Project3 - FYS-STK3155

December 16, 2020

1 Abstract

In this project we have used a set of decision tree models to perform classification on two datasets. Decision trees are generally considered a 'weaker' machine learning method, without the same predictive power as many of the more complex models, like neural networks. It is nonetheless a very popular method, in part due to the ease of interpreting such models. What we have found in this project is that depending on the data at hand, decision trees may give more than satisfactory performance results.

We have looked at two datasets, one for mushroom classification, and one for classification of fetal health risk groups. For the former decision trees appeared to outperform logistic regression even for small trees, with the added plus that the resulting model was very easy to interpret, with rules one could learn and actually use while out picking mushrooms. While the dataset had many features, making it difficult to see what features to focus on, our decision tree illuminated some key features and accompanying values to focus on. Logistic regression on this dataset gave an accuracy of 0.958, while we with decision trees reached perfect accuracy. We found that our created decision trees to an extent mirrored the four known rules to 100% accuracy for this dataset.

For our fetal health dataset, which was also one of high dimensionality, we saw a poorer performance for a single, small tree, but by increasing the size of the tree and also adding ensemble methods we reached an improved accuracy compared to logistic regression. While logistic regression gave an accuracy of about 0.85, we managed to get as high as 0.96 when using adaptive boosting and trees of depth five. Exploring this final dataset also showed us the key pitfall when working with unbalanced datasets; a falsely elevated accuracy. We saw how simply classifying all data to the largest class gave an fairly high accuracy, while not actually being of any use.

2 Introduction

3 Data

3.1 Mushrooms[1]

The mushroom dataset contains descriptive data for (hypothetical) samples of 23 species of gilled mushrooms in the Agaricus and Lepiota Family. The samples are drawn from *The Audubon Society Field Guide to North American Mushrooms* (1981)[2]. Each species is classified as either edible (class e) or poisonous (class p), where the poisonous category includes both species known to be poisonous as well as those where edibility is unknown.

The data set contains a total of 8124 samples, each described with 22 descriptors. To reduce the size of the dataset, each attribute value is coded to a letter. These attributes are as follows:

1. cap-shape:
   bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
2. cap-surface:
   fibrous=f,grooves=g,scaly=y,smooth=s
3. cap-color:
   brown=n,buff=b,cinnamon=c,gray=g,green=r,pink=p,purple=u,red=e,white=w,yellow=y
4. bruises?:
   bruises=t,no=f
5. odor:
   almond=a,anise=l,creosote=c,fishy=y,foul=f,musty=m,none=n,pungent=p,spicy=s
6. gill-attachment:
   attached=a,descending=d,free=f,notched=n
7. gill-spacing:
   close=c,crowded=w,distant=d
8. gill-size:
   broad=b,narrow=n
9. gill-color:
   black=k,brown=n,buff=b,chocolate=h,gray=g,green=r,orange=o,pink=p,purple=u,red=e,white=w,yellow=
10. stalk-shape:
    enlarging=e,tapering=t
11. stalk-root:
    bulbous=b,club=c,cup=u,equal=e,rhizomorphs=z,rooted=r,missing=?
12. stalk-surface-above-ring:
    fibrous=f,scaly=y,silky=k,smooth=s
13. stalk-surface-below-ring:
    fibrous=f,scaly=y,silky=k,smooth=s
14. stalk-color-above-ring:
    brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
15. stalk-color-below-ring:
    brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
16. veil-type:
    partial=p,universal=u
17. veil-color:
    brown=n,orange=o,white=w,yellow=y
18. ring-number:
    none=n,one=o,two=t

19. ring-type:
cobwebby=c,evanescent=e,flaring=f,large=l,none=n,pendant=p,sheathing=s,zone=z
20. spore-print-color:
black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y
21. population:
abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
22. habitat:
grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d

In analysing this data we have mapped the letter coding to numbers, as scikit-learn did not seem to handle letters in their DecisionTreeClassifier. Our own decision tree class handles both versions, hich makes for easier to read trees.

3.1.1 Known Simple Rules

As this data set has been studied extensively, several more or less complex rules have been found for deciding whether a given mushroom is edible or not. Particularly a set of four markedly simple rules have been found that together give a 100 % accuracy on classifying poisonous mushrooms [1]:

- P_1: `odor=NOT(almond.OR.anise.OR.none)`
120 poisonous cases missed, 98.52% accuracy
- P_2: `spore-print-color=green`
48 cases missed, 99.41% accuracy
- P_3: `odor=none.AND.stalk-surface-below-ring=scaly.AND.(stalk-color-above-ring=NOT.brown)`
8 cases missed, 99.90% accuracy
- P_4: `habitat=leaves.AND.cap-color=white`
0 cases missed, 100% accuracy

3.2 Fetal Health[3][4]

A dataset of 2126 entries, each described by 22 features extracted from cardiotocogram exams on fetuses. There are no missing or `Null` values. Table 8 shows values for descriptive statistics like min and max values, mean, and standard deviation. The target value is `fetal_health` which can take one of three classes:

- Normal
- Suspect
- Pathological

as determined by three expert obstetricians.

We see from figure 9 that, not unexpectedly, most entries are in the *Normal* class. As such the dataset is quite unbalanced. Many of the features describe related values, like various descriptive statistics for the histograms from the cardiotocogram exams, making them correlated.

4 Methods

4.1 Logistic Regression

See Methods section in project 2[13].

4.2 Decision Trees

A decision tree is a type of supervised learning model that can be used for both regression and classification problems. They are named *trees* as their structure consists of a root node recursively split into nodes, or "branches", ending in the end-nodes also known as "leaves". Each split of a node is based on a choice or decision for one of the features of the data, like for instance "is height>=2.0m?".

When using a decision tree for prediction on new data we move down the tree, for each node determining whether to move left or right down the tree based on the value of the input data for the relevant "decision feature" at that node. Is the value below or above some threshold, or equal/not equal to some value? When we reach the end of the tree, one of the leaf nodes, this node tells us the resulting prediction.

Decision trees are popular models for real life problems as they produce easily interpretable models that resemble human decision making. They do not require normalization of the inputs, and they can be used to model non-linear relationships.

They are, however, prone to over-fitting and generally do not provide the best predictive accuracy. Other challenges for decision trees are that small changes in the data may lead to a completely different tree structure, and unbalanced datasets with a target feature value that occur much more often than others may lead to biased trees since the frequently occurring feature values are preferred over the less frequently occurring ones. In addition, features with many levels may be preferred over features with fewer levels as it is then easier to split the dataset such that the splits only contain pure target feature values.

Many of these issues can be improved upon by using ensemble methods, methods that aggregate several decision trees. This generally comes at the cost of interpretability.

Available algorithms for building a decision tree include ID3, C4.5 and CART. These algorithms typically use different criteria for how to perform splitting, ID3 uses information gain, C4.5 uses gain ratio, while CART uses the gini index.

4.2.1 CART Algorithm

Originally the term Classification And Regression Tree (CART) was introduced by Breiman et al.[9] as an umbrella term used for analysis of regression as well as classification trees. The CART algorithm is the most commonly used algorithm for building decision trees. It is a non-parametric learning technique.

With the CART algorithm trees are constructed using a top-down approach. We start by looking at all the available training data, and selecting the split that minimizes the cost function. This is then the root node. Split is performed in the same way moving down the tree until a stopping criteria is met.

To decide on the best split, a measure of impurity, `G`, is used. For CART this is typically the Gini index, while other options include the information entropy, or the miss-classification rate, see the following sections.

At each node we split the dataset into two subsets a and b using a single feature k and a threshold t_k, by finding the pair (k, t_k) giving the lowest impurity for the subsets according to the chosen impurity measure. This minimizes our cost function for this problem,

$$C(k, t_k) = \frac{m_{\mathrm{a}}}{m} G_{\mathrm{a}} + \frac{m_{\mathrm{b}}}{m} G_{\mathrm{b}},$$

where $G_{\mathrm{a/b}}$ measures the impurity of each of the subsets, and $m_{\mathrm{a/b}}$ is the number of instances in subset a and subset b, respectively.

There are several possible stopping criteria, like maximum depth of tree, all members of the node belonging to the same class, the impurity factor decreasing by less than some threshold for further splits, or that the minimum number of node members is reached.

Building the Tree

When building the tree we start with the root node and move recursively down the tree as indicated in the following pseudocode: [6]

```
def find_split(input_data, target):
    start_impurity = find_impurity(data, target)
    split_threshold, split_feature, split_impurity
    for each feature in input_data:
        for each unique_value in feature:
            threshold = value
            impurity = find_impurity(feature, threshold)
            if impurity is better than split_impurity:
                split_threshold = threshold
                split_impurity = impurity
                split_feature = feature
    split_node(split_feature, split_threshold, input_data, target)
```

4.2.2 Gini Index

The Gini index is also called the Gini impurity, and it measures the probability of a particular variable that is randomly chosen being wrongly classified. As such it takes values between 0 and 1. Another way to look at it is that it measures the lack of 'purity' of the variables. A node is pure if all its variables or members belong to one class. The gini index then takes the value 0. The more of the members of the node that belong to a different class, the more impure the node is.

Denoting the fraction of observations (or members) of node/region m being classified to a particular class k as p_{mk}, the Gini index, g can be defined as

$$g = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}) = 1 - \sum_{k=1}^{K} p_{mk}^2.$$

The fraction p_{mk} can be calculated as

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k).$$

When building a decision tree using CART with the Gini index as impurity measure we choose the attribute/feature with the smallest Gini index as the root node.

4.2.3 Entropy

Entropy is another measure for impurity. It is known from thermodynamics as a measure of disorder. In the classification case the entropy, or information entropy, is a measure for how much information we gain by knowing the value (or classification) of more features.

The entropy, s, can be defined in terms of the fraction p_{mk} defined in the section above, as

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}.$$

4.3 Ensemble Methods

Ensemble methods use a set, or ensemble, of so-called weak learners, and use their combined predictive power to make predictions. While each individual model in the ensemble may have a poor performance, say only just above random guessing, the resulting ensemble may perform very well. Ensemble methods can use any (weak) learner as its base learner, and even a combination of different ones. We will, however look exclusively at ensembles of decision trees.

An individual decision tree is prone to over-fitting and high variance. The idea is that when averaging over many of these weak learners the variance of each tree averages out, reducing the total variance and thereby error of the resulting model. We will be exploring three kinds of ensemble methods where decision trees are the base learner; bagging, boosting, and random forests.

4.3.1 Bagging

Bagging is a simple form of an ensemble method. A set of `N` trees is built from the input data, with the twist that each tree is built only on a subset of the total input data. The subset is chosen by randomly sampling of the data, with substitution. In that way each tree is built on a bootstrap sample of the original training data. This can effectively reduce the variance of the model. This improved performance comes at the expense of the interpretability of the model.

```
Bagging Algorithm:
With training data X and y, and N trees.
for i from 1 to N:
    1. set X_ and y_ equal to a subset of X and Y, drawn from X and y with
       replacement
    2. fit tree i to X_ and y_
    3. store tree i for future prediction
```

4.3.2 Random Forests

Random forests take bagging one step further by adding randomness in what features are available when fitting each tree to the data. In essence, in addition to fitting to a random sample of the input or training data, the fit is done using a random subset of the available features.

Often every tree will be dominated by one or more strong or defining features, with every tree having the same root node. By only looking at a random subset of the features per tree we increase the randomness in the resulting tree ensemble, and hope to further reduce the variance. We are essentially reducing the correlation between the individual trees, and as such expect an improved reduction in variance over bagging where the trees will often remain very correlated.

One typically looks at $m \approx \sqrt{p}$ predictors for each tree, with p being the total number of predictors in the dataset.

```
Random Forest Algorithm:
With training data X and y, with X made up of F features, and
N trees.
for i from 1 to N:
    1. set X_ and y_ equal to a subset of X and Y, drawn with
       replacement
    2. draw f features randomly from the F features
    3. fit tree i to X_[f] and y_
    4. store tree for future prediction
```

4.3.3 Adaptive Boosting

While the ensemble methods described until now can easily be performed in parallel, adaptive boosting, or AdaBoost, uses the resulting prediction from each subsequent fit to improve the fit in the next. After each tree is fit to the data the algorithm makes note of what data is miss-classified in the resulting model, and gives these data increased weight in further model fitting. Optional features include performing a similar weighing of the features used, where features leading to good predictions are given more weight, or discarding individual models if their accuracy is below some level.

```
AdaBoost Algorithm:
With training data X and y, and N trees.
First assign equal weight to each observation
weights = np.ones()*1./X.shape[0]
for i from 1 to N:
    1. fit tree i to the data, weighing the data according to weights
        2. calculate error by summing up the weight of misclassified
           observations
       error = sum(weights of misclassified observation)/sum(weights)
        3. update weights using the quantity
           alpha=log(1 - error)/error:
           weight_i = weight_i*exp(alpha) if incorrectly classified
           weight_i = weight_i*exp(-alpha) if correctly classified
```

4.4 Data Processing

Our datasets in this projects are fairly clean and well-defined already, and do not require much pre-processing. As we are working with decision trees we do not need to normalize our input data, it will not affect the results. As we are using datasets with fairly high dimensionality, we explore some methods of feature selection to see if we can reduce dimensionality without reducing model performance

Variance Threshold

Feature selection by variance threshold is a simple form of feature selection where only features with variance above some threshold are kept. The logic behind this method is that the variance of a features signifies its spread. A feature that is more spread out across the entries will make it easier to separate entries based on this feature. In contrast, imagine if a feature takes only one value. This feature will then provide no information we can use to separate the entries. The strictness of this method is determined by the variance threshold used.

Univariate Feature Selection [8][12]

In univariate feature selection features are selected based on their value according to some set scoring function. While the variance threshold can be used for any problem, we must in univariate feature selection select a scoring function suitable for our needs. We are only looking at classification problems, and our mushroom dataset consists solely of categorical input features. This limits the suitable scoring functions.

We have used the χ^2 test statistic, which measures the dependence between stochastic variables, and can be used for categorical features. The χ^2 test statistic is a statistical hypothesis test and uses the assumption that the observed frequencies for each categorical variable match its expected frequencies. With this scoring function the features that are the most likely to be independent of the class and therefore irrelevant for classification are scored low, and can be removed.

Our fetal health dataset on the other hand has continuous features, meaning the χ^2 test statistic is not suitable. In place of it we have used the ANOVA F-value. The ANOVA (Analysis of variance) f-value uses the f test statistic to evaluate each individual feature's ability to distinguish the classes of the variables.

For both datasets we also use the mutual information. The mutual information measures the dependency between two random variables, with lower variables indicating more independent variables.

4.5 Performance Measures

4.5.1 Accuracy

See Methods section in project 2[13].

4.5.2 Confusion Matrix

A confusion matrix is a useful tool for visualizing the predictive accuracy of a model. Unlike the measure above, the confusion matrix illustrates the predictive accuracy per class. This way we

can see what classes the model performs well of less well for, and information like what class misclassifications en up in instead.

The matrix consists of one row and one column per class. The rows represent the predicted class, while the columns represent the actual or true class (or vice versa). Where the row and column for a class meet we have the number of correctly classified instances. Where rows and columns of different classes meet we have a wrongly classified instance, and we can see trends in misclassifications. See for instance figure 6 for an example.

4.5.3 ROC Curve

A receiver operating characteristic curve, or ROC curve, is another tool for visualizing a model's predictive performance, more specifically a binary model. It consists of a graphical plot of false positive rate vs true positive rate for a set of discriminative thresholds. The area under the curve is one measure of model performance, with a perfect model getting a score of one.

4.6 External Libraries and Code Structure

While We have written my own code for the decision tree code, as well ass the various ensemble methods used, we have used functionality from the library scikit-learn[14] both to compare with our own code, and to increase performance for the ensemble methods. I have also used this library for splitting the data set. This python library is based on numpy and and scipy, and contains a wide array of machine learning algorithms, including tree based methods.

Other packages I've used is numpy[15] for array handling, matplotlib.pyplot[16] for plots and visualizations, and scikitplot[17] for plotting the cunfusion matrices and ROC curves.

I have developed a class `DecisionTree` for creating decision trees. This is located in a file of the same name in the Code directory and aslo includes some relevant helper functions like for printing the tree, calculating accuracy, and comparing with the equivalent functionality in scikit-learn. The file also contains two examples for using the code, using datasets from scikit-learn.

The class`TreeEnsemble` is a base class for ensembles of trees, and the file with the same name in the Code directory contains all the ensemble classes I've made, and again a usage example using datasets from scikit-learn. The Ensemble methods use scikit-learn's `DecisionTreeClassifier()` in place of the `DecisionTree` class simply due to the latter being quite slow.

I have done testing continuously by comparing results to results using the scikit-learn for a number of datasets.

Code available from the following github repository: https://github.com/emiliefj/FYS-STK3155.

5 Results and Discussion

5.1 Mushroom Dataset

5.1.1 Pre-processing the Data

As we know, all the features of this dataset are categorical. We can first explore the data by looking at the unique values for each feature, see table 1.

`Table 1: Unique values of the features in the mushroom data set.`

```
                                                 unique values
class                                                    [p, e]
cap-shape                                   [x, b, s, f, k, c]
cap-surface                                      [s, y, f, g]
cap-color                          [n, y, w, g, e, p, b, u, c, r]
bruises                                                  [t, f]
odor                                  [p, a, l, n, f, c, y, s, m]
gill-attachment                                          [f, a]
gill-spacing                                             [c, w]
gill-size                                                [n, b]
gill-color                  [k, n, g, p, w, h, u, e, b, r, y, o]
stalk-shape                                              [e, t]
stalk-root                                    [e, c, b, r, ?]
stalk-surface-above-ring                         [s, f, k, y]
stalk-surface-below-ring                         [s, f, y, k]
stalk-color-above-ring                [w, g, p, n, b, e, o, c, y]
stalk-color-below-ring                [w, p, g, b, n, e, y, o, c]
veil-type                                                  [p]
veil-color                                       [w, n, o, y]
ring-number                                         [o, t, n]
ring-type                                     [p, e, l, f, n]
spore-print-color                     [k, n, u, h, w, r, o, y, b]
population                                 [s, n, a, v, y, c]
habitat                                 [u, g, m, d, p, w, l]
```

As we can see, there is only one used value for `veil-type`, 'p' or partial. This feature then provides us with no information that we can use to distinguish the different mushrooms, and we remove the feature completely. We also check to make sure none of the entries are missing values, see table 2.

Table 2: The number of missing values for each features. We see that no values are missing.

[90]:

class	0
cap-shape	0
cap-surface	0
cap-color	0
bruises	0
odor	0
gill-attachment	0
gill-spacing	0
gill-size	0
gill-color	0
stalk-shape	0
stalk-root	0
stalk-surface-above-ring	0
stalk-surface-below-ring	0

```
stalk-color-above-ring      0
stalk-color-below-ring      0
veil-color                  0
ring-number                 0
ring-type                   0
spore-print-color           0
population                  0
habitat                     0
dtype: int64
```

Another point of interest is whether the data are fairly evenly divided among the two categories. We check this by comparing the number of entries belonging to each class as shown in figure 1.

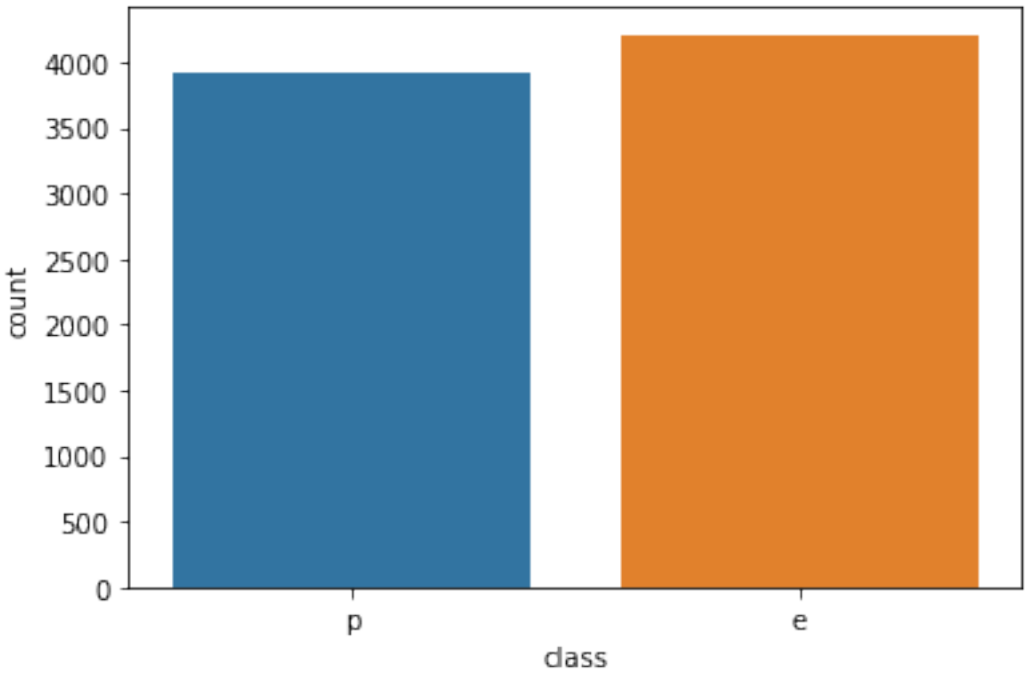
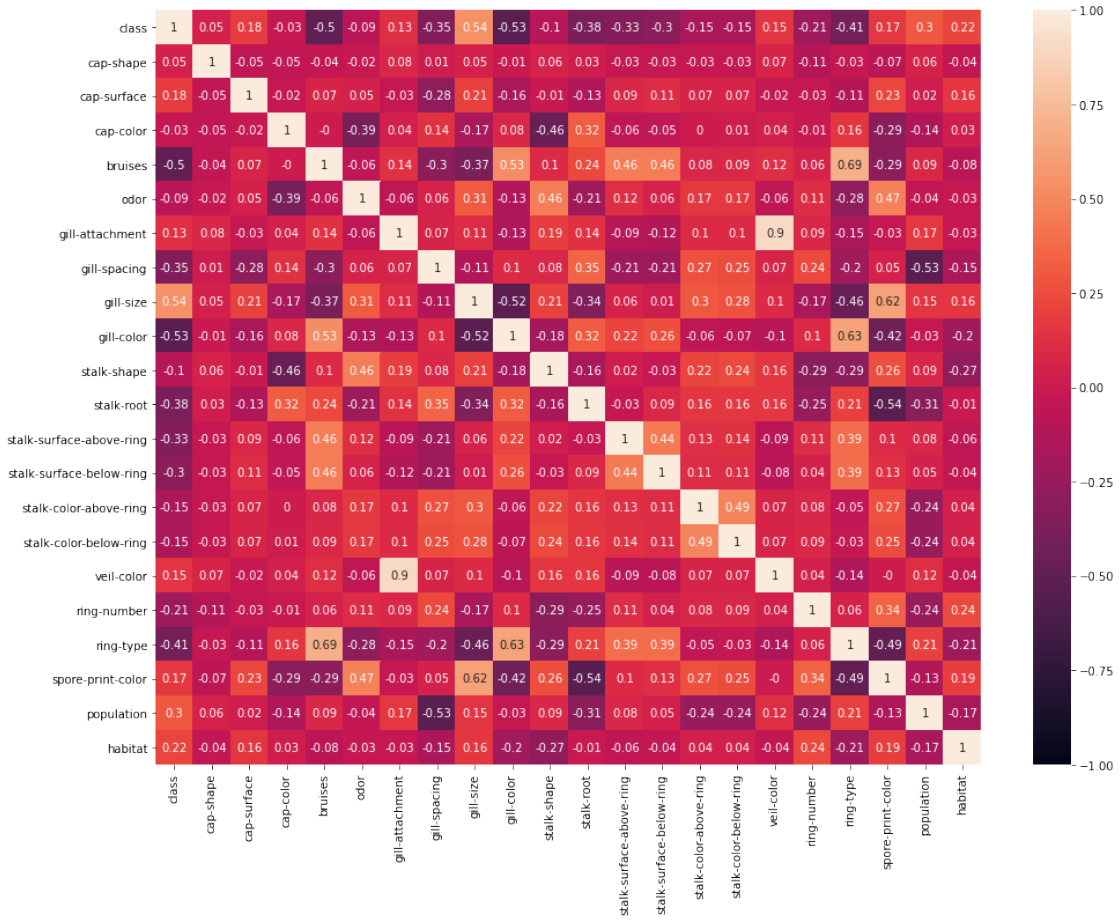


Figure 1: Histogram of class distribution. We can see the dataset contains a fairly even split between the two classes.

Figure 2 shows a heatmap of the correlations between the features.

Figure 2: A heatmap showing the correlation matrix for the features in the mushroom dataset.



We see that `class` is most highly correlated with `gill-size`, `gill-color`, and `bruises`, followed by `ring-type`, `stalk-root`, and `gill-spacing`. The most highly corrolated features (meaning they provide much of the same information) are `gill-attachment` and `ring-number`, with a correlation of 0.9. Other fairly correlated features are `bruises` and `ring-type`, `gill-color` and `ring-type`, and `gill-size` and `spore-print-color`.

To prepare for model selection and model comparisons we have split the data into three parts, a training set made up of 60% of the dataset, and a validation and test set, each making up 20% of the total set.

5.1.2 Feature Selection

As we have a fairly high-dimensional problem we performed feature selection in order to reduce the number of features. The goal for this was both to increase speed of fitting a model to the data, and to hopefully end up with a simpler model that is easier to interpret, understand, and use.

We have tried three methods for this; a simple variance threshold where features with a variance below a certain threshold or cutoff value are excluded, as well as univariate feature selection using two different scoring functions.

We use a cutoff value of 0.8 as variance threshold, which results in excluding four of the features,

namely `gill-attachment`, `gill-spacing`, `veil-color`, and `ring-number`. The number of features is then reduced from 21 to 17.

We can compare this to univariate feature selection. The result when using chi-squared statistic, can be seen in figure 3. We find from this result that comparing the features using this statistic the key features to include are features 8, 17, 7, 3, 10, 20, and 6, corresponding to `gill-color`, `ring-type`, `gill-size`, `bruises`, `stalk-root`, `habitat`, and `gill-spacing`.

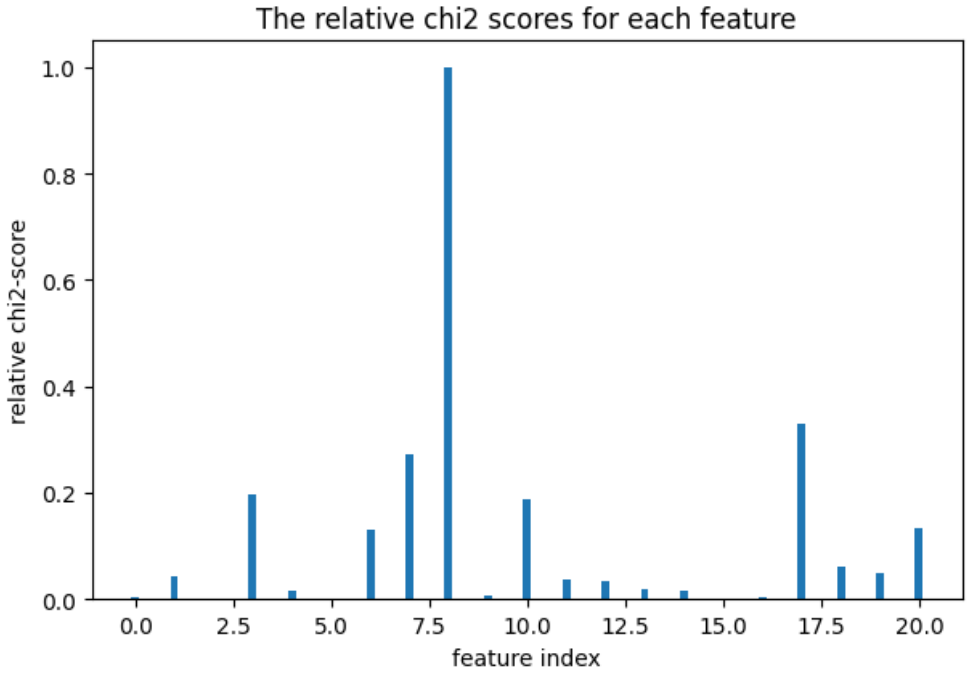


Figure 3: Bar plot showing the chi-squared scores of the the features in the mushroom dataset. We see that feature eight has the clear highest score. This corresponds to gill-color.

Using instead the mutual information as scoring function we find that more features are included, Most notably feature four, `odor`, has gone from not being included to being the main feature. A bar plot is shown in figure 4. This result correlates more closely with our expectations as we know from the known rules described in the Data section that using only `odor` is enough to get a 98.52% accuracy on this dataset.

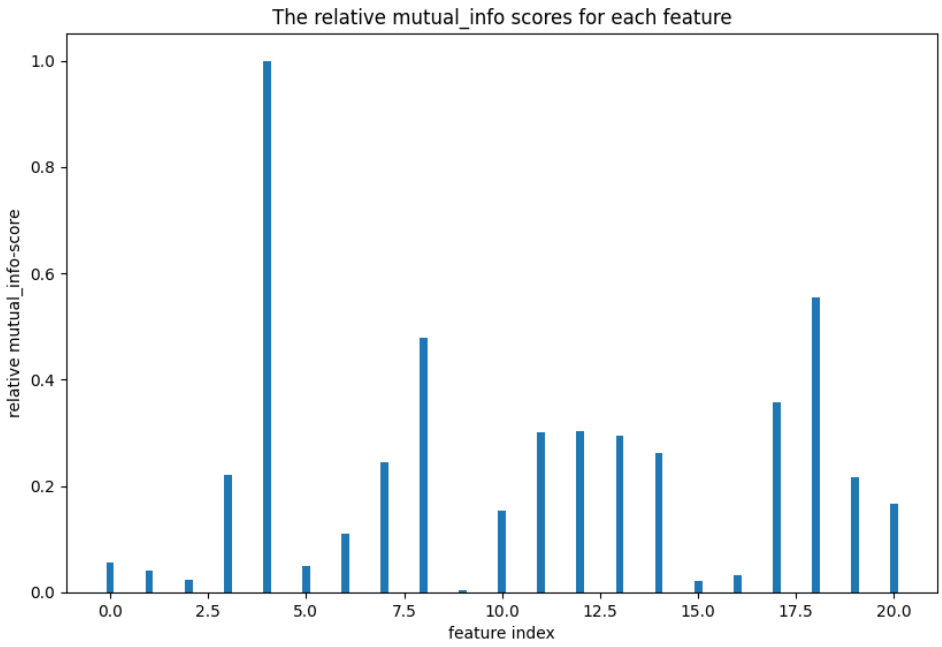


Figure 4: Bar plot showing the mutual information scores of the the features in the mushroom dataset. We see that feature four, `odor` scores highest.

5.1.3 Model Comparisons

We have fit several models to our mushroom dataset. As a baseline I have used logistic regression to classify the data. Using cross validation I determined the optimal learning rate on this data to be 1.0, and using 100 epochs and a batchsize of 50 the resulting accuracy is 0.958 on the validation dataset.

We know from the four simple rules in the description of the dataset from the Data section, that one should not need a very complex model to achieve a fairly high precision. Table 3 shows the accuracy scores for a single 'stub', a tree with just a root node and two leaves. My DecisionTree chooses to split the root node using the `odor`-feature, for all trees except when using chi2 as scoring function, which uses `ring-type`. Alternatively `scikit-learn` splits using `gill-color`. An example of a tree is shown in figure 5, where we see the tree built with our own decision tree code using a max depth of five. Table 4 and 5 show accuracy results for trees with a max depth of two and five, respectively. We see from these results that even for very small trees we get a fairly high accuracy, and already at a depth of five we are nearing a perfect score, while we with a depth of two have a performance equivalent to that of logistic regression. Somewhat surprisingly over-fitting, the bane of decision trees, does not seem to be an issue.

We see that our feature selection has not proved particularly useful. As expected feature selection with mutual information provides a better end result than the χ^2 test statistic, but neither are better than using all features, and considering we need such a small tree for a high accuracy, we have no need for the performance boost.

`Table 3: The accuracy score for a single decision tree using various`

configurations of the input features. All trees have a maximum depth of 1, and two leaf nodes, also known as a 'stub'.

Model	Training	Validation
Single decision tree, own code	0.887	0.892
Single decision tree, scikit-learn	0.793	0.785
Single decision tree, 17 features selected with variance threshold	0.887	0.892
Single decision tree, 7 features selected with chi2	0.773	0.775
Single decision tree, 14 features selected with mutual information	0.887	0.892

Table 4: The accuracy score for a single decision tree using various configurations of the input features. All trees have a maximum depth of 2, and max four leaf nodes.

Model	Training	Validation
Single decision tree, own code	0.956	0.944
Single decision tree, scikit-learn	0.911	0.913
Single decision tree, 17 features selected with variance threshold	0.956	0.944
Single decision tree, 7 features selected with chi2	0.903	0.904
Single decision tree, 14 features selected with mutual information	0.956	0.944

Table 5: The accuracy score for a single decision tree using various configurations of the input features. All trees have a maximum depth of 5, and max 15 leaf nodes.

Model	Training	Validation
Single decision tree, own code	0.9996	0.9994
Single decision tree, scikit-learn	0.9803	0.9797
Single decision tree, 17 features selected with variance threshold	0.9996	0.9993
Single decision tree, 7 features selected with chi2	0.9602	0.9674
Single decision tree, 14 features selected with mutual information	1.0	1.0

```
The tree using my own code:
|--- odor == 5
|   |--- spore-print-color == 5
|   |   |--- class: 1,    prediction: ['0.0', '1.0']
|   |--- spore-print-color !=  5
|   |   |--- stalk-surface-below-ring == 3
|   |   |   |--- ring-type == 0
|   |   |   |   |--- class: 1,    prediction: ['0.0', '1.0']
|   |   |   |--- ring-type !=  0
|   |   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |   |--- stalk-surface-below-ring !=  3
|   |   |   |--- cap-surface == 1
|   |   |   |   |--- class: 1,    prediction: ['0.0', '1.0']
|   |   |   |--- cap-surface !=  1
|   |   |   |   |--- gill-size == 0
|   |   |   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |   |   |   |--- gill-size !=  0
|   |   |   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|--- odor !=  5
|   |--- stalk-root == 2
|   |   |--- odor == 4
|   |   |   |--- class: 1,    prediction: ['0.0', '1.0']
|   |   |--- odor !=  4
|   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |--- stalk-root !=  2
|   |   |--- stalk-root == 4
|   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |   |--- stalk-root !=  4
|   |   |   |--- odor == 3
|   |   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |   |   |--- odor !=  3
|   |   |   |   |--- odor == 0
|   |   |   |   |   |--- class: 0,    prediction: ['1.0', '0.0']
|   |   |   |   |--- odor !=  0
|   |   |   |   |   |--- class: 1,    prediction: ['0.0', '1.0']
```

Figure 5: An example of a decision tree created with our own code. The 'prediction' list gives the probabilities per class for that leaf node according to the training data. We see that the features used in creating the tree are tightly connected to those used in the four simple rules defined for the dataset.

In figure 6 and 7 we see a comparison of the confusion matrices for a stub and a larger decision tree. We see that the stub mostly misclassifies class 0, the edible class. This is good news as wrongly classifying a poisonous mushroom as edible will potentially have much more dire consequences than misclassifying an edible mushroom as poisonous.

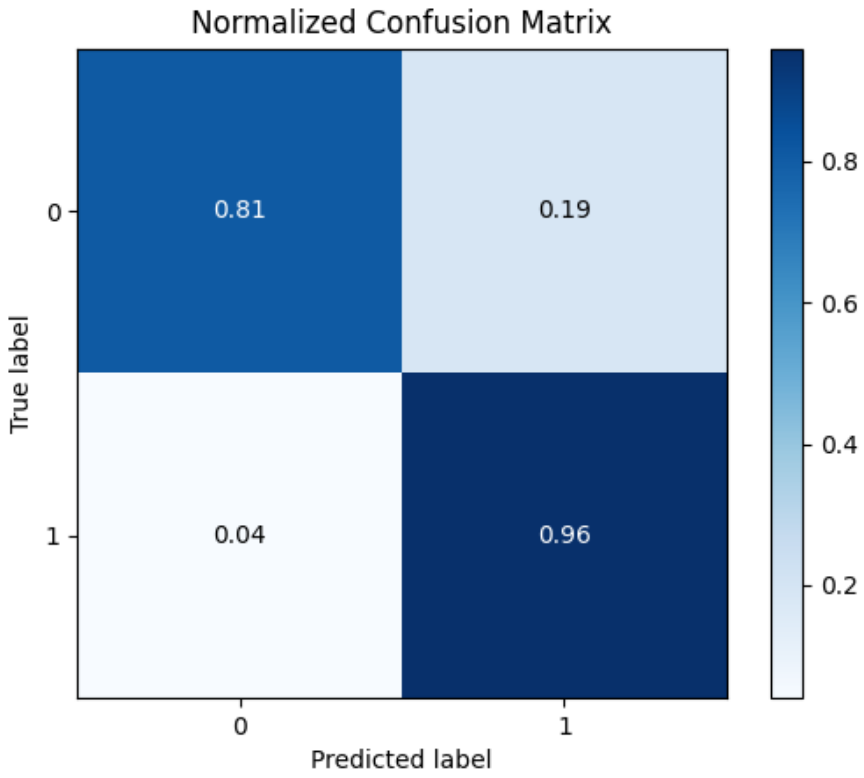


Figure 6: The confusion matrix for a single decision tree with a depth of one.

.

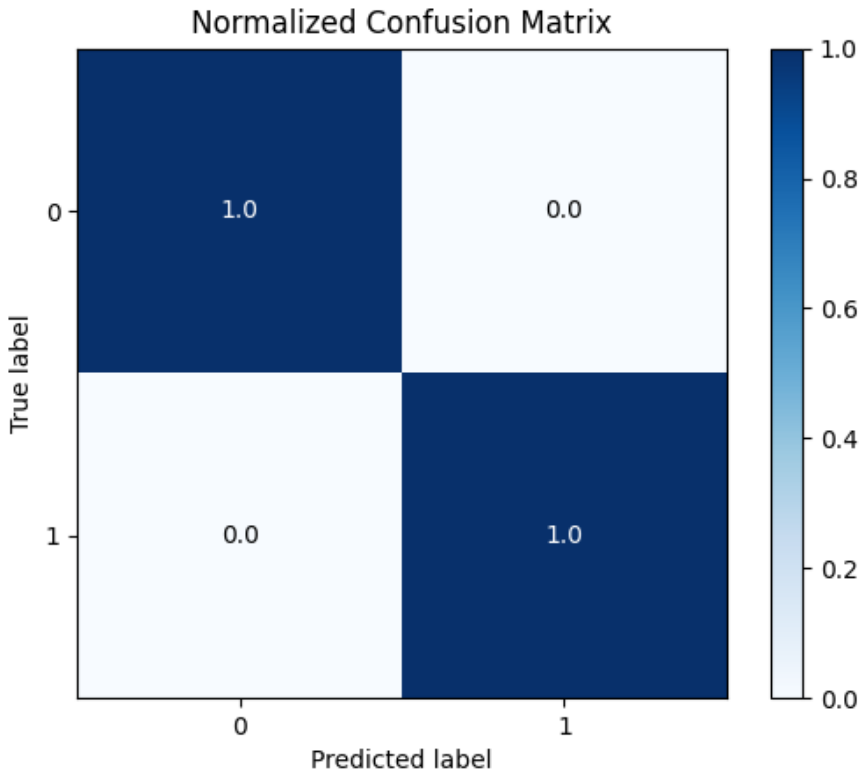


Figure 7: The confusion matrix for a single decision tree with a depth of five.

In tables 6 and 7 we look at how our ensemble methods fare. We see that bagging and random forests give no improvement to our accuracy score. This is unsurprising, as our tree did not seemed plagued by high variance. We see that using adaptive boosting gives the best accuracy, reaching an accuracy of 100% using trees with a max depth of only two.

Table 6: The accuracy score for ensemble methods. All trees have a maximum depth of one, and two leaf nodes. All methods use 100 trees to build the ensemble. In random forest five features are picked at random for each tree.

Model	Training	Validation
Bagging	0.793	0.784
Random forest	0.879	0.884
Adaptive boosting	0.985	0.983

Table 7: The accuracy score for ensemble methods. All trees have a maximum depth of two, and four leaf nodes. All methods use 100 trees to build the ensemble. In random forest five features are picked at random for each tree. The confusion matrix here is not very interesting as it is identical to that in figure 7. Figure 8 shows the confusion matrix for adaptive boosting using only stubs. We see once again that only edible mushrooms are misclassified, and very

rarely.

Model	Training	Validation
Bagging	0.911	0.913
Random forest	0.934	0.938
Adaptive boosting	1.0	1.0

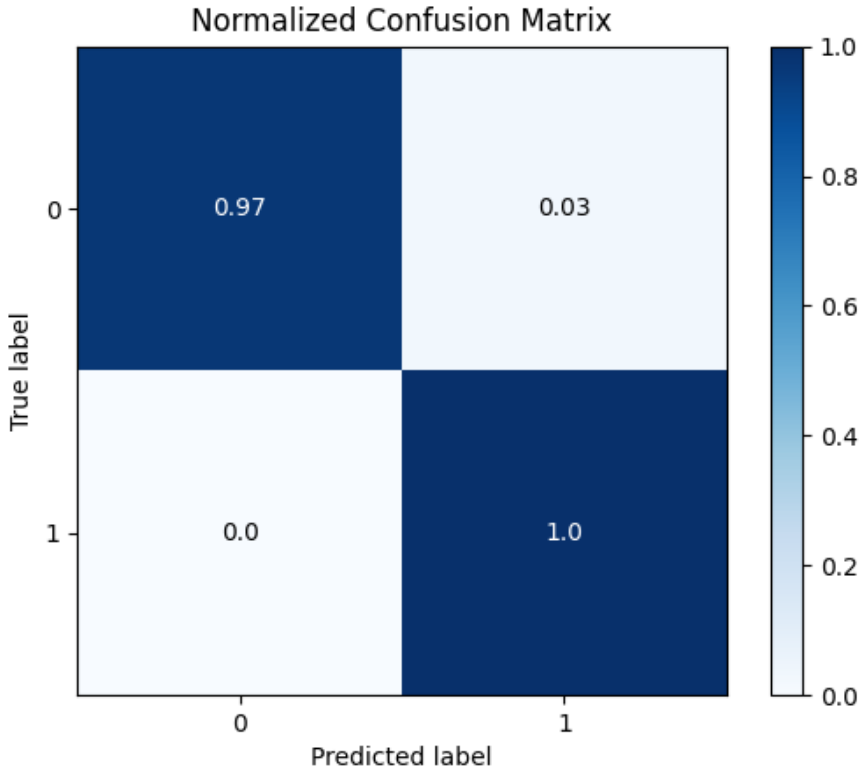


Figure 8: The confusion matrix for adaptive boosting with 100 stubs.

5.2 Fetal Health

5.2.1 Pre-processing the Data

The fetal health dataset consists of 22 input features, all of which are numerical, and mostly continuous. Table 8 shows descriptive statistics for the 22 features, and the target `fetal_health`. We know that the target takes one of three classes, `normal`, `suspect`, or `pathological`, yet we see that the 75% percentile has the value 1. This indicates that our dataset is quite skewed. Figure 9 shows the distribution of entries between the three classes. We see that more than three quarters of the data belong to the first class. We could attempt to balance out the dataset and create one with a more equal distribution between the classes, vut we have decided not to in order to explore how such a skewed dataset affects prediction accuracy for our models.

Table 8: A table showing key statistical values for the features in the fetal␣
↪health dataset.

	count	mean
baseline value	2126.0	133.303857
accelerations	2126.0	0.003178
fetal_movement	2126.0	0.009481
uterine_contractions	2126.0	0.004366
light_decelerations	2126.0	0.001889
severe_decelerations	2126.0	0.000003
prolongued_decelerations	2126.0	0.000159
abnormal_short_term_variability	2126.0	46.990122
mean_value_of_short_term_variability	2126.0	1.332785
percentage_of_time_with_abnormal_long_term_vari...	2126.0	9.846660
mean_value_of_long_term_variability	2126.0	8.187629
histogram_width	2126.0	70.445908
histogram_min	2126.0	93.579492
histogram_max	2126.0	164.025400
histogram_number_of_peaks	2126.0	4.068203
histogram_number_of_zeroes	2126.0	0.323612
histogram_mode	2126.0	137.452023
histogram_mean	2126.0	134.610536
histogram_median	2126.0	138.090310
histogram_variance	2126.0	18.808090
histogram_tendency	2126.0	0.320320
fetal_health	2126.0	1.304327

	std	min	25%
baseline value	9.840844	106.0	126.000
accelerations	0.003866	0.0	0.000
fetal_movement	0.046666	0.0	0.000
uterine_contractions	0.002946	0.0	0.002
light_decelerations	0.002960	0.0	0.000
severe_decelerations	0.000057	0.0	0.000
prolongued_decelerations	0.000590	0.0	0.000
abnormal_short_term_variability	17.192814	12.0	32.000
mean_value_of_short_term_variability	0.883241	0.2	0.700
percentage_of_time_with_abnormal_long_term_vari...	18.396880	0.0	0.000
mean_value_of_long_term_variability	5.628247	0.0	4.600
histogram_width	38.955693	3.0	37.000
histogram_min	29.560212	50.0	67.000
histogram_max	17.944183	122.0	152.000
histogram_number_of_peaks	2.949386	0.0	2.000
histogram_number_of_zeroes	0.706059	0.0	0.000
histogram_mode	16.381289	60.0	129.000
histogram_mean	15.593596	73.0	125.000
histogram_median	14.466589	77.0	129.000
histogram_variance	28.977636	0.0	2.000
histogram_tendency	0.610829	-1.0	0.000
fetal_health	0.614377	1.0	1.000

	50%	75%	max
baseline value	133.000	140.000	160.000
accelerations	0.002	0.006	0.019
fetal_movement	0.000	0.003	0.481
uterine_contractions	0.004	0.007	0.015
light_decelerations	0.000	0.003	0.015
severe_decelerations	0.000	0.000	0.001
prolongued_decelerations	0.000	0.000	0.005
abnormal_short_term_variability	49.000	61.000	87.000
mean_value_of_short_term_variability	1.200	1.700	7.000
percentage_of_time_with_abnormal_long_term_vari...	0.000	11.000	91.000
mean_value_of_long_term_variability	7.400	10.800	50.700
histogram_width	67.500	100.000	180.000
histogram_min	93.000	120.000	159.000
histogram_max	162.000	174.000	238.000
histogram_number_of_peaks	3.000	6.000	18.000
histogram_number_of_zeroes	0.000	0.000	10.000
histogram_mode	139.000	148.000	187.000
histogram_mean	136.000	145.000	182.000
histogram_median	139.000	148.000	186.000
histogram_variance	7.000	24.000	269.000
histogram_tendency	0.000	1.000	1.000
fetal_health	1.000	1.000	3.000

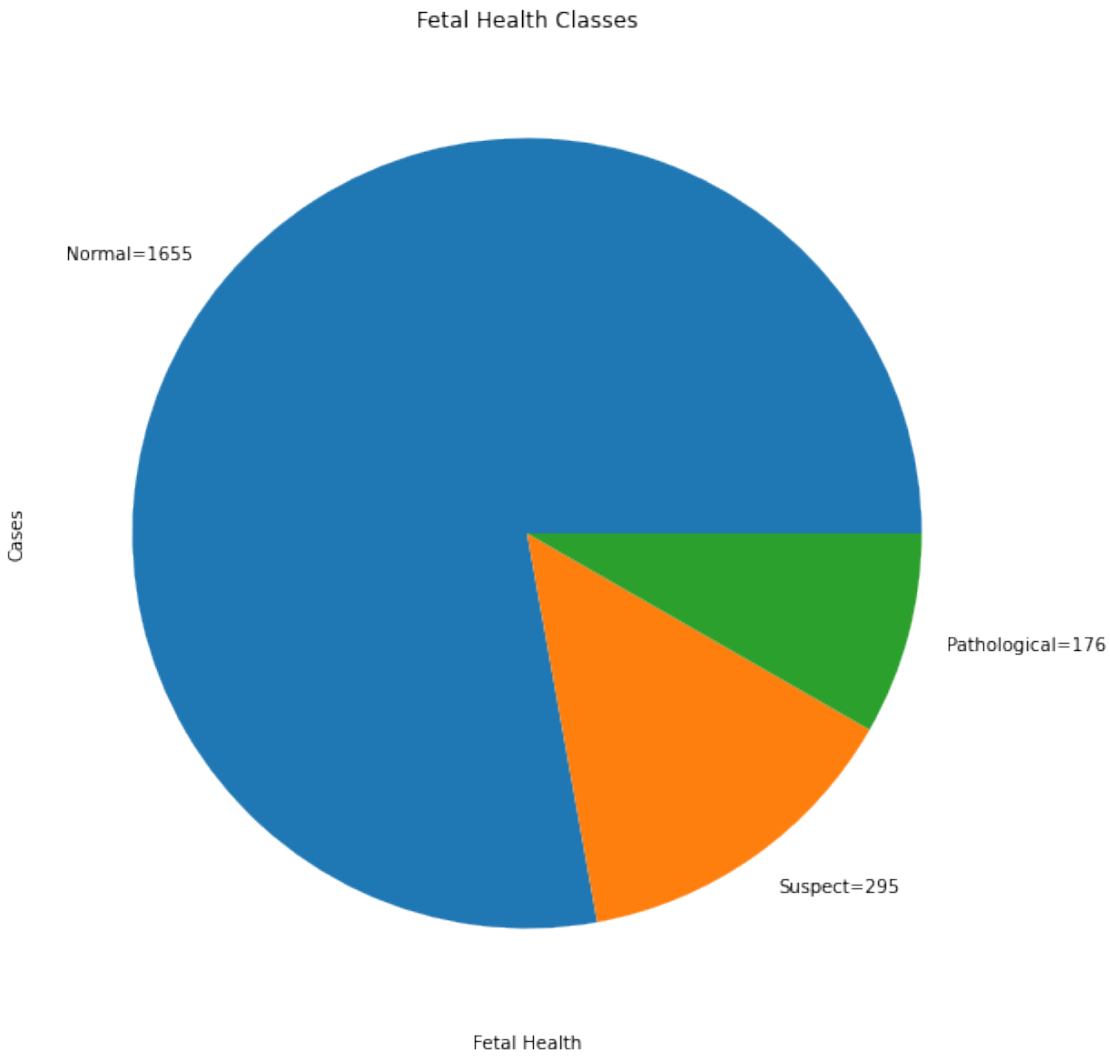


Figure 9: A pie plot showing how the target variable fetal health is distributed among the three possible classes.

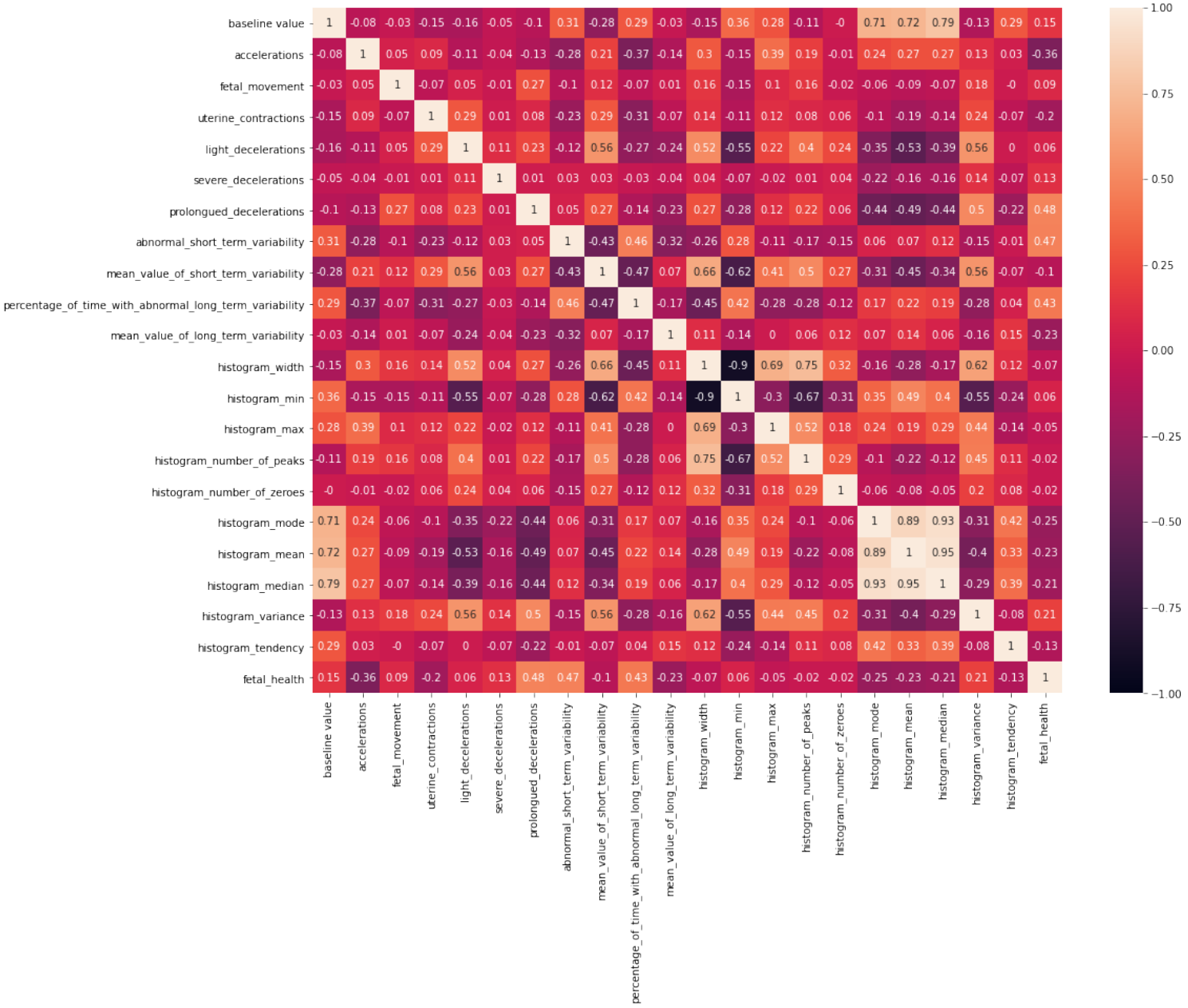


Figure 10: A heatmap showing the correlation matrix for the features in the fetal health dataset.

Table 9: Focusing on the correlations between the features and the target, fetal health. We see that $prolongued_decelerations$, $abnormal_short_term_variability$, and $percentage_of_time_with_abnormal_long_term_variability$ are most closely

correlated with the resulting fetal health classification. Note that the absolute values of the correlations are shown, in descending order.

	fetal_health
fetal_health	1.000000
prolongued_decelerations	0.480000
abnormal_short_term_variability	0.470000
percentage_of_time_with_abnormal_long_term_variability	0.430000
accelerations	0.360000
histogram_mode	0.250000
mean_value_of_long_term_variability	0.230000
histogram_mean	0.230000
histogram_median	0.210000
histogram_variance	0.210000
uterine_contractions	0.200000
baseline value	0.150000
histogram_tendency	0.130000
severe_decelerations	0.130000
mean_value_of_short_term_variability	0.100000
fetal_movement	0.090000
histogram_width	0.070000
histogram_min	0.060000
light_decelerations	0.060000
histogram_max	0.050000
histogram_number_of_peaks	0.020000
histogram_number_of_zeroes	0.020000

Figure 10 shows the correlation matrix for this dataset, while table 9 makes the correlation of the features to the target more clear. We see that `prolongued_decelerations`, `abnormal_short_term_variability`, and `percentage_of_time_with_abnormal_long_term_variability` are the features that correlate most strongly with the classification. Between the features we see that there is naturally a very

strong correlation between histogram_mode, histogram_mean, and histogram_median. as well as between histogram_width, histogram_min, and histogram_max. Particularly for the mode, mean and median we see that they correlate in a very similar manner to all the other features as well and as such contribute with mush the same information, which means we do not gain much, if any, more insight by including all three in our model.

To prepare for model selection and model comparisons we have split the data into three parts, a training set made up of 60% of the dataset, and a validation and test set, each making up 20% of the total set.

5.2.2 Feature Selection

We have performed feature selection for this dataset as well. As we do not have categorical features we have used ANOVA f-test in place of the chi-squared statistic when performing univariate feature selection. For the variance threshold variable selection we have again used 0.8 as the cutoff value.

Variance threshold variable selection leads to selecting 15 out of the 22 features, while we have chosen the 9 and 16 highest scoring features with ANOVA f-test and mutual information, respectively. All methods have in common the features baseline value, abnormal_short_term_variability, percentage_of_time_with_abnormal_long_term_variability, histogram_mode, histogram_mean, histogram_median, and histogram_variance',

with variance threshold including also mean_value_of_short_term_variability, mean_value_of_long_term_variability, histogram_min, histogram_max, histogram_number_of_peaks, histogram_number_of_zeroes, histogram_tendency , and histogram_width.

Using ANOVA f-test we select in addition to the common features, the two features prolongued_decelerations and accelerations, giving a total of nine. A bar plot of the scoring is seen in figure 11.

Scoring the features using mutual information we select the top 16 features, which are the common seven plus the following nine: mean_value_of_short_term_variability, histogram_width, accelerations, histogram_min, prolongued_decelerations, mean_value_of_long_term_variability, uterine_contractions, histogram_max, and light_decelerations. A bar plot of the scoring is seen in figure 12.

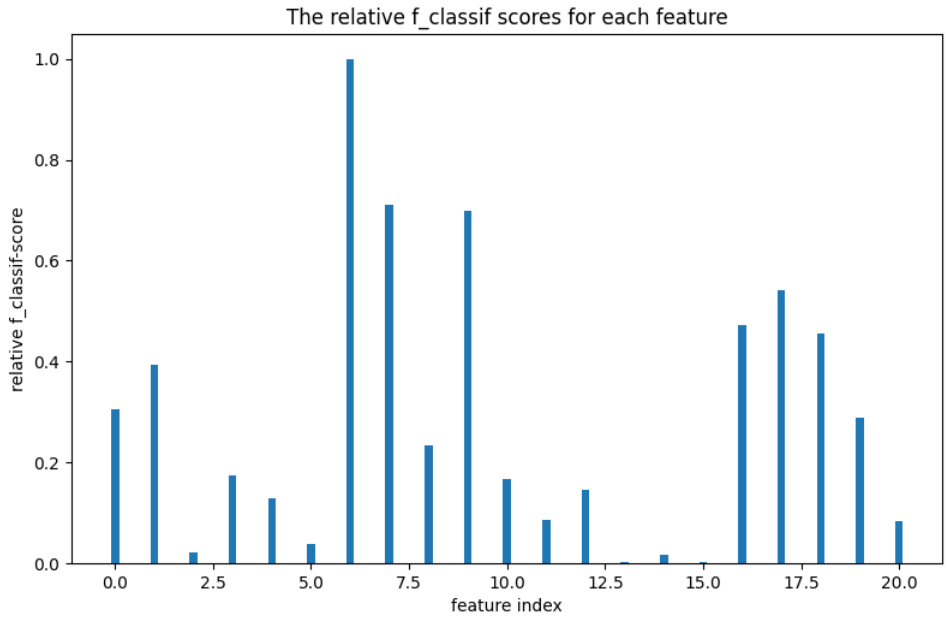


Figure 11: Bar plot showing the f-test scores of the the features in the fetal health dataset. We see that feature six has the clear highest score. This corresponds to prolongued_decelerations.

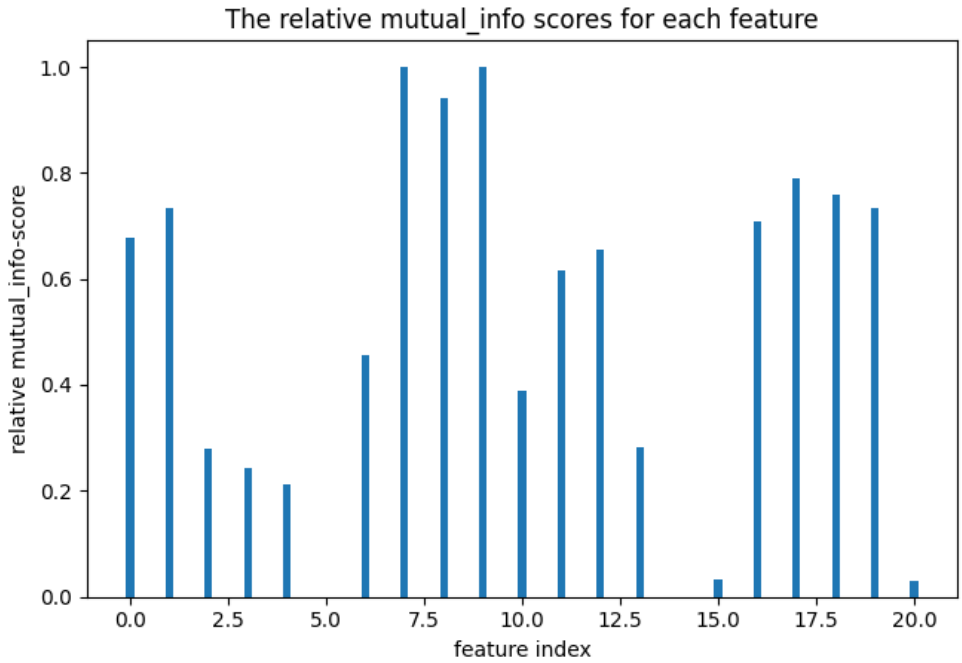


Figure 12: Bar plot showing the mutual information scores of the the features

in the fetal health dataset. We see that features seven, eight and nine have the highest scores. They correspond to abnormal_short_term_variability, mean_value_of_short_term_variability and percentage_of_time_with_abnormal_long_term_variability, so they all describe the variability.

5.2.3 Model Comparisons

As a baseline I have again used logistic regression on the data. Using cross validation I determined the optimal learning rate on this data to be 0.001, and using 100 epochs and a batchsize of 50 the resulting accuracy is 0.849 on the validation dataset.

We begin by comparing model performance using a single tree with the features as selected above. Using a stub, a tree with depth one, all models create the same tree, as shown in figure 13. The resulting accuracy is then 0.778 both on the training and validation sets.

Using a slightly larger tree, with a depth of two, we get the results as seen in table 10. All models seem to have fairly equal performance. This is unsurprising as they use the same root node like for the stubs, and as such do not vary much between them. We saw that all three feature selection methods ended with very many features in common, which is to say they agree fairly well on the most important features.

```
The tree using my own code:
|--- abnormal_short_term_variability <= 59.00
|   |--- class: 0,    prediction: ['0.9', '0.1', '0.0']
|--- abnormal_short_term_variability >  59.00
|   |--- class: 0,    prediction: ['0.4', '0.3', '0.2']
```

Figure 13: The fetal health stub.

Table 10: The accuracy score for a single decision tree using various configurations of the input features. All trees have a maximum depth of two, and four leaf nodes.

Model	**Training**	**Validation**
Single decision tree, own code	0.827	0.845
Single decision tree, scikit-learn	0.851	0.866
Single decision tree, 15 features selected with variance threshold	0.843	0.832
Single decision tree, 9 features selected with f-test	0.827	0.807
Single decision tree, 16 features selected with mutual information	0.827	0.807

It is tempting to say that the accuracy we get is good, especially considering how small of a tree we are using. Looking at the confusion matrix in figure 14, however, we see that the total accuracy is deceiving. Our accuracy measure is strongly influenced by the accuracy in determining class 1 (0 in the figure), or `normal` which dominates the dataset. While performance for the `suspect` class is adequate, the model is quite useless at classifying entries to the `pathelogical` class.

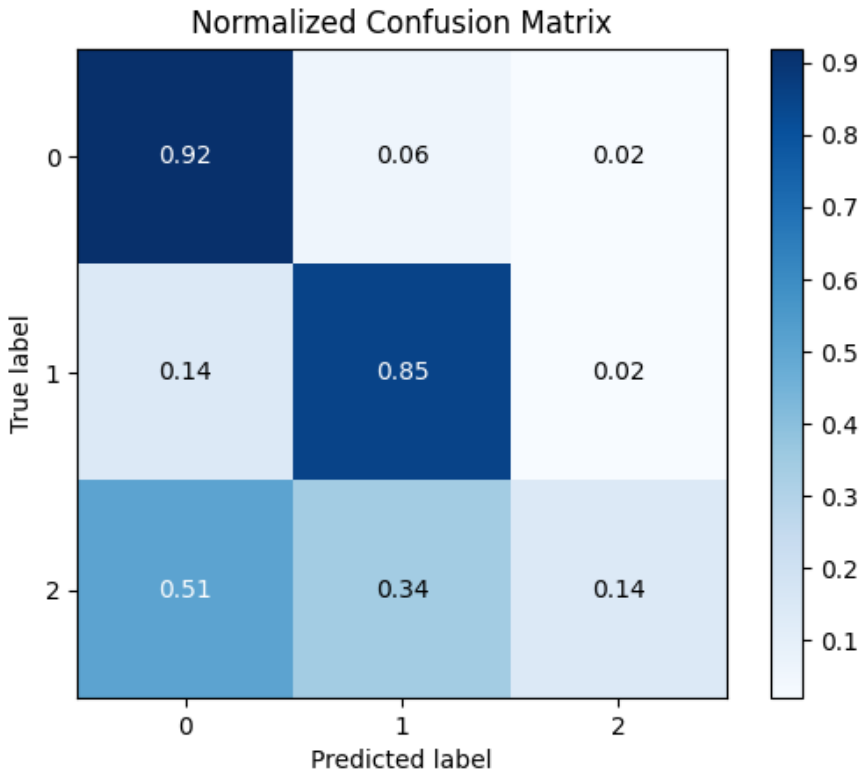


Figure 14: The confusion matrix for a single decision tree on full dataset with a depth of two.

We move on to explore our ensemble methods, to see if they fare better. Table 11 shows the resulting accuracy measures when our ensembles are made up of trees with depth two. Unlike for our mushroom dataset bagging now seems to come out on top, while adaptive boosting is more disappointing. The confusion matrices shown in figures 15 and 16 support this assessment. We see that our adaptive boosting ensemble is classifying almost all entries to the `normal` class. Bagging, while it has lost some accuracy on the middle class as compared to a single tree, performs much better for the `pathelogical` class.

Table 11: The accuracy score for ensemble methods. All trees have a maximum depth of two, and four leaf nodes. All methods use 100 trees to build the ensemble. In random forest five features are picked at random for each tree.

Model	**Training**	**Validation**
Bagging	0.887	0.882
Random forest	0.845	0.847
Adaptive boosting	0.827	0.807

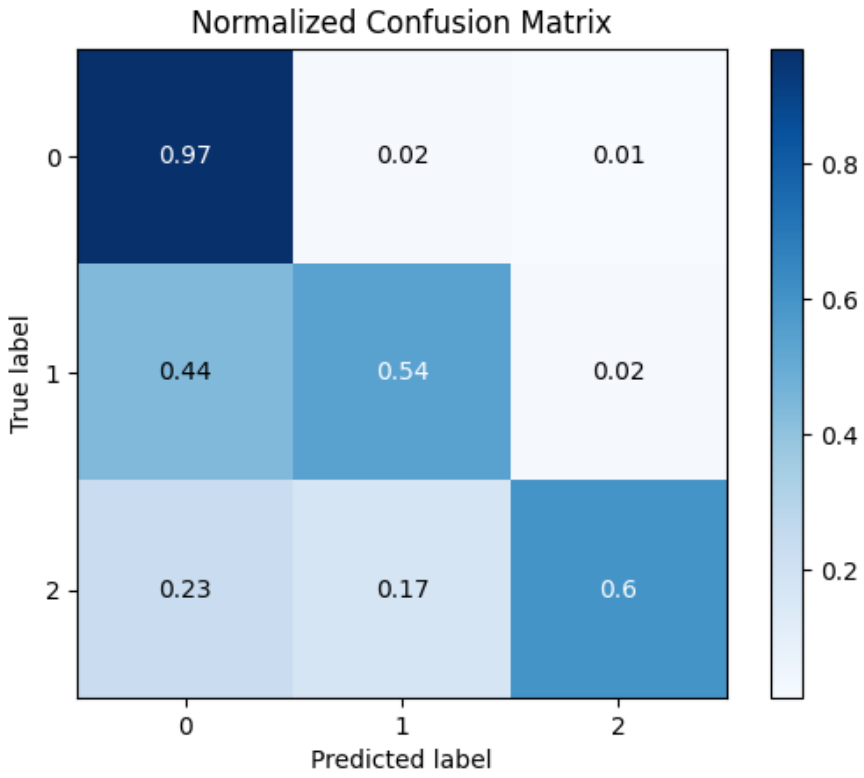


Figure 15: The confusion matrix for bagging 100 decision trees with a depth of two.

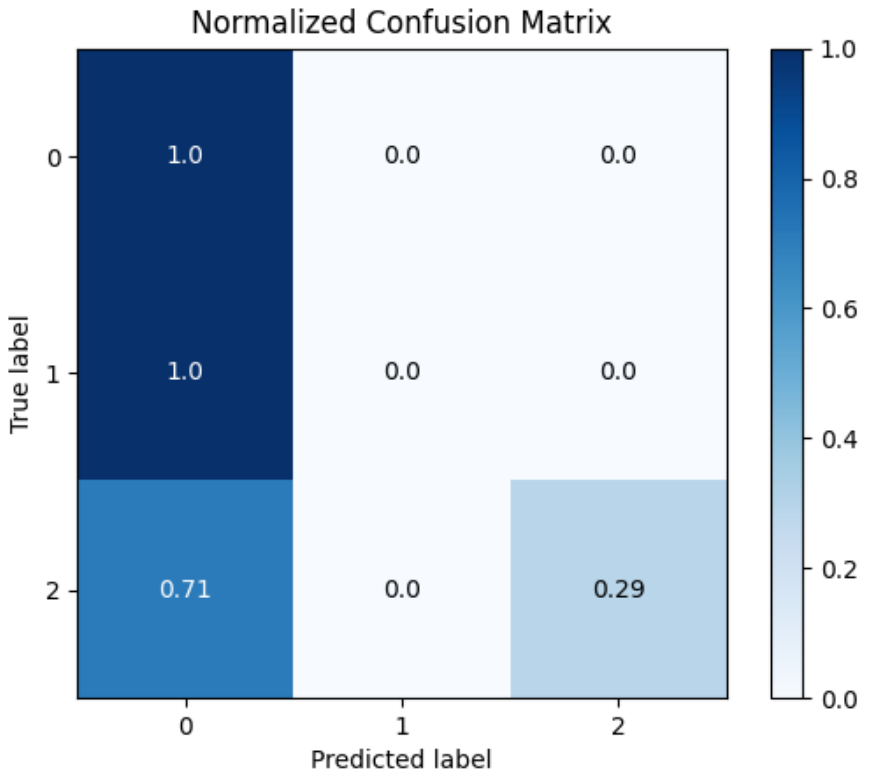


Figure 16: The confusion matrix for adaptive boosting with 100 trees, all with a depth of two.

To achieve a more even performance across the class labels we need to use larger trees. Figures 17, 18, and 19 show the confusion matrices and ROC curves for a single tree, a bagging ensemble, and an adaptive boosting ensemble, respectively. We see that for all three models the accuracy is more even across the class labels, and that once again adaptive boosting comes out as the winner in terms of best performance. Performance is still best for the first class, with the two arguably most important classes to identify lagging behind. Balancing the dataset may improve this situation.

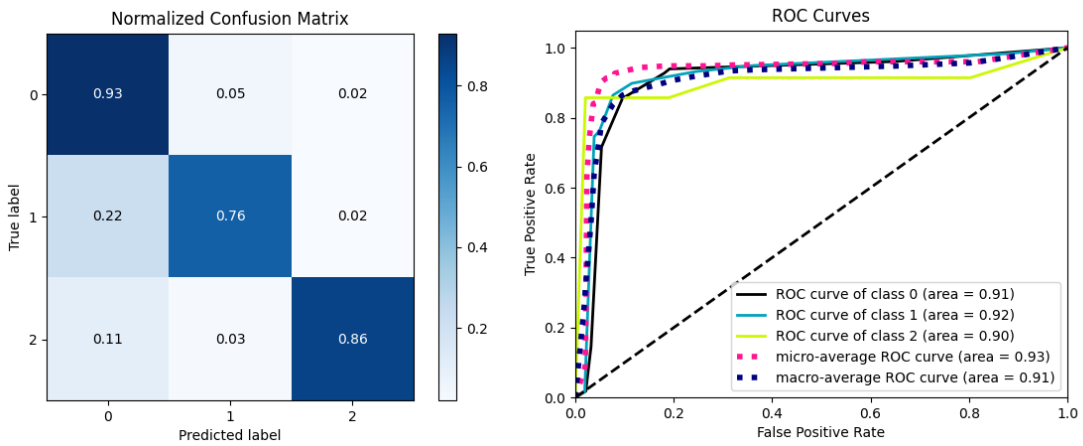


Figure 17: The confusion matrix and ROC curves for a single tree

of depth five, on the test set.

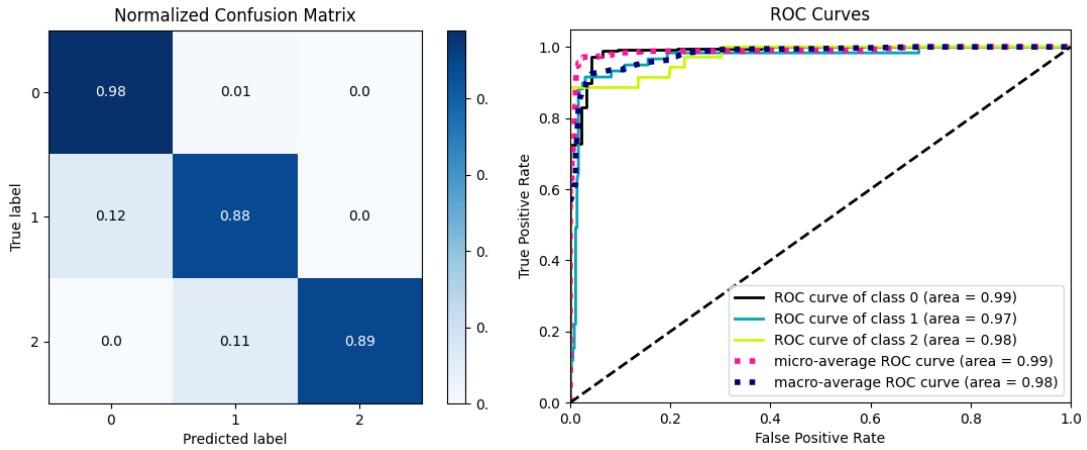


Figure 18: The confusion matrix and ROC curves for an AdaBoost ensemble with 100 trees of depth five, on the test set.

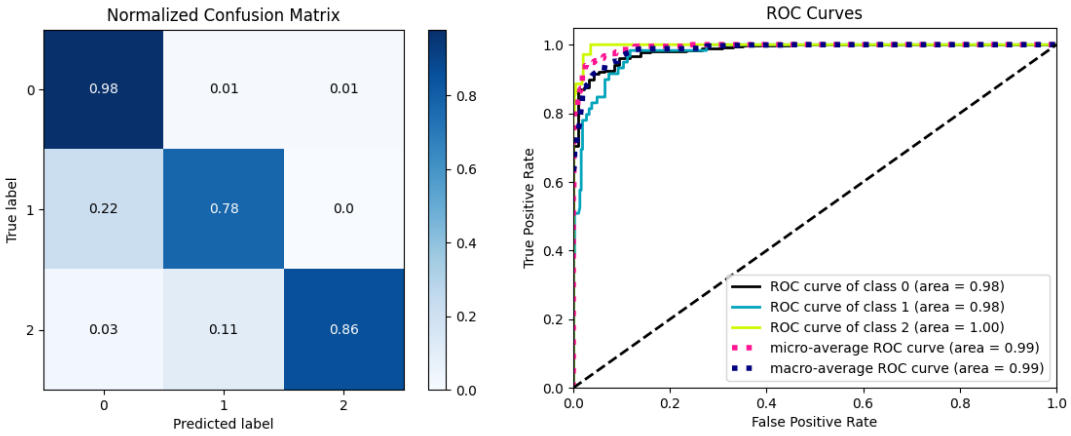


Figure 19: The confusion matrix and ROC curves for a bagging ensemble with 100 trees of depth five, on the test set.

The final scores for the key models when tested on the test set, is shown in table 12. As we saw from the confusion matrices, adaptive boosting gives the optimal accuracy.

Table 12: The accuracy score of our models on the test set. All trees have a maximum depth of five. The ensemble methods use 100 trees to build the ensemble..

Model	**Training**
Singel tree	0.899
Bagging	0.944
Adaptive boosting	0.962

6 Conclusion

In this project we have looked at the relatively simple machine learning model that is the decision tree, and we have added complexity by creating ensemble methods from many such trees. What we have seen is that for some datasets, a single decision tree may be perfectly adequate to give a very good performance. We have also seen that where a single tree fails to give the desired performance, adding several together in an ensemble may be the solution.

As we looked at datasets with a fairly high number of dimensions we looked at feature selection to reduce the problem. This did not give us improvements in accuracy, but depending on method used for selection the accuracy was equivalent to that of the full method. This can provide increased efficiency without loss of performance, where speed is an issue. However we say no great cost of calculation using the full model.

While decision trees generally suffer from high variance, we have nod been plagues by this issue much for our chosen datasets. This is in part because we have looked at smaller trees, and the variance problem will likely increase the bigger you grow your trees, as a larger tree will be more closely adapted to the specifics of the training data.

Because of this missing variance issue, for lack of a better term, our standard ensemble methods added a limited boost to performance. We saw a better increase for the fetal health dataset, which makes sense as there was more to gain, and likely slightly larger variance issues.

For both our datasets we reached a performance comparable to that of logistic regression even when using just a single tree, and we found that the best performance was found for adaptive boosting, where wrongly classified data are given more weight in future tree fittings.

A particular issue we encountered in the fetal health dataset was the importance of not looking solely at the accuracy measure to determine the model performance, especially for unbalanced datasets. A future improvement may be to try balancing the dataset before fitting the models to it.

We would recommend giving decision trees a fair chance when selecting a machine learning model for your problems, particularly if you are looking to use the model for interpretation. It is easy to be attracted to the more complex, perhaps more 'fancy' models, but often times a simpler model will do, and perhaps even be easier to implement, saving you time to focus on interpretation.

7 Bibliography

[1] Mushrooms data set: https://archive.ics.uci.edu/ml/datasets/Mushroom, downloaded 04.12.2020

[2] Source mushroom data: Mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf

[3] Fetal health data set: https://www.kaggle.com/andrewmvd/fetal-health-classification, downloaded 11.12.2020

[4] Source Fetal health data: Ayres de Campos et al. (2000) SisPorto 2.0 A Program for Automated Analysis of Cardiotocograms. J Matern Fetal Med 5:311-318 (link)

[5] Fetal health walkthrough: https://www.kaggle.com/pariaagharabi/step-by-step-fetal-health-prediction-99-detailed

[6] Guide to building a decision tree: https://sefiks.com/2018/08/27/a-step-by-step-cart-decision-tree-example/, visited 04.12.2020

[7] Decision tree learning and the CART algorithm: https://en.wikipedia.org/wiki/Decision_tree_learning, visited 04.12.2020

[8] Feature selection with scikit-learn: https://scikit-learn.org/stable/modules/feature_selection.html#univariat feature-selection, visited 08.12.2020

[9] Breiman, Leo; Friedman, J. H.; Olshen, R. A.; Stone, C. J. (1984). Classification and regression trees. Monterey, CA: Wadsworth & Brooks/Cole Advanced Books & Software. ISBN 978-0-412-04841-8.

[10] Logical rules for classifying mushrooms: logical rules for mushrooms: Duch W, Adamczak R, Grabczewski K, Ishikawa M, Ueda H, Extraction of crisp logical rules using constrained backpropagation networks - comparison of two new approaches, in: Proc. of the European Symposium on Artificial Neural Networks (ESANN'97), Bruge, Belgium 16-18.4.1997, pp. xx-xx

[11] On pruning decision trees: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_prun glr-auto-examples-tree-plot-cost-complexity-pruning-py, visited 03.12.2020

[12] Decision tree code: https://medium.com/datadriveninvestor/easy-implementation-of-decision-tree-with-python-numpy-9ec64f05f8ae, visited 06.12.2020

[12] Guide on feature selection for categorical variables: https://machinelearningmastery.com/feature-selection-with-categorical-data/, visited 08.12.2020

[13] Project 2: https://github.com/emiliefj/FYS-STK3155/blob/master/Project2/Report/Project

[14] SciKit-learn: https://scikit-learn.org/stable/index.html

[15] Numpy: https://numpy.org/

[16] MatPlotLib.PyPlot: https://matplotlib.org/api/pyplot_api.html