to prevent some of the issues attached to a single tree, mainly being overfitting. In a random forrest each tree protect each other from a singles tree's overfitting problem. Due to the low correlation between each tree, it is very unlikely that the majority will be overfitting the same part of the data. These attributes generally makes the random forest classifier outperform a single decision tree. The Random Forest Classifier used in this report, is the classifier implemented by scikit-learn.

2.5.1 Hyperparameters and results

In this machine learning method we are going to set the hyperparameters Max depth, min samples leaf and min samples split. The performance is maximazied with the help of scikit-learn's Gridsearch. The rest of the parameters can run between:

	W standardize and PCA	W/O Standardize	W/O PCA and standardize
max_depth	{7,...,13}	{5,...,9}	{7,...,12}
min_samples_leaf	{1,...,5}	{1,...,4}	{1,...,5}
min_samples_split	{3,...,8}	{2,...,7}	{3,..., 11}
PC's	{5,...,9}	{5,...,9}	N/A

This gives the results:

Random forest	W/o standard scaling	W standard scaling	W/O scaling and PCA
Best score	0.525	0.554	0.772
max_depth	7	9	11
min_samples_leaf	1	1	1
min_samples_split	3	4	6
n_components	9	8	N/A

The grid search shows that the best model gives a F1 macro score of 0.772. All of the combinations of hyperparameters will be applied to the model, and the performance will be tested on the test set. For the non-standardized we get:

{'F1 Macro': 0.7210041592394534, 'Accuracy': 0.7846153846153846, 'Precision Macro': 0.7390206411945542, 'Recall Macro': 0.7242178513917644}

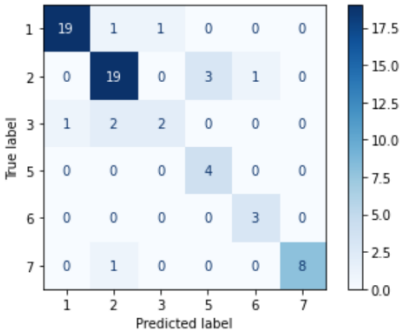
And the standardized

{'F1 Macro': 0.7628142299164501, 'Accuracy': 0.8153846153846154, 'Precision Macro': 0.8208333333333333, 'Recall Macro': 0.728818725557856}

Finally without PCA and standardization

{'F1 Macro': 0.7964180466363736, 'Accuracy': 0.8461538461538461, 'Precision Macro': 0.7940303657694961, 'Recall Macro': 0.8366229583620889}

It is seen that the performance increases on the test set if we do not use PCA and scaling before training our model. If we look at the results in a confusion matrix we get:



From the confusion matrix we can see that the classifier in generel has a high performance with a F1 score on 0.796 and it is classifying correct 84.6% of the time. It is seen that the classifier has problems classifying class 3. It only has 2 out of 5 correct in class 3. The random forrest classification seems to perform well on the rest of the classes. Overall the results are quite satisfying, and it is likely that this method would be able to perform very well, with just a few more training examples on class 3.

3 Results

	F1 macro	Accuracy	Precision macro	Recall macro
LDA	0.513	0.631	0.539	0.528
LDA Non-standardized	0.494	0.646	0.468	0.543
Decision Tree Non-standardized	0.532	0.600	0.539	0.573
Decision Tree Standardized	0.634	0.754	0.630	0.670
Decision Tree w/o PCA	0.717	0.785	0.707	0.747
SVM	0.699	0.723	0.692	0.731
SVM Non-standardized	0.661	0.754	0.642	0.730
KNN	0.519	0.569	0.550	0.499
KNN Non-standardized	0.501	0.600	0.516	0.502
KNN $\geq$ 2 Components	0.748	0.769	0.730	0.815
Random Forest Non-standardized	0.721	0.785	0.739	0.724
Random Forest Standardized	0.762	0.815	0.821	0.729
Random Forest w/o PCA	0.796	0.846	0.794	0.836

Looking at the F1 macros, we see the Non-standardized KNN with LDA and PCA, and Random Forest w/o PCA outperforming. It makes sense that most models would outperform LDA, since the assumptions for the model do not seem to hold true. Furthermore, as could be gleaned from its confusion matrix, the classes do not seem to be linearly separable. It also makes sense that if the Decision Tree is better off not being fed principle components from PCA, the same would be the case for a Random Forest.

The Random forest outperforms the Decision Tree models, which also makes sense since it is an ensemble method of Decision Trees. All things considered, we would recommend a Random Forest without PCA, due to its superior performance in evaluations metrics F1 and accuracy. That being said, if precision should be valued considerably higher than recall, a PCA followed by Random Forest on standardized data would be preferred. It should also be noted that the performance of the Decision

Tree without PCA is not that much worse, and if its trait of being a white box model is desirable, it could also hold great value.

4 Further Considerations

4.1 If we had been given a single dataset

Had we been given the training data and test data as a single dataset, we would have to split it ourselves. However, due to the classes being heavily unbalanced, a simple shuffled train and test split could yield issues. We could risk not having enough class 6 observations in our test data, promoting models not properly capable of classifying for this class. Therefore, we would first force our training, and test data to both have at least a couple of observations from each class. Afterwards, we would simply add samples without replacement to our test data. We would choose a train test split of 70/30, valuing properly test of our models highly.

4.2 Uncertainty of classification

In forensic science, it is paramount to have an idea about the uncertainty of a classification. For this, we would recommend using the predicted probabilities from each model. For SVM (that cannot in and of itself predict probabilities), we would recommend looking at the decision function to get an idea of how certain the classifier is. Since this is to be used for case work, the models might be too uncertain as it stands, and more data is needed to be able to create a strong enough model.