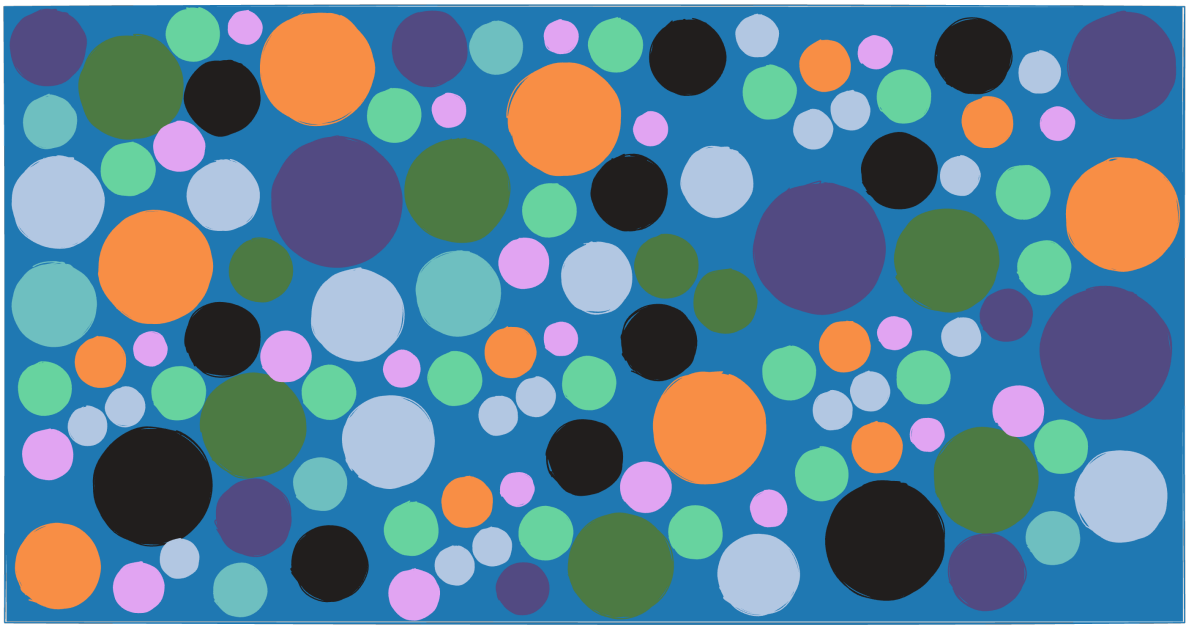
ZZSC5836 Assignment 3

Predicting the Age of abalone from physical measurements

Ty Lange-Smith
z5463091
t.langesmith@student.unsw.edu.au



Hand-drawn Abstract Representation of an Ensemble Fit on an Abalone's Physical Characteristics

Table of Contents

Introduction

This report aims to conduct exploration, analysis and modelling on an abalone dataset. The goal is to develop a predictive model that can classify the age class of abalones from their physical characteristics. These classes are as follows:

- Class 1: 0 – 7 years
- Class 2: 8 – 10 years
- Class 3: 11 – 15 years
- Class 4: > 15 years

Currently, the method for determining the age of an abalone is both invasive and time-consuming. This method involves a multi-step procedure where the abalone's shell must be cut, stained to highlight the growth rings, and then examined under a microscope to count these rings which corresponds to the abalone's age (Nash et al., 1995).

There are other more accessible and non-invasive measurements that can serve as an indicator of an abalone's age. Using these measurements, we aim to construct a predictive model that offers an alternative to the traditional approach.

Data Set Overview

The dataset can be found https://archive.ics.uci.edu/dataset/1/abalone

The dataset contains a variety of information about various abalone physical characteristics. In the dataset we have 4177 rows and 9 columns. These columns are as follows:

Column Name	Type	Description
Sex	Feature	Male, Female or Infant
Length	Feature	Longest shell measurement (mm)
Diameter	Feature	Perpendicular to length (mm)
Height	Feature	with meat in shell (mm)
Whole Weight	Feature	Grams weight of meat (grams)
Shucked Weight	Feature	Grams weight of meat (grams)
Viscera Weight	Feature	Grams gut weight, after bleeding (grams)
Shell Weight	Feature	Grams after being dried (grams)
Rings	Label	The age in years (integer)

These measurements were collected from a diverse range of abalone to ensure a representative sample of the population was surveyed. However, it's worth noting that the representative sample may contain unknown variations which may impact the model's ability to generalise.

Data Processing

All code can be found on https://github.com/tylangesmith/ZZSC5836-assignment-3

Data Processing code can be found in the data_processing.ipynb notebook.

Data Cleaning and Class Conversion

1. Clean the data and convert the rings column into the 4 major class groups.

Taking an initial look at our data:

```
# Lets take a quick look at our data
df.head()
```

✓ 0.0s

	sex	length	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

We can see that we have the previously described feature and label columns. Let's now take a quick look at some high-level statistics for our data.

```
# Lets also have a quick describe of our data
df.describe()
```

✓ 0.0s

	length	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	rings
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000
mean	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.238831	9.933684
std	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.139203	3.224169
min	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.001500	1.000000
25%	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.130000	8.000000
50%	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.234000	9.000000
75%	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.329000	11.000000
max	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.005000	29.000000

One notable observation here is the presence of 0 values in the height column, which is biologically implausible. This is something we'd want to further investigate and clarify with our stakeholders.

We do have a few options to address this issue, we could apply an imputation method to the rows with 0 values, however because the number of impacted rows is very small we will instead just remove the impacted rows from the dataset rather than potentially introducing bias via imputation.

```
# Remove the rows where height is 0
df = df[df['height'] != 0]
```

✓ 0.0s

Continuing we can again, see that our sex column is nominal and for modelling we require this to be in a numeric form – let's process this.

```
# We first need to convert our sex to a numeric value
def sex_func(sex):
    if sex == 'M':
        return 0
    elif sex == 'F':
        return 1
    return 2

df['sex'] = df['sex'].apply(sex_func)
df.head()
```

✓ 0.0s

	sex	length	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	rings
0	0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	0	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	1	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	0	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	2	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Now we need to convert our rings column into the 4 main age group classes. These are: 0 – 7 years, 8 – 10 years, 11 – 15 years, and greater than 15 years.

These 4 classes are what we will be trying to classify in our modelling.

```
def age_group_func(rings):
    if rings <= 7:
        return 'Class 1: 0-7 Years'
    elif rings <= 10:
        return 'Class 2: 8-10 Years'
    elif rings <= 15:
        return 'Class 3: 11-15 Years'
    else:
        return 'Class 4: > 15 Years'

df['age_group'] = df['rings'].apply(age_group_func)
df.head()
```

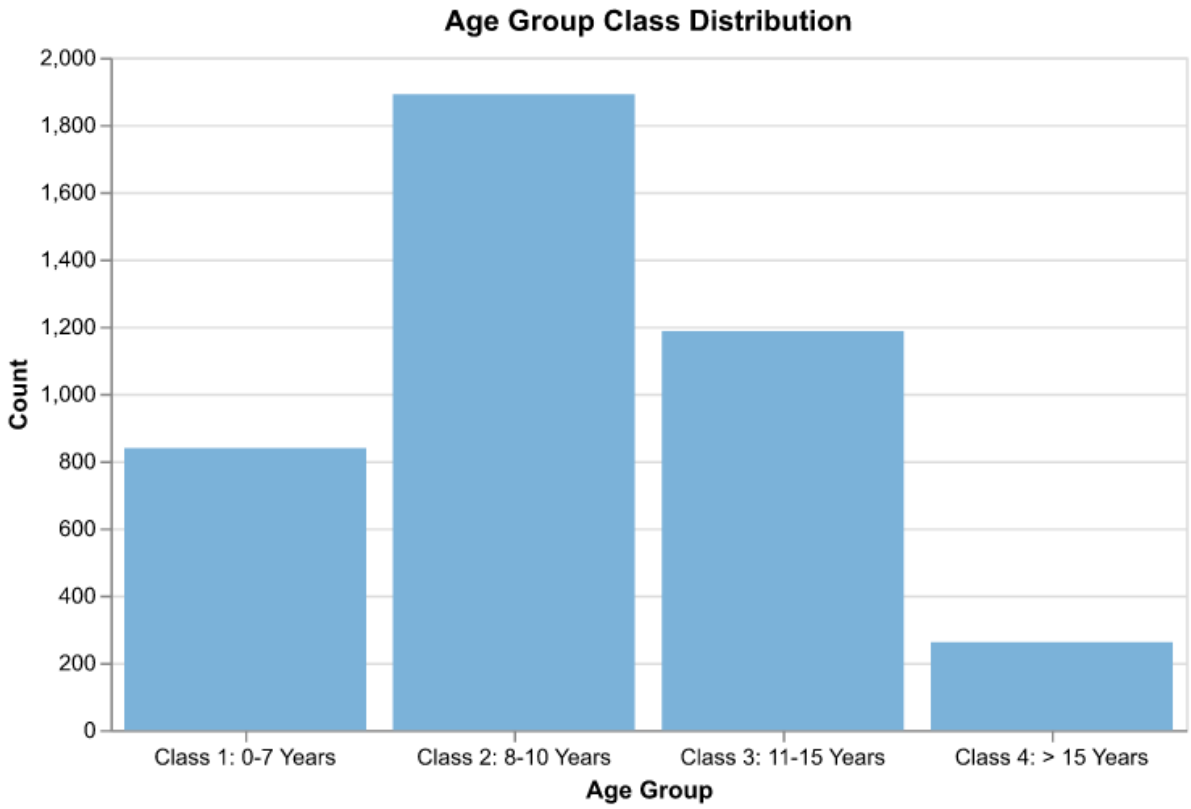
✓ 0.0s

	sex	length	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	rings	age_group
0	0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15	Class 3: 11-15 Years
1	0	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7	Class 1: 0-7 Years
2	1	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9	Class 2: 8-10 Years
3	0	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10	Class 2: 8-10 Years
4	2	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7	Class 1: 0-7 Years

Data Analysis and Visualisation

2. Analyse and visualise the given data sets by reporting the distribution of class, distribution of features and any other visualisation you find appropriate.

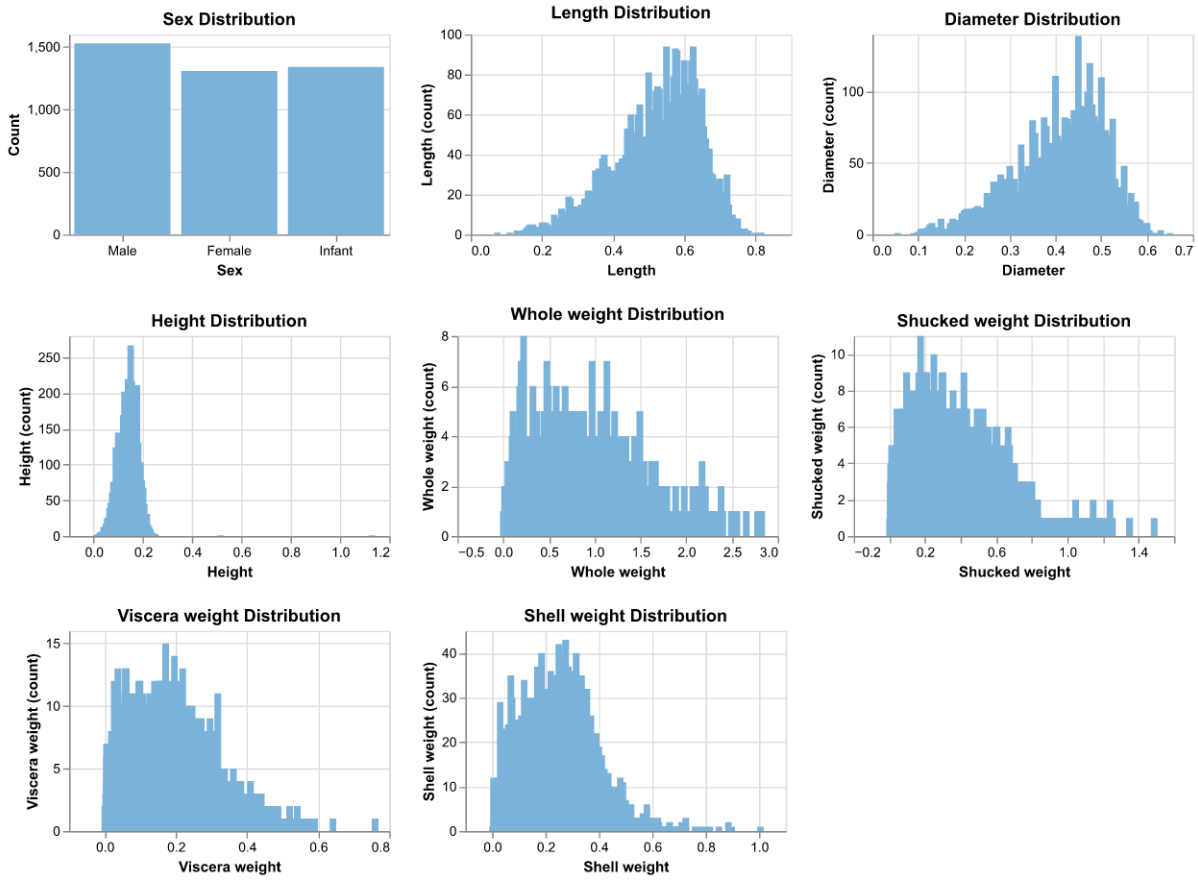
Let's start off by looking at our class distribution.



In the above we can observe that each of our 4 classes have varying instance counts; with Class 2 (abalones aged between 8 and 10 years) being the most common and Class 4 (abalones older than 15 years) being the least common.

This class imbalance is something we need to consider when evaluating our model. For example, if our model predicted every abalone was of class 2 the model would be moderately accurate – however that isn't useful in practice. Instead, we'll use F1 score which balances precision and recall. This metric will provide a better evaluation from that of accuracy.

Next, let's look at the feature distribution to get an overview of the physical characteristics of the abalone in our dataset.



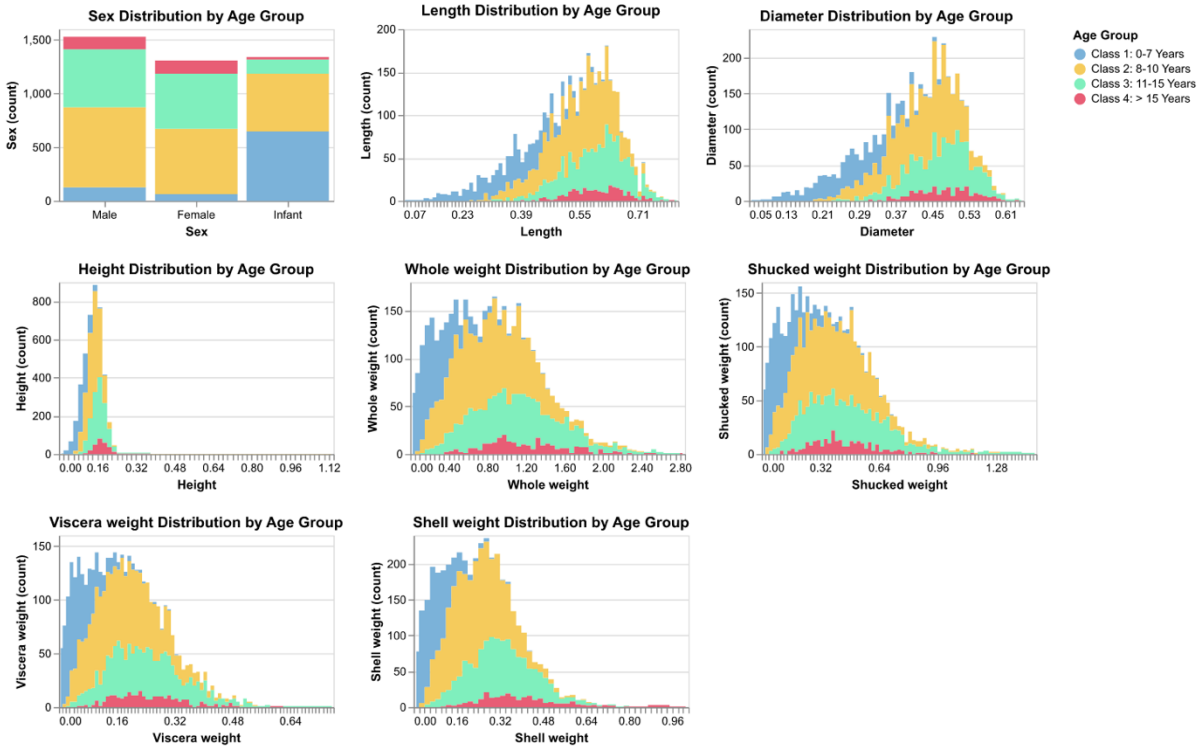
From this visualisation we can draw a few key observations. Firstly, our sex distribution is relatively balanced with a slight predominance of males.

Next, both the length and diameter distributions look to be normally distributed indicating that most abalones have a moderate length and diameter with fewer abalone in our dataset having smaller or larger values.

We can also see that although height looks to follow a normal distribution it isn't as symmetrically balanced as the length and diameter. This indicates that the height column has some extreme outliers. Again, this is something we'd want to clarify with our project stakeholders.

Finally, we have our last 4 feature distributions whole weight, shucked weight, viscera weight and shell weight. These 4 features have a distribution that is skewed right. This skewness indicates that most abalones have a lower weight across the 4 features; this may be indicative of underlying growth patterns, where fewer abalones achieve higher weights.

Now that we have a general understanding of these feature distributions at a high level let's see how they relate to the age groups.



In the above we can again see the same distributions as before; however, we can now see how these distributions relate to age group.

Firstly, looking at our sex feature we see that most of our infant abalones falls within the youngest age group class. We can also see an inverse trend across our male and female abalones with only a small portion being from this youngest age group. This overall trend makes sense and is in line with what you'd expect from the abalone population as they mature from infants to adolescence.

Looking again at our length and diameter features we can observe that both features increase as the age group increase. This gives a good indication that these features will be a good predictor of age. A similar trend is also observed for the height feature.

Finally, looking at our last 4 features whole weight, shucked weight, viscera weight and shell weight we can again observe that as these features increase so does the age group.

Modelling

Modelling with CART

3. Now you need to apply CART for the same problem and report the classification performance on the train and test set using the same train/test split.

We now want to fit a classification and regression tree (CART) model to our data. We'll run 10 experiments and report the performance accuracy where our performance accuracy will be the F1 score.

Let's fit the default sklearn.tree.DecisionTreeClassifier CART model. Notably this uses Gini as it's impurity measure and does not set a maximum tree depth.

Each experiment will fit and evaluate the CART model with a random training and testing set with a split of 60% and 40% respectively.

```
# Now lets calculate the mean f1 score and the confidence interval
f1_train_mean = np.mean([result.f1_train for result in results])
f1_test_mean = np.mean([result.f1_test for result in results])
f1_test_ci = stats.norm.interval(0.95, loc=f1_test_mean, scale=np.std([result.f1_test for result in results]))

print(f'f1_train_mean: {f1_train_mean}')
print(f'f1_test_mean: {f1_test_mean}')
print(f'f1_test_confidence_interval: {f1_test_ci}')
```

✓ 0.0s

```
f1_train_mean: 1.0
f1_test_mean: 0.5338225915615815
f1_test_confidence_interval: (0.5166866197068232, 0.5509585634163399)
```

Calculating the classification performance of our default CART model across our 10 experiments we can see a significant difference in performance between the training and testing sets. The training F1 score is 1.0, indicating a perfect classification on the training data. This is a common occurrence with decision trees, especially when they are allowed to grow without constraints, as they tend to overfit the training data by creating very complex models that capture all the noise in the training set.

On the test set, however, the mean F1 score is approximately 0.534, which is significantly lower than the training performance. This suggests that the model's ability to generalise to unseen data is limited. The confidence interval for the test F1 score ranges from approximately 0.517 to 0.551, providing an estimate of the precision of the F1 score mean estimate.

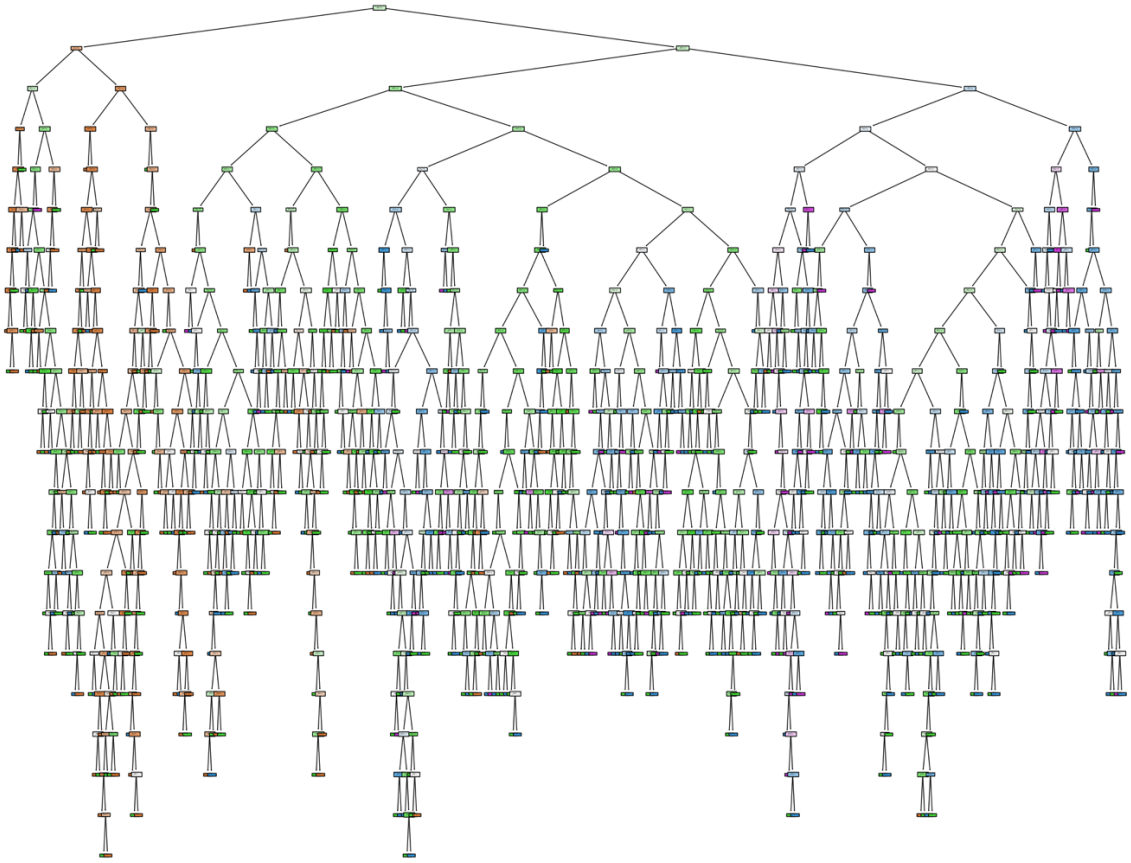
The perfect training score accompanied by a lower test score underlines the importance of setting constraints on the decision tree, such as pre-pruning or post-pruning to prevent the model from overfitting and to improve its generalization to new, unseen data.

Visualising the CART Tree

4. Report the Tree Visualisation (show your tree and also translate few selected nodes and leaves into IF and Then rules)

Naturally, because our CART model doesn't have any pre-pruning regularisation techniques applied the tree can grow as required, this result in our model growing in complexity.

With a total of 710 leaves and a depth of 27 we get the following tree visualisation.



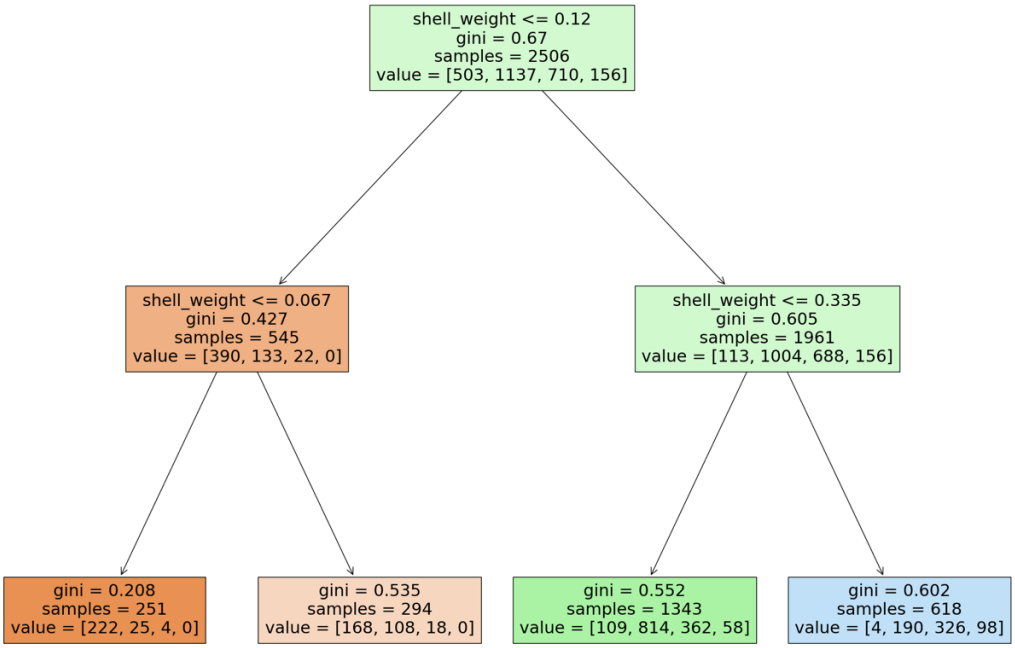
As we can see the model has grown in complexity, and as we saw in our evaluation metrics has overfit to the training data set and doesn't generalise to new unseen data well.

Let's fit a new CART model and restrict the max depth so we can better visualise what's happening.

```
# Lets fit a small CART model so we can visualise it
result = cart(df_X, df_y, max_depth=1)
```

✓ 0.0s

Now with a total of 4 leaves and a depth of 2 we get the following tree visualisation.



This tree can be translated into a set of if else conditional statements.

```
def predict(shell_weight):
    # Start at root Node 0
    if shell_weight <= 0.12:
        # Go to Node 1
        if shell_weight <= 0.067:
            # Go to Node 2
            # Return class 0 because that is the majority class that
            # made it to this particular node
            # as dictated by the value array = [222, 25, 4, 0]
            return 0
        else:
            # Go to Node 3
            # Return class 0 because that is the majority class
            # as dictated by the value array = [168, 108, 18, 0]
            return 0
    else:
        # Go to Node 4
        if shell_weight <= 0.335:
            # Go to Node 5
            # Return class 1 because that is the majority class that
            # made it to this particular node
            # as dictated by the value array = [109, 814, 362, 58]
            return 1
        else:
            # Go to Node 6
            # Return class 2 because that is the majority class that
            # made it to this particular node
            # as dictated by the value array = [4, 190, 326, 98]
            return 2
```

This sort of translation is great for model explainability and enables us to effectively communicate, understand and govern why our model made a specific decision (Scikit-Learn Developers, 2023).

Post-Pruning CART

5. Do an investigation about improving performance further by either pre-pruning or post-pruning the tree: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html

As we previously mentioned the default CART model was unrestricted in how it was able to grow its tree structure to fit the training dataset. This resulted in the tree overfitting the training dataset and not being able to generalise well to the testing dataset.

To address this overfitting problem and to try and improve the performance of our model let's explore post pruning to regularise our decision tree.

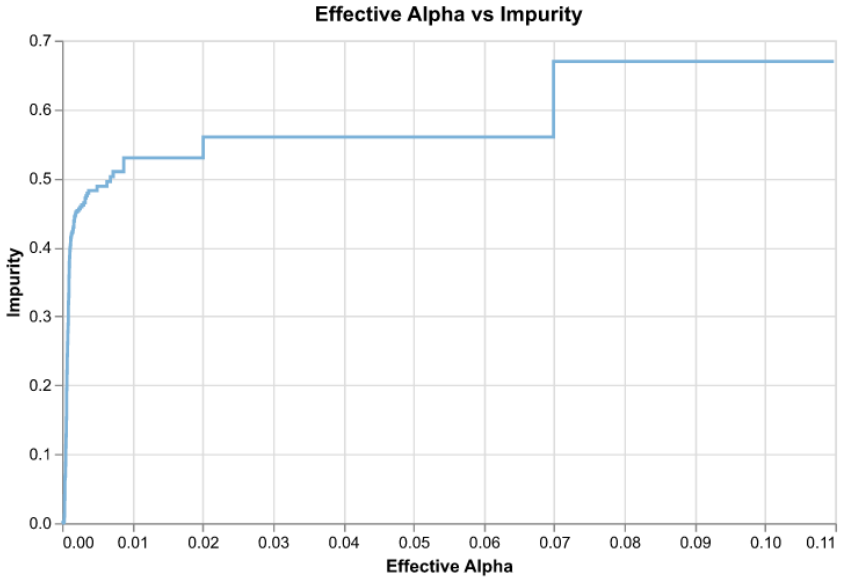
Sklearn provides a cost_complexity_pruning_path function which implements minimal cost complexity pruning. Minimal cost complexity pruning is used to simplify the decision tree by recursively removing nodes that provide little classification power which is characterised by an effective alpha. As the effective alpha increase more of the tree is pruned resulting in a higher total impurity.

The cost_complexity_pruning_path function returns a list of effective alphas and total impurities at each pruning step. Let's fit another CART model, generate the cost complexity pruning path.

```
# Fit a cart model without restricting the depth
result = cart(df_X, df_y)

# Generate the cost complexity pruning path
ccp_results = result.model.cost_complexity_pruning_path(result.df_X_train, result.df_y_train)
```

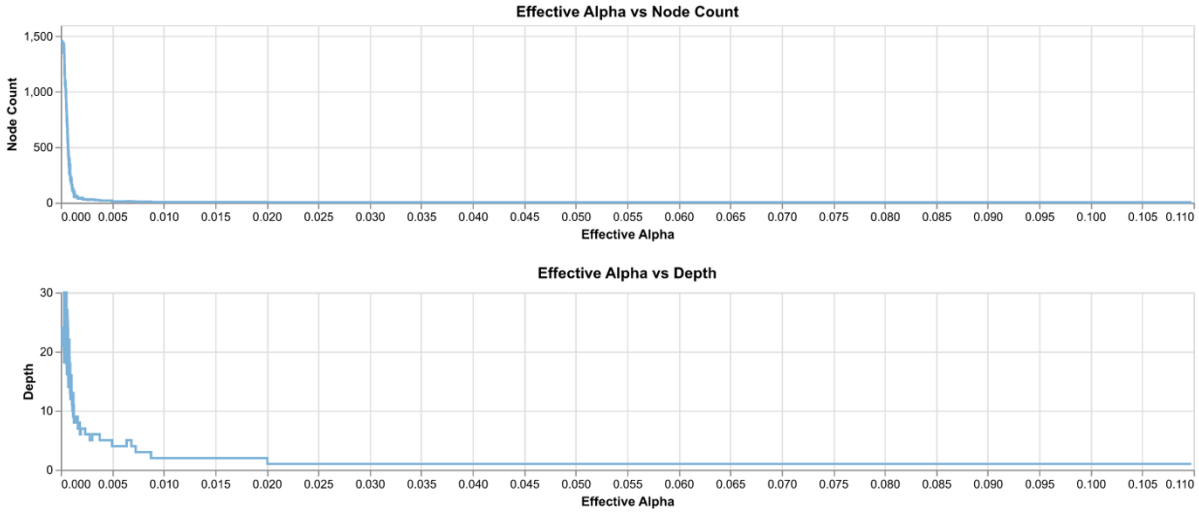
✓ 0.0s

Now let's visualise the effective alpha and impurities.



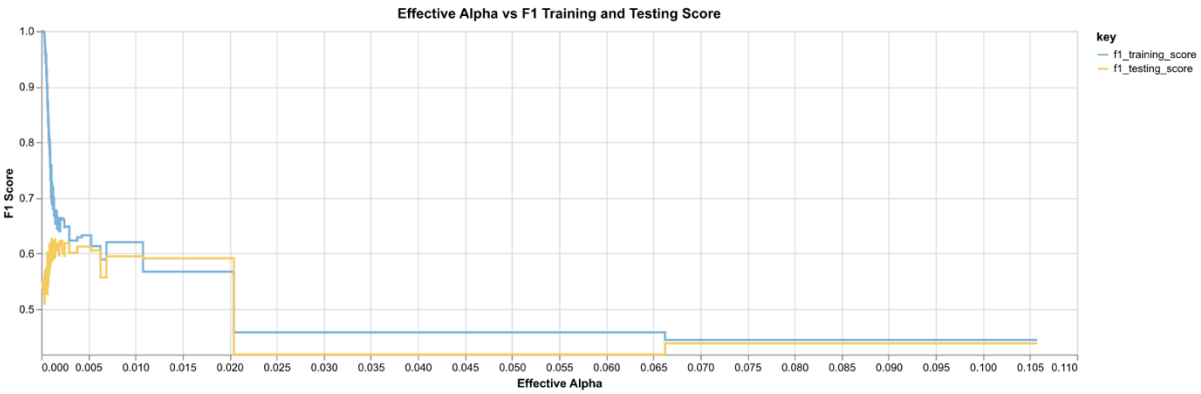
This visualisation shows the expected behaviour of impurity increasing as effective alpha increases.

Next, we want to train a CART model for each of these effective alphas.



In the above visualisation we can see the expected result of as the effective alpha increase the complexity (node count and depth) of the model decreases.

Finally, let's look at how this relates to our training and testing performance.



This visualisation shows as the effective alpha increases the testing score begins to improve from the unpruned initial tree with 0 effective alpha. This indicates as we reduce the complexity of the tree the tree begins to generalise and as a result, we see an improved performance on unseen data (Scikit-Learn Developers, 2023).

However, as we start to overly simplify the tree the performance on both the training and testing sets degrades, indicating the tree is starting to underfit.

```
# Let's output our best model
highest_f1_testing_score = max(ccp_results['f1_testing_score'])
result_index = ccp_results['f1_testing_score'].index(highest_f1_testing_score)

ccp_alpha = ccp_results['ccp_alphas'][result_index]
impurity = ccp_results['impurities'][result_index]
node_count = ccp_results['node_count'][result_index]
depth = ccp_results['depth'][result_index]
f1_training_score = ccp_results['f1_training_score'][result_index]
f1_testing_score = ccp_results['f1_testing_score'][result_index]

print(f"Effective Alpha: {ccp_alpha:.6f}, Impurity: {impurity:.6f}, Node Count: {node_count}, Max Depth: {depth}, F1 Training:
{f1_training_score:.3f}, F1 Testing {f1_testing_score:.3f}")
```

✓ 0.0s

```
Effective Alpha: 0.001112, Impurity: 0.371634, Node Count: 115, Max Depth: 11, F1 Training: 0.698, F1 Testing 0.627
```

Picking the best model from these pruning iterations we get the model pruned with an effective alpha of 0.001112, impurity of 0.371634, has 115 nodes, a max depth of 11, has a training F1 score of 0.698 and a testing F1 score of 0.627.

Bagging of Trees via Random Forest

6. Apply Bagging of Trees via Random Forests and show performance (eg. accuracy score) as your number of trees in the ensembles increases.

After examining the performance of a single decision tree in our data modelling, we now turn our attention to an ensemble method: the Random Forest. By utilising the sklearn.ensemble.RandomForestClassifier, we aim to leverage the combined strength of multiple decision trees.

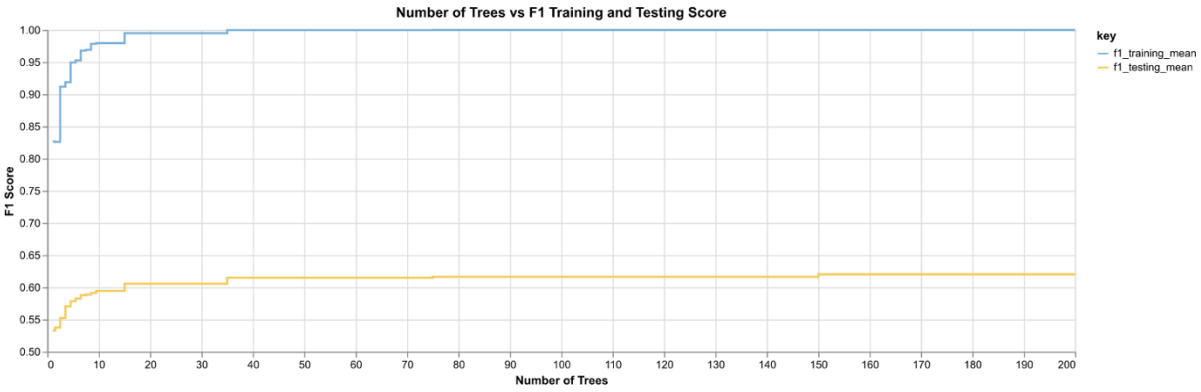
This method works by fitting numerous decision tree classifiers, each on different subsamples of the training dataset. The Random Forest algorithm then aggregates the predictions from all the trees to arrive at a final prediction. This process, often referred to as "bagging", enhances the model's accuracy by reducing variance and preventing overfitting.

Let's now look at how this model performs as the number of trees in the ensemble increases. For this we'll use the following number of tree variations: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100, 200. For each of these variations we'll run 50 experiments fitting a random forest classifier with different random training and testing samples all with again a split of 60% and 40% respectively.

```
results_f1_training_mean = []
results_f1_testing_mean = []
number_of_trees = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100, 200]

for trees in number_of_trees:
    results = []

    for i in range(50):
        result = random_forest(df_X, df_y, n_estimators=trees)
        results.append(result)

    results_f1_training_mean.append(np.mean([result.f1_train for result in results]))
    results_f1_testing_mean.append(np.mean([result.f1_test for result in results]))
```

✓ 1m 0.6s

Let's now look at how each of these variations performed on average.



In the above visualisation we can see a few interesting observations. As the number of trees in the ensemble increases so does both the training and testing performance. However, after a certain point, the increase in the number of trees yields diminishing returns in terms of model performance improvement.

This indicates that adding more trees above a certain threshold does not significantly improve the model's ability to generalise to unseen data. Furthermore, this suggests that there is an optimal number of trees where the model performance is maximised and adding additional trees past this point adds additional complexity.

References

Nash,Warwick, Sellers,Tracy, Talbot,Simon, Cawthorn,Andrew, and Ford,Wes. (1995). Abalone. UCI Machine Learning Repository. https://doi.org/10.24432/C55C7W.

Scikit-Learn Developers. (2023). Post pruning decision trees with cost complexity pruning. Scikit-Learn 1.3.2 documentation. Retrieved from https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html.

Scikit-Learn Developers. (Year). Decision Trees. Scikit-Learn 1.3.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/tree.html.