Assignment 2

July 9, 2023

22975276 Lachlan Bassi ## 1.0 Importing the breast cancer data

The load_breast_cancer() function in Scikit learn loads the Wisconsin breast cancer dataset. This data set is made of two classes: data and target - target is a binary value indicating whether the patient is diagnosed with cancer and data is the measured variables. More information about the function can be found at:

https://scikit-learn.org/0.20/modules/generated/sklearn.datasets.load_breast_cancer.html

0.0.1 Target and Feature Variables

The target variable is assigned to Y and the feature variables are assigned to X.

The results show that there are 30 feature variables and 569 records.

[1]:
```
import numpy as np
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from graphviz import Source
from sklearn.tree import export_graphviz
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix, precision_score, recall_score
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVR
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.ensemble import VotingRegressor
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import BaggingRegressor
from sklearn.svm import SVR
from sklearn.feature_selection import SelectFromModel
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import VotingRegressor
from sklearn.tree import DecisionTreeRegressor


# Load the breast cancer dataset
data = load_breast_cancer()
# Access features and target variables
X = data.data
y = data.target

# Access feature names and target names
feature_names = data.feature_names
target_names = data.target_names

# Print the shape of the dataset
print("Data shape:", X.shape)
print("Target shape:", y.shape)
```

```
Data shape: (569, 30)
Target shape: (569,)
```

0.1 Visualising the variables

The results show the feature variables and target variable in a table structure.

[2]:
```
# Create a DataFrame with the data and feature names
df = pd.DataFrame(X, columns=feature_names)

# Add the target variable to the DataFrame
df["target"] = y
df.head()
```

[2]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	\
0	17.99	10.38	122.80	1001.0	0.11840	
1	20.57	17.77	132.90	1326.0	0.08474	
2	19.69	21.25	130.00	1203.0	0.10960	
3	11.42	20.38	77.58	386.1	0.14250	
4	20.29	14.34	135.10	1297.0	0.10030	

	mean compactness	mean concavity	mean concave points	mean symmetry	\
0	0.27760	0.3001	0.14710	0.2419	
1	0.07864	0.0869	0.07017	0.1812	
2	0.15990	0.1974	0.12790	0.2069	
3	0.28390	0.2414	0.10520	0.2597	
4	0.13280	0.1980	0.10430	0.1809	

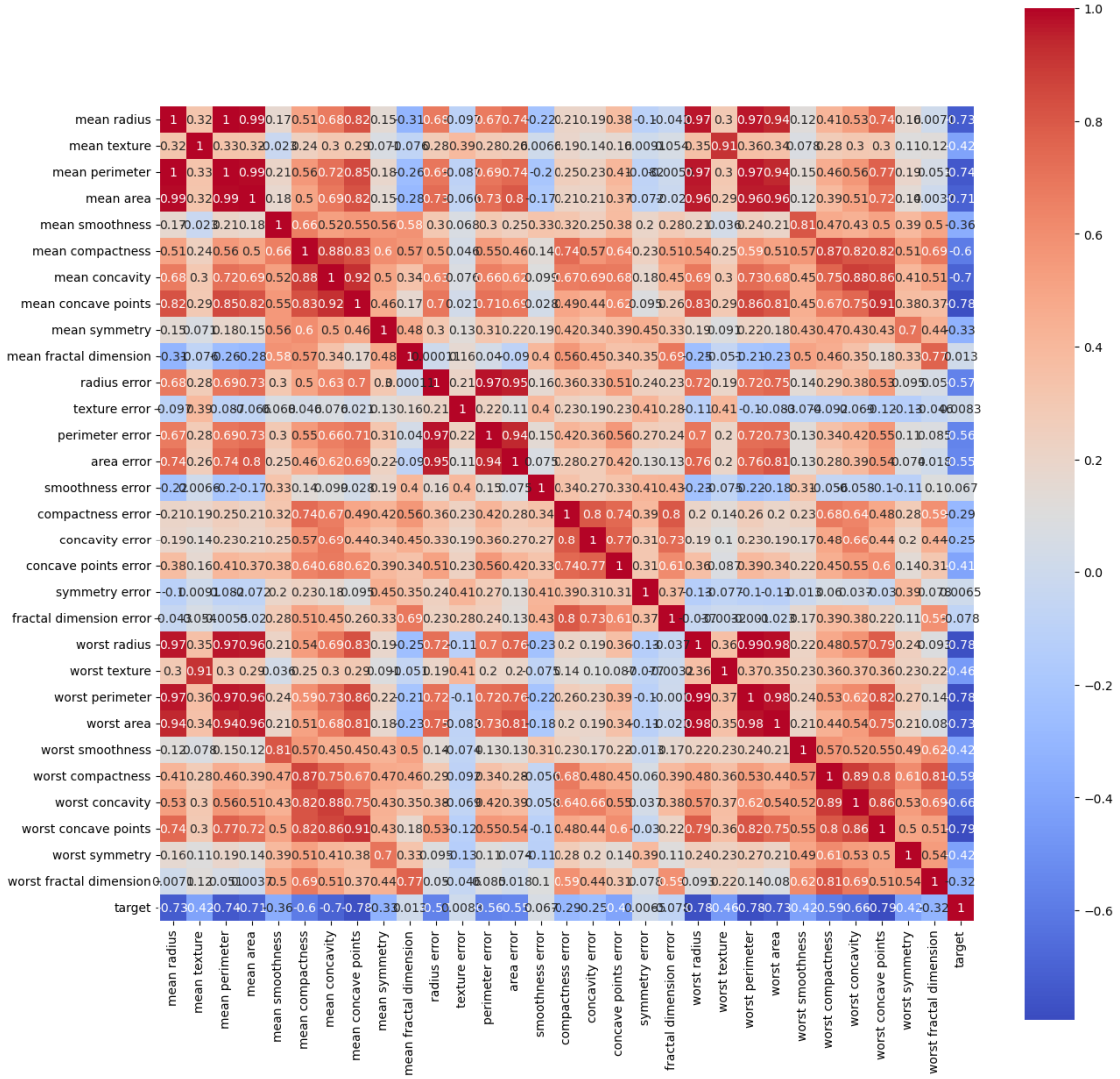
```
   mean fractal dimension  …  worst texture  worst perimeter  worst area  \
0                 0.07871  …          17.33           184.60      2019.0
1                 0.05667  …          23.41           158.80      1956.0
2                 0.05999  …          25.53           152.50      1709.0
3                 0.09744  …          26.50            98.87       567.7
4                 0.05883  …          16.67           152.20      1575.0

   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   worst symmetry  worst fractal dimension  target
0          0.4601                  0.11890       0
1          0.2750                  0.08902       0
2          0.3613                  0.08758       0
3          0.6638                  0.17300       0
4          0.2364                  0.07678       0

[5 rows x 31 columns]
```

0.2 Visualising the Data

The correlation matrix created shows the correlation between the variables to eachother.

The correlation matrix shows there are a number of variables highly correlated to each other. These should be removed to reduce complexity especially since highly correlated variables tend to have little significant impact on the final model.

[3]:
```
# Compute the correlation matrix
corr_matrix = df.corr()

# Plot the heatmap
plt.figure(figsize=(15, 15))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", square=True)
plt.show()
```



0.3 1.1 Removing correlated features

I set a threshold value of 0.9 meaning that if there exists a variable that posseses a correlation greater then 0.9 with another variable then it should be removed.

The results of this show 20 remaining feature variables (originally 30).

The removed features are:

1. mean perimeter
2. mean concave points
3. mean concavity
4. mean texture
5. radius error
6. perimeter error
7. mean area

8. worst radius
9. mean radius
10. worst perimeter

[4]:
```
# Define the threshold
threshold = 0.9

to_drop = []
while True:
    dropped = False
    for idx_i, col_i in enumerate(corr_matrix.columns):
        for idx_j, col_j in enumerate(corr_matrix.columns):
            if idx_i != idx_j and np.abs(corr_matrix.loc[col_i, col_j]) >=␣
 ↪threshold:
                to_drop.append(col_i)
                df = df.drop(col_i, axis=1)
                corr_matrix = df.corr()
                dropped = True
                break
        if dropped:
            break
    if not dropped:
        break

to_drop = list(set(to_drop))
print("Features to be dropped:", to_drop)

df.head()
```

```
Features to be dropped: ['mean radius', 'radius error', 'mean area', 'mean
concavity', 'mean texture', 'perimeter error', 'worst radius', 'mean perimeter',
'worst perimeter', 'mean concave points']
```

[4]:

	mean smoothness	mean compactness	mean symmetry	mean fractal dimension	\
0	0.11840	0.27760	0.2419	0.07871	
1	0.08474	0.07864	0.1812	0.05667	
2	0.10960	0.15990	0.2069	0.05999	
3	0.14250	0.28390	0.2597	0.09744	
4	0.10030	0.13280	0.1809	0.05883	

	texture error	area error	smoothness error	compactness error	\
0	0.9053	153.40	0.006399	0.04904	
1	0.7339	74.08	0.005225	0.01308	
2	0.7869	94.03	0.006150	0.04006	
3	1.1560	27.23	0.009110	0.07458	
4	0.7813	94.44	0.011490	0.02461	

	concavity error	concave points error	…	fractal dimension error	\

```
0          0.05373               0.01587  …                 0.006193
1          0.01860               0.01340  …                 0.003532
2          0.03832               0.02058  …                 0.004571
3          0.05661               0.01867  …                 0.009208
4          0.05688               0.01885  …                 0.005115

   worst texture  worst area  worst smoothness  worst compactness  \
0          17.33      2019.0            0.1622             0.6656
1          23.41      1956.0            0.1238             0.1866
2          25.53      1709.0            0.1444             0.4245
3          26.50       567.7            0.2098             0.8663
4          16.67      1575.0            0.1374             0.2050

   worst concavity  worst concave points  worst symmetry  \
0           0.7119                0.2654          0.4601
1           0.2416                0.1860          0.2750
2           0.4504                0.2430          0.3613
3           0.6869                0.2575          0.6638
4           0.4000                0.1625          0.2364

   worst fractal dimension  target
0                  0.11890       0
1                  0.08902       0
2                  0.08758       0
3                  0.17300       0
4                  0.07678       0

[5 rows x 21 columns]
```

0.4 1.2 Creating training and testing

I now remove the target variable from the data frame so it only contains the feature variables. Then I create and 85% to 15% split of the data into training and testing the dimensions of these classses are printed below.

[5]:
```
# Update the feature matrix X
X = df.drop("target", axis=1).values

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,␣
 ↪random_state=123)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print("Training set size:", X_train.shape)
```

```
print("Test set size:", X_test.shape)
```

```
Training set size: (483, 20)
Test set size: (86, 20)
```

0.5 1.3 Training Decision Tree Classifiers

tree_clf uses the default hyperparameters and is trained on the training set

[6]:
```
tree_clf = DecisionTreeClassifier()

tree_clf.fit(X_train, y_train)
```

[6]:
```
DecisionTreeClassifier()
```

0.5.1 Predicting Accuracy using Training Data

The result of the accuracy being 1 demonstrates a perfect fit for the training data.

[7]:
```
y_pred_clf_train = tree_clf.predict(X_train)
accuracy_score(y_train, y_pred_clf_train)
```

[7]:
```
1.0
```

0.5.2 Using the testing set to form an accuracy score

The result of 0.930 is still quite accurate but there is a significant difference between training and testing.

[8]:
```
y_pred_clf_test = tree_clf.predict(X_test)
accuracy_score(y_test, y_pred_clf_test)
```

[8]:
```
0.9302325581395349
```

0.5.3 Comparing testing and training

While an accuracy score of 1.0 may seem impressive it is important to remember the goal of a machine learning model is to generalise well to new and unseen data and this result is from the training data.

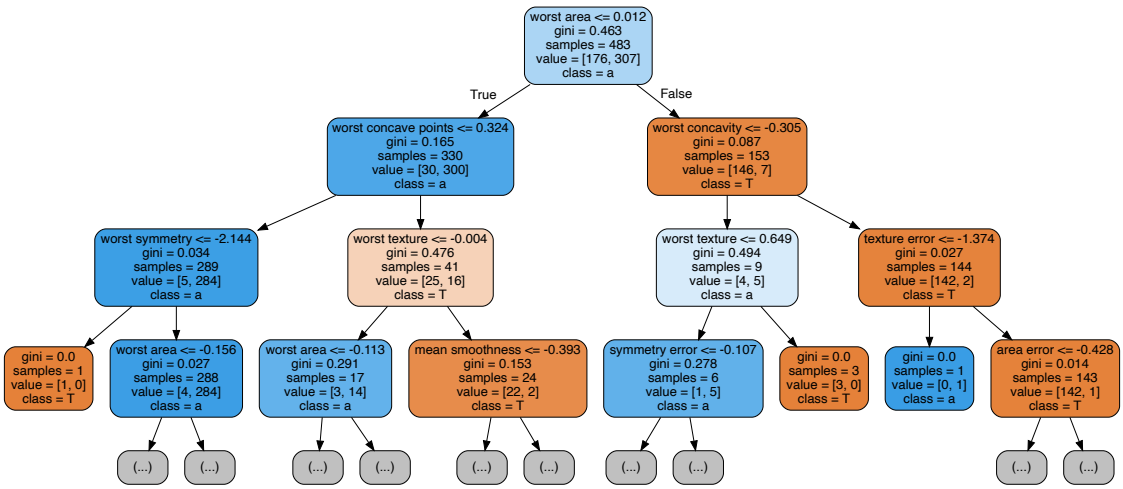
Based on the results generated the performance of the testing set is a more accurate representation of the models performance on unseen data. The accuracy value of 0.930 is quite accurate but still isn't excellent.

0.5.4 Visualising the decision tree classifier in a decision tree

Analysing the decision tree the classes are too specific. Looking only at these three levels there is a number of leaf nodes with a Gini impurity of 0 meaning the decision tree has successfully seperated the data into distinct classes at those nodes.

However, due to the large number of nodes having a Gini impurity of 0, there is more nodes when the tree is expanded, this is a major sign of overfitting due to the algorithms classification criteria being too precise. This can also be observed in the difference in the number of samples between leaf nodes with some on the third level having one sample and others having 143. This demonstrates the splitting criteria has become too complex and is capturing noise in the data a major sign of over fitting.

[9]:
```
feature_names = list(df.columns)
feature_names = feature_names[:-1]
target_names = "Target"

IMAGES_PATH = ""
export_graphviz(
        tree_clf,
        out_file=os.path.join(IMAGES_PATH, "iris_tree.dot"),
        feature_names= feature_names,
        class_names= target_names,
        rounded=True,
        filled=True,
        max_depth=3
    )

Source.from_file(os.path.join(IMAGES_PATH, "iris_tree.dot"))
```

[9]:


0.5.5 Training a new decision tree

This decision tree will have max_depth set to 3 to prevent overfitting as it limits the tree's complexity. This should help create a shallower tree and forces the model to make simpler decision rules that capture broader patterns in the data instead of learning noise specific to the training set.

[10]:
```
tree_clf1 = DecisionTreeClassifier(max_depth = 3)

tree_clf1.fit(X_train, y_train)
```

[10]:
```
DecisionTreeClassifier(max_depth=3)
```

0.5.6 Comparing Accuracy Score of training and testing set

The accuracy scores are closer but there still appears to be evidence of overfitting as the difference is still noticable.

The accuracy score of 0.978 on the training data indicates a high accuracy meaning it is an effective model at learning the patterns in the training set and correctly classifying most samples.

The accuracy score of 0.942 on the testing data demonstrates the model is a lot more accurate then the previous model. Also the difference between the accuracy score of the training and testing sets is a lot smaller meaning overfitting has been reduced and the model is more generalised.

[11]:
```
y_pred_clf1_train = tree_clf1.predict(X_train)
accuracy_score(y_train, y_pred_clf1_train)
```

[11]:
```
0.9772256728778468
```

[12]:
```
y_pred_clf1_test = tree_clf1.predict(X_test)
accuracy_score(y_test, y_pred_clf1_test)
```

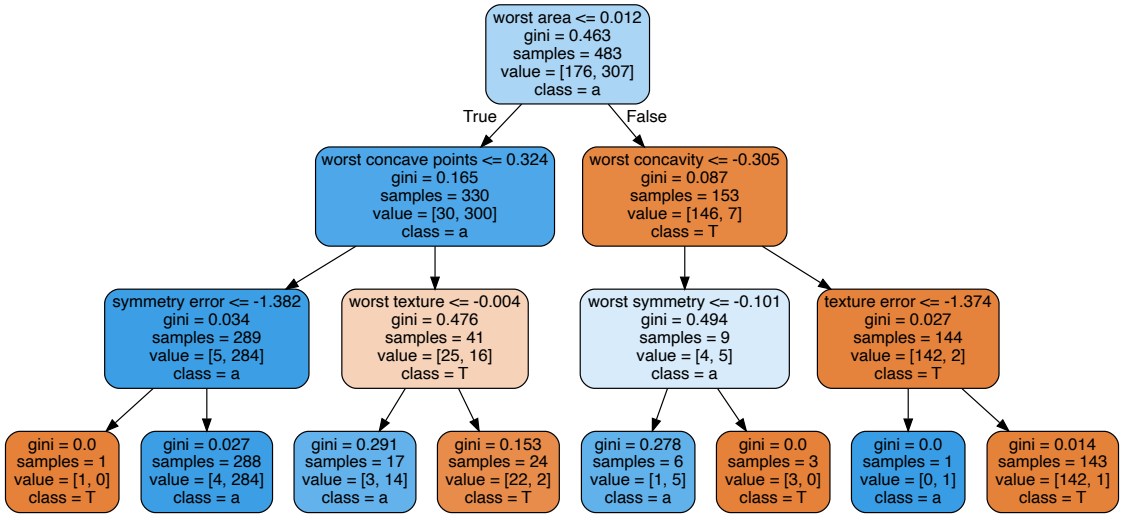
[12]:
```
0.9418604651162791
```

0.5.7 Comparing Decision Tree

The decision tree still doesn't do a great job at splitting the data with the same number of nodes having multiple gini values of 0. The decision tree has identical nodes on the first three levels but cuts it off at depth of 3 before it can split every individual node meaning the model is now predicting and not sorting. The performance of a max depth of 3 appears to do better then the previous decision tree with a slightly higher accuracy.

[13]:
```
IMAGES_PATH = ""
export_graphviz(
        tree_clf1,
        out_file=os.path.join(IMAGES_PATH, "iris_tree.dot"),
        feature_names= feature_names,
        class_names= target_names,
        rounded=True,
        filled=True
    )
Source.from_file(os.path.join(IMAGES_PATH, "iris_tree.dot"))
```

[13]:



0.6 Fitting a decision tree with min_samples_split = 5

The results of this change show a model with an increased training performance compared to the previous max_depth model but has a slightly worse testing accuracy.

[14]:
```
tree_clf2 = DecisionTreeClassifier(min_samples_split = 5)

tree_clf2.fit(X_train, y_train)
```

[14]:
```
DecisionTreeClassifier(min_samples_split=5)
```

0.6.1 Comparing accuracy scores

There is a greater difference in training and testing and the accuracy is lower for testing when compared to max_depth = 3, this hyperparameter still improves on the initial decision tree but isn't the best performing.

[15]:
```
y_pred_clf2_train = tree_clf2.predict(X_train)
accuracy_score(y_train, y_pred_clf2_train)
```

[15]:
```
0.9937888198757764
```

[16]:
```
y_pred_clf2_test = tree_clf2.predict(X_test)
accuracy_score(y_test, y_pred_clf2_test)
```

[16]:
```
0.9302325581395349
```

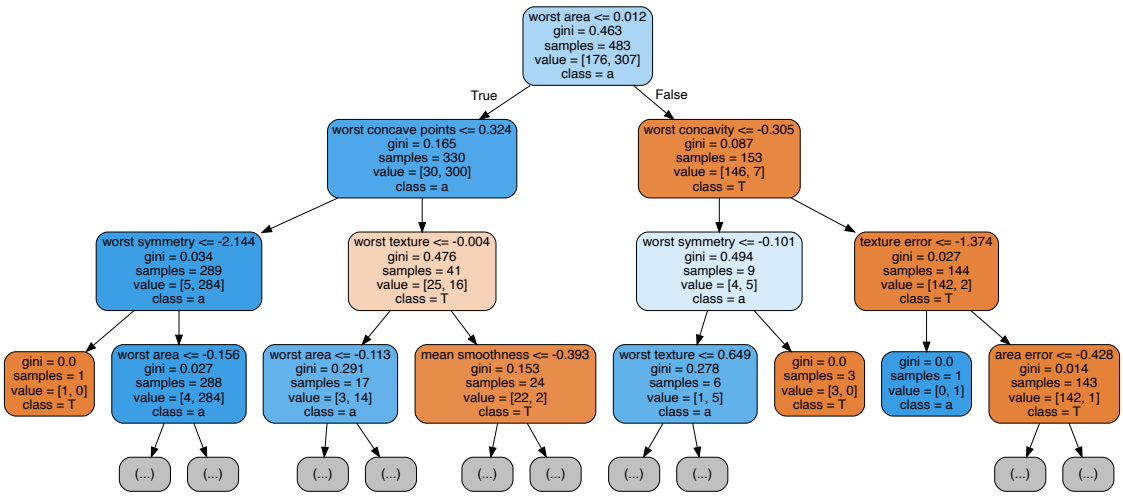
0.7 Analysing Decision Tree

Very similar to the previous two decision trees and once again it appears to suffer from over fitting. The gini values of 0 plus the large difference in samples in the level 3 leaf nodes indicates a complex

sorting method that has been effected by the noise in the data, a clear sign of over fitting.

[17]:
```
IMAGES_PATH = ""
export_graphviz(
        tree_clf2,
        out_file=os.path.join(IMAGES_PATH, "iris_tree.dot"),
        feature_names= feature_names,
        class_names= target_names,
        rounded=True,
        filled=True,
        max_depth=3
    )
Source.from_file(os.path.join(IMAGES_PATH, "iris_tree.dot"))
```

[17]:



0.8 Creating a model Fitting min_samples_leaf = 5

This is not the best performing hyper parameter on the unseen testing data as that still belongs to max_depth however there is still improvement compared to the intial decision tree classifier.

[18]:
```
tree_clf3 = DecisionTreeClassifier(min_samples_leaf = 5)

tree_clf3.fit(X_train, y_train)
```

[18]:
```
DecisionTreeClassifier(min_samples_leaf=5)
```

[19]:
```
y_pred_clf3_train = tree_clf3.predict(X_train)
accuracy_score(y_train, y_pred_clf3_train)
```

[19]:
```
0.968944099378882
```

[20]:
```
y_pred_clf3_test = tree_clf3.predict(X_test)
accuracy_score(y_test, y_pred_clf3_test)
```

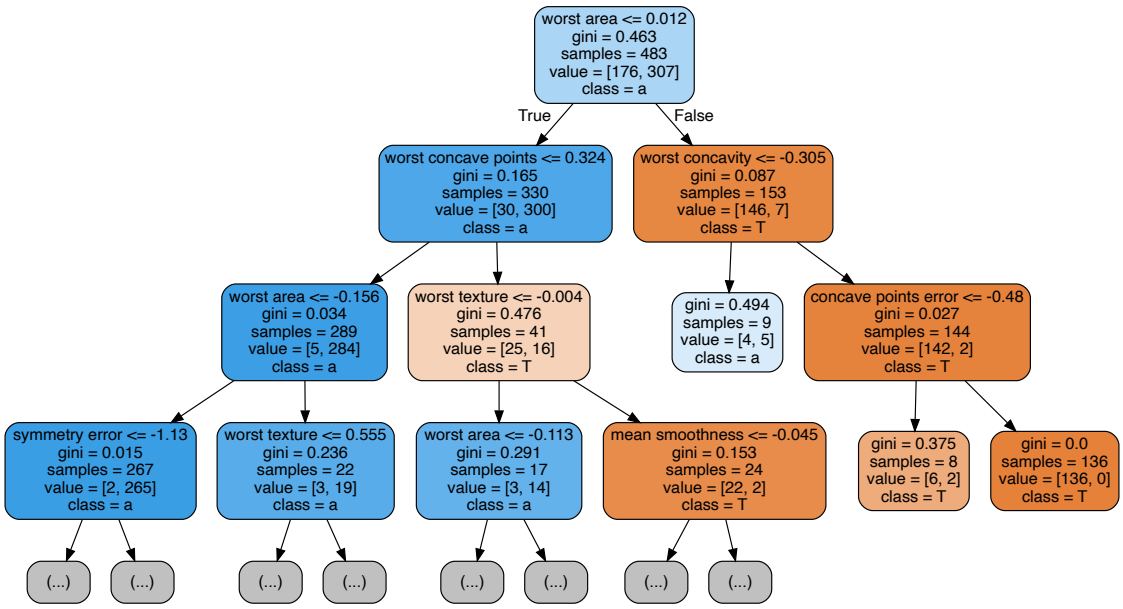
[20]:
```
0.9302325581395349
```

0.8.1 Comparing Decision Tree

Once again the decision tree is not significantly better then the previous trees analysed and still shows signs that it suffers from over fitting: Multiple gini values of 0 and low vs high samples in leaf nodes at the same level

[21]:
```
export_graphviz(
        tree_clf3,
        out_file=os.path.join(IMAGES_PATH, "iris_tree.dot"),
        feature_names= feature_names,
        class_names= target_names,
        rounded=True,
        filled=True,
        max_depth=3
    )
Source.from_file(os.path.join(IMAGES_PATH, "iris_tree.dot"))
```

[21]:



0.9 1.4 3-Fold cross validation

Using 3-fold cross validation and grid search the best values for the hyperparemets min_samples_leaf, max_depth, and min_samples_split can be found.

After conducting the analysis the best values for these were:

- min_samples_leaf = 4
- max_depth = 3
- min_samples_split = 9

[22]:
```
cross_val_score(tree_clf3, X_train, y_train, cv=3, scoring="accuracy")
```

[22]:
```
array([0.95031056, 0.93167702, 0.88198758])
```

[23]:
```
k_values = [2, 3, 4, 5, 6, 7, 8, 9, 10]  # Try different values of k
param_grid = {'min_samples_leaf': k_values,
              'max_depth': k_values,
              'min_samples_split': k_values}

tree_clf = DecisionTreeClassifier(random_state = 123)
grid_search = GridSearchCV(tree_clf, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)

best_k_min_sample_leaf = grid_search.best_params_['min_samples_leaf']
best_k_max_depth = grid_search.best_params_['max_depth']
best_k_min_sample_split = grid_search.best_params_['min_samples_split']
print(f"Best k min sample leaf = {best_k_min_sample_leaf}")
print(f"Best k max depth = {best_k_max_depth}")
print(f"Best k min sample split = {best_k_min_sample_split}")

# Evaluate the model with the best hyperparameter on the test set
best_tree_clf = grid_search.best_estimator_
y_pred_best_tree_clf_test = best_tree_clf.predict(X_test)
score_test = accuracy_score(y_test, y_pred_best_tree_clf_test)

y_pred_best_tree_clf_train = best_tree_clf.predict(X_train)
score_train = accuracy_score(y_train, y_pred_best_tree_clf_train)

print(f"Accuracy score with training best k = {score_train}")
print(f"Accuracy score with testing best k = {score_test}")
```

```
Best k min sample leaf = 4
Best k max depth = 3
Best k min sample split = 9
Accuracy score with training best k = 0.9710144927536232
Accuracy score with testing best k = 0.9302325581395349
```

0.9.1 1.5 Confusion Matrices and Performance Metrics

The following is the performance metrics for the previous classifiers.

Variables in the confusion matrix are defined as - 0: has cancer - 1: does not have cancer

0.9.2 clf initial hyperparameters

The first confusion matrix is for the initial clf (no hyperparameters). Precision and recall are perfect with no false negatives or positives on the training set but this performance is not the same for the testing set and clearly indicates over fitting.

[24]:
```
# Calculate the confusion matrix

#CLF
cm = confusion_matrix(y_train, y_pred_clf_train)
print("Confusion Matrix Training:")
print(pd.DataFrame(cm))

report = classification_report(y_train, y_pred_clf_train)
print("Classification Report Training:")
print(report)

cm = confusion_matrix(y_test, y_pred_clf_test)
print("Confusion Matrix Testing:")
print(pd.DataFrame(cm))

report = classification_report(y_test, y_pred_clf_test)
print("Classification Report Testing:")
print(report)
```

```
Confusion Matrix Training:
     0    1
0  176    0
1    0  307
Classification Report Training:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       176
           1       1.00      1.00      1.00       307

    accuracy                           1.00       483
   macro avg       1.00      1.00      1.00       483
weighted avg       1.00      1.00      1.00       483

Confusion Matrix Testing:
    0   1
0  32   4
1   2  48
Classification Report Testing:
              precision    recall  f1-score   support

           0       0.94      0.89      0.91        36
           1       0.92      0.96      0.94        50
```

```
    accuracy                           0.93        86
   macro avg       0.93      0.92      0.93        86
weighted avg       0.93      0.93      0.93        86
```

0.9.3 clf1 max_depth = 3

Implementing max_depth = 3, the precision and recall values are much higher for the training compared to the testing and this can be visualised in the confusion matrix. This difference is a clear indication of overfitting.

[25]:
```
#CLF1
cm = confusion_matrix(y_train, y_pred_clf1_train)
print("Confusion Matrix Training:")
print(pd.DataFrame(cm))

report = classification_report(y_train, y_pred_clf1_train)
print("Classification Report Training:")
print(report)

cm = confusion_matrix(y_test, y_pred_clf1_test)
print("Confusion Matrix Testing:")
print(pd.DataFrame(cm))

report = classification_report(y_test, y_pred_clf1_test)
print("Classification Report Testing:")
print(report)
```

```
Confusion Matrix Training:
     0    1
0  168    8
1    3  304
Classification Report Training:
              precision    recall  f1-score   support

           0       0.98      0.95      0.97       176
           1       0.97      0.99      0.98       307

    accuracy                           0.98       483
   macro avg       0.98      0.97      0.98       483
weighted avg       0.98      0.98      0.98       483

Confusion Matrix Testing:
    0   1
0  32   4
1   1  49
Classification Report Testing:
              precision    recall  f1-score   support
```

```
              0       0.97      0.89      0.93        36
              1       0.92      0.98      0.95        50

    accuracy                            0.94        86
   macro avg       0.95      0.93      0.94        86
weighted avg       0.94      0.94      0.94        86
```

0.9.4 CLF2: min_samples_split is set to 5

the performance of this hyper parameter is once again overfitting. The number of false negatives and false positives for having cancer significantly increases in the testing set compared to the training when comparing the confusion matrixes.

[26]:
```
#CLF2
cm = confusion_matrix(y_train, y_pred_clf2_train)
print("Confusion Matrix Training:")
print(pd.DataFrame(cm))

report = classification_report(y_train, y_pred_clf2_train)
print("Classification Report Training:")
print(report)

cm = confusion_matrix(y_test, y_pred_clf2_test)
print("Confusion Matrix Testing:")
print(pd.DataFrame(cm))

report = classification_report(y_test, y_pred_clf2_test)
print("Classification Report Testing:")
print(report)
```

```
Confusion Matrix Training:
     0    1
0  174    2
1    1  306
Classification Report Training:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       176
           1       0.99      1.00      1.00       307

    accuracy                           0.99       483
   macro avg       0.99      0.99      0.99       483
weighted avg       0.99      0.99      0.99       483

Confusion Matrix Testing:
    0   1
```

```
0  32   4
1   2  48
Classification Report Testing:
              precision    recall  f1-score   support

           0       0.94      0.89      0.91        36
           1       0.92      0.96      0.94        50

    accuracy                           0.93        86
   macro avg       0.93      0.92      0.93        86
weighted avg       0.93      0.93      0.93        86
```

0.9.5 CLF3 min_samples_leaf = 5

While the total number of false negatives and false positives is lower in the testing set compared to the training set, the proportion is higher resulting in lower precision and recall values.

This difference can be demonstrated by looking at the precision of predicting cancer is 0.92 on the testing vs precision of having cancer is 0.95 on the training set. Clearly demonstrating the model overfits in this case on predicting cancer and in general on predicting and not predicting cancer.

[27]:
```
#CLF3
cm = confusion_matrix(y_train, y_pred_clf3_train)
print("Confusion Matrix Training:")
print(pd.DataFrame(cm))

report = classification_report(y_train, y_pred_clf3_train)
print("Classification Report Training:")
print(report)

cm = confusion_matrix(y_test, y_pred_clf3_test)
print("Confusion Matrix Testing:")
print(pd.DataFrame(cm))

report = classification_report(y_test, y_pred_clf3_test)
print("Classification Report Testing:")
print(report)
```

```
Confusion Matrix Training:
     0    1
0  170    6
1    9  298
Classification Report Training:
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       176
           1       0.98      0.97      0.98       307
```

```
    accuracy                           0.97       483
   macro avg       0.96      0.97      0.97       483
weighted avg       0.97      0.97      0.97       483

Confusion Matrix Testing:
    0   1
0  33   3
1   3  47
Classification Report Testing:
              precision    recall  f1-score   support

           0       0.92      0.92      0.92        36
           1       0.94      0.94      0.94        50

    accuracy                           0.93        86
   macro avg       0.93      0.93      0.93        86
weighted avg       0.93      0.93      0.93        86
```

0.9.6 Overall Conclusions

- All the models suffer from over fitting and this can be visualised with all models possessing higher recall and precision values for having and not having cancer compared to training and testing data.
- CLF1 was the best performing

[28]:
```
#best tree clf
cm = confusion_matrix(y_train, y_pred_best_tree_clf_train)
print("Confusion Matrix Training:")
print(pd.DataFrame(cm))

report = classification_report(y_train, y_pred_best_tree_clf_train)
print("Classification Report Training:")
print(report)

cm = confusion_matrix(y_test, y_pred_best_tree_clf_test)
print("Confusion Matrix Testing:")
print(pd.DataFrame(cm))

report = classification_report(y_test, y_pred_best_tree_clf_test)
print("Classification Report Testing:")
print(report)
```

```
Confusion Matrix Training:
     0    1
0  167    9
1    5  302
Classification Report Training:
```

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       176
           1       0.97      0.98      0.98       307

    accuracy                           0.97       483
   macro avg       0.97      0.97      0.97       483
weighted avg       0.97      0.97      0.97       483

Confusion Matrix Testing:
    0   1
0  33   3
1   3  47
Classification Report Testing:
              precision    recall  f1-score   support

           0       0.92      0.92      0.92        36
           1       0.94      0.94      0.94        50

    accuracy                           0.93        86
   macro avg       0.93      0.93      0.93        86
weighted avg       0.93      0.93      0.93        86
```

1 2.0 Concrete Slump Test (20 marks)

The concrete slump data set consists of the following attributes:

Input variables (7) (component kg in one M^3 concrete):

1. Cement
2. Slag
3. Fly ash
4. Water
5. SP
6. Coarse Aggr.
7. Fine Aggr.

Output variables (3):

1. SLUMP (cm)
2. FLOW (cm)
3. 28-day Compressive Strength (Mpa)

[29]:
```
data_file_path = "slump_test.data"

delimiter = ','
```

```
data = pd.read_csv(data_file_path, delimiter=delimiter)

print(data.head())
data.columns
```

```
   No  Cement   Slag  Fly ash  Water    SP  Coarse Aggr.  Fine Aggr.  \
0   1   273.0   82.0    105.0  210.0   9.0         904.0       680.0
1   2   163.0  149.0    191.0  180.0  12.0         843.0       746.0
2   3   162.0  148.0    191.0  179.0  16.0         840.0       743.0
3   4   162.0  148.0    190.0  179.0  19.0         838.0       741.0
4   5   154.0  112.0    144.0  220.0  10.0         923.0       658.0

   SLUMP(cm)  FLOW(cm)  Compressive Strength (28-day)(Mpa)
0       23.0      62.0                               34.99
1        0.0      20.0                               41.14
2        1.0      20.0                               41.81
3        3.0      21.5                               42.08
4       20.0      64.0                               26.82
```

[29]:
```
Index(['No', 'Cement', 'Slag', 'Fly ash', 'Water', 'SP', 'Coarse Aggr.',
       'Fine Aggr.', 'SLUMP(cm)', 'FLOW(cm)',
       'Compressive Strength (28-day)(Mpa)'],
      dtype='object')
```

1.1 Removing Columns

The output variables Flow and Slump are removed as the interest of the study is in the 28-day Compressive Strength data

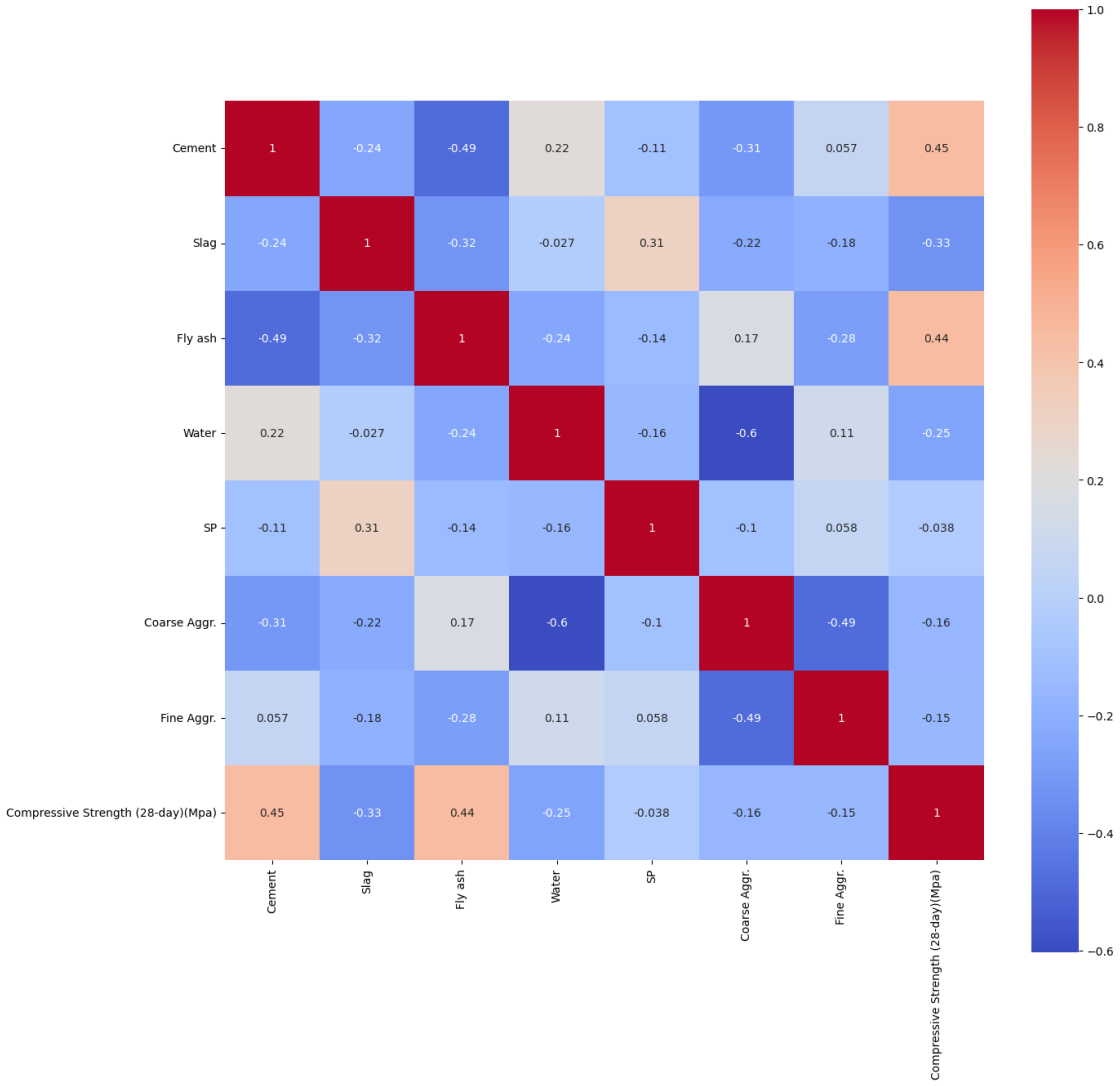
[30]:
```
data = data.drop(columns=['No', 'FLOW(cm)', 'SLUMP(cm)'])
```

1.2 2.1 Visualising Dependancies

Creating a correlation matrix to visualise correlations between feature variables.

Based on the correlation matrix there appears to be no highly correlated variables.

[31]:
```
# Compute the correlation matrix
corr_matrix = data.corr()

# Plot the heatmap
plt.figure(figsize=(15, 15))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", square=True)
plt.show()
```



1.2.1 Seperating into training (size = 0.8) and testing (size = 0.2) sets

[32]:
```
X = data.drop(columns=['Compressive Strength (28-day)(Mpa)'])
y = data['Compressive Strength (28-day)(Mpa)']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=123)

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

1.3 2.2 Voting Regressor

1.3.1 Implementation on the training data

We fit the voting regressor using the intial voting regressor and three different base estimators: linear SVM regressor (svm), linear regressor (lr) using (LinearRegression class), and a Stochastic Gradient Descent regressor (sgd).

The RMSE values for these regressors are

- svm: 2.817
- lr: 2.394
- sgd: 2.403
- Initial Voting Regressor: 2.448

The lowest RMSE is lr meaning it has the best peformance among the three base estimators and the initial voting regressor on the training data. This suggests that using a linear regression model as a base estimator is more suitable for the concrete slump training data.

[33]:
```
estimators = [
    ('svm', LinearSVR()),
    ('lr', LinearRegression()),
    ('sgd', SGDRegressor(max_iter=500))
]
# Create and train the VotingRegressor
voting_regressor = VotingRegressor(estimators)
voting_regressor.fit(X_train_scaled, y_train)

# Calculate RMSE for each model
train_rmse_scores = []
train_predictions = []
for name, estimator in voting_regressor.named_estimators_.items():
    y_pred = estimator.predict(X_train_scaled)
    rmse = np.sqrt(mean_squared_error(y_train, y_pred))
    train_rmse_scores.append((name, rmse))
    train_predictions.append((name, y_pred))

y_pred = voting_regressor.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_pred))
train_rmse_scores.append(('VotingRegressor', rmse))
train_predictions.append(('VotingRegressor', y_pred))  # add voting regressor␣
 ↪predictions

# Print RMSE scores
for name, rmse in train_rmse_scores:
    print(f'{name}: {rmse:.3f}')
```

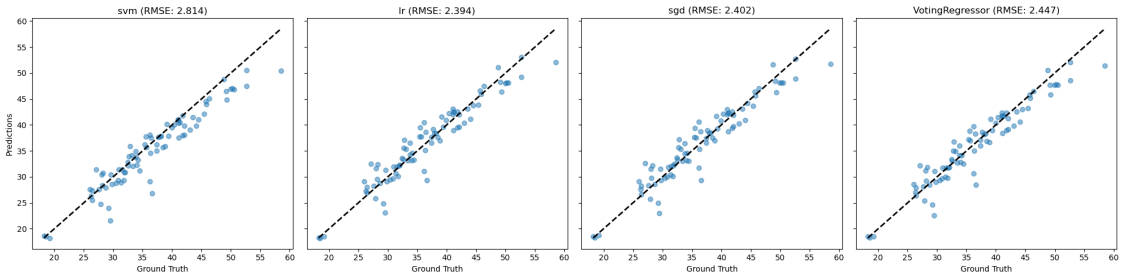
```
svm: 2.814
lr: 2.394
sgd: 2.402
```

```
VotingRegressor: 2.447
```

1.3.2 Visualising plots

The plots show the ground truth values and the predictions, all the predictions tend to be close to the ground truth models meaning all the tested voting regressors do a good job at predicting. This is supported with their similar rmse and it was expected they would have similar plots.

[34]:
```
# Create a scatter plot for each model
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(20, 5), sharey=True)
for i, (name, y_pred) in enumerate(train_predictions):
    rmse = np.sqrt(mean_squared_error(y_train, y_pred))
    axes[i].scatter(y_train, y_pred, alpha=0.5)
    axes[i].plot([y_train.min(), y_train.max()], [y_train.min(), y_train.
 ↪max()], 'k--', lw=2)
    axes[i].set_title(f'{name} (RMSE: {rmse:.3f})')
    axes[i].set_xlabel('Ground Truth')
    if i == 0:
        axes[i].set_ylabel('Predictions')

plt.tight_layout()
plt.show()
```



1.3.3 Implementation on the testing data

RMSE values on testing: - svm: 3.954 - lr: 3.294 - sgd: 3.199 - Initial Voting Regressor: 3.419

A significant increase in RMSE values has occured indicating the models are overfitting as performance is worse on the testing data. In this case the lowest RMSE is sgd with 3.199 meaning it performs the best on the testing Concrete Slump Test data.

[35]:
```
# Calculate RMSE for each model
test_rmse_scores = []
test_predictions = []
for name, estimator in voting_regressor.named_estimators_.items():
    y_pred = estimator.predict(X_test_scaled)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    test_rmse_scores.append((name, rmse))
```

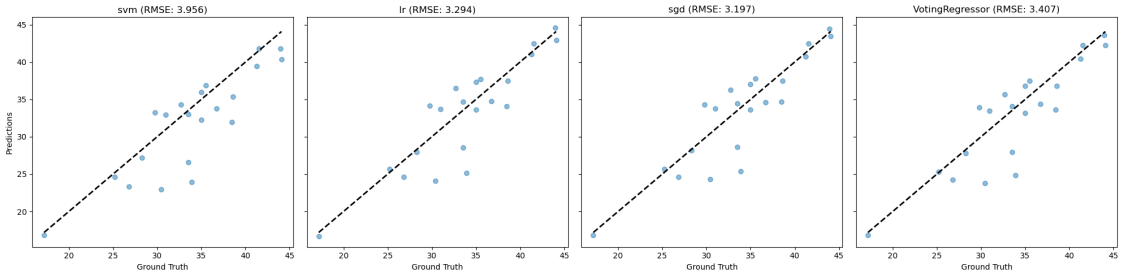
```
    test_predictions.append((name, y_pred))

# Create and train the VotingRegressor
voting_regressor = VotingRegressor(estimators)
voting_regressor.fit(X_train_scaled, y_train)
y_pred = voting_regressor.predict(X_test_scaled)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
test_rmse_scores.append(('VotingRegressor', rmse))
test_predictions.append(('VotingRegressor', y_pred))  # add voting regressor␣
 ↪predictions

# Print RMSE scores
for name, rmse in test_rmse_scores:
    print(f'{name}: {rmse:.3f}')
```

```
svm: 3.956
lr: 3.294
sgd: 3.197
VotingRegressor: 3.407
```

1.3.4 Visualising plots

The plots show the ground truth values and the predictions. The difference between the predictions and the ground truth values is much greater for the testing set demonstrating a lower prediction accuracy. This is a clear sign of overfitting when comparing to the more precise predictions on the graphs for the trainind data. RMSE values greater then 3 are not ideal and this is supported with the poor performing predictions represented on the graph.

[36]:
```
# Create a scatter plot for each model
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(20, 5), sharey=True)
for i, (name, y_pred) in enumerate(test_predictions):
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    axes[i].scatter(y_test, y_pred, alpha=0.5)
    axes[i].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
 ↪'k--', lw=2)
    axes[i].set_title(f'{name} (RMSE: {rmse:.3f})')
    axes[i].set_xlabel('Ground Truth')
    if i == 0:
        axes[i].set_ylabel('Predictions')

plt.tight_layout()
plt.show()
```



1.4 2.3 Tuning Hyperparameters

Using the code below SVM's best hyperparameters were: - C = 5 - epsilon: 0.1

SGD's best hyperparameters were: - alpha = 0.001 - eta0 = 0.1 - learning_rate = constant - penalty = l2

Linear regressor does not have hyperparameters.

After implementing these hyperparameters the results were:

(tuned): means with hyperparameters

SVM(tuned) = 3.514, SVM = 3.953 SGD(tuned) = 3.102, SGD = 3.193

Note these values can change when running they were just the results I got when I ran.

There doesn't appear to be significant improvement with SGD but there does with SVM, meaning SGD's hyperparameters must have already been close to ideal.

[37]:
```
# Hyperparameter tuning with GridSearchCV
param_grids = [
    {'C': [1, 2, 3, 4, 5], 'epsilon': [0.1, 0.5, 1]},
    {}, # No hyperparameters to tune for Linear Regression
    {'alpha': [0.0001, 0.001, 0.01], 'penalty': ['l1', 'l2', 'elasticnet'],␣
 ↪'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive'], 'eta0':␣
 ↪[0.01, 0.1, 0.5]}
]

for i, (name, estimator) in enumerate(estimators):
    if not param_grids[i]:
        continue

    grid_search = GridSearchCV(estimator, param_grid=param_grids[i],␣
 ↪scoring='neg_mean_squared_error', cv=3)
    grid_search.fit(X_train_scaled, y_train)
    print(f'Best parameters for {name}: {grid_search.best_params_}')

    # Update the estimator with the best parameters
    estimators[i] = (name, grid_search.best_estimator_)
```

```python
# Train the VotingRegressor with tuned hyperparameters
voting_regressor_tuned = VotingRegressor(estimators)
voting_regressor_tuned.fit(X_train_scaled, y_train)

# Calculate new RMSE scores
rmse_scores_tuned = []
for name, estimator in voting_regressor_tuned.named_estimators_.items():
    y_pred = estimator.predict(X_test_scaled)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    rmse_scores_tuned.append((name, rmse))

# Print new RMSE scores
for name, rmse in rmse_scores_tuned:
    print(f'{name} (tuned): {rmse:.3f}')
```

```
Best parameters for svm: {'C': 5, 'epsilon': 0.1}
Best parameters for sgd: {'alpha': 0.001, 'eta0': 0.1, 'learning_rate':
'invscaling', 'penalty': 'l1'}
svm (tuned): 3.522
lr (tuned): 3.294
sgd (tuned): 3.235
```

```
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
```

```
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
```

```
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/Users/lachlanbassi/Applications/envs/cits5508-2023/lib/python3.10/site-
packages/sklearn/linear_model/_stochastic_gradient.py:1549: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
```

2 3 Abalone Dataset

First the contents are read in, sex is converted to a numeric value and the number of null values are counted to ensure they aren't present.

[38]:
```
data_file_path = "abalone.data"

delimiter = ','

column_names = ["Sex", "Length", "Diameter", "Height", "Whole weight", "Shucked␣
↪weight", "Viscera weight", "Shell weight", "Rings"]

data = pd.read_csv(data_file_path, delimiter=delimiter, header = None, names =␣
↪column_names)

# Create a LabelEncoder object
encoder = LabelEncoder()

# Fit the encoder on the "Sex" column
```

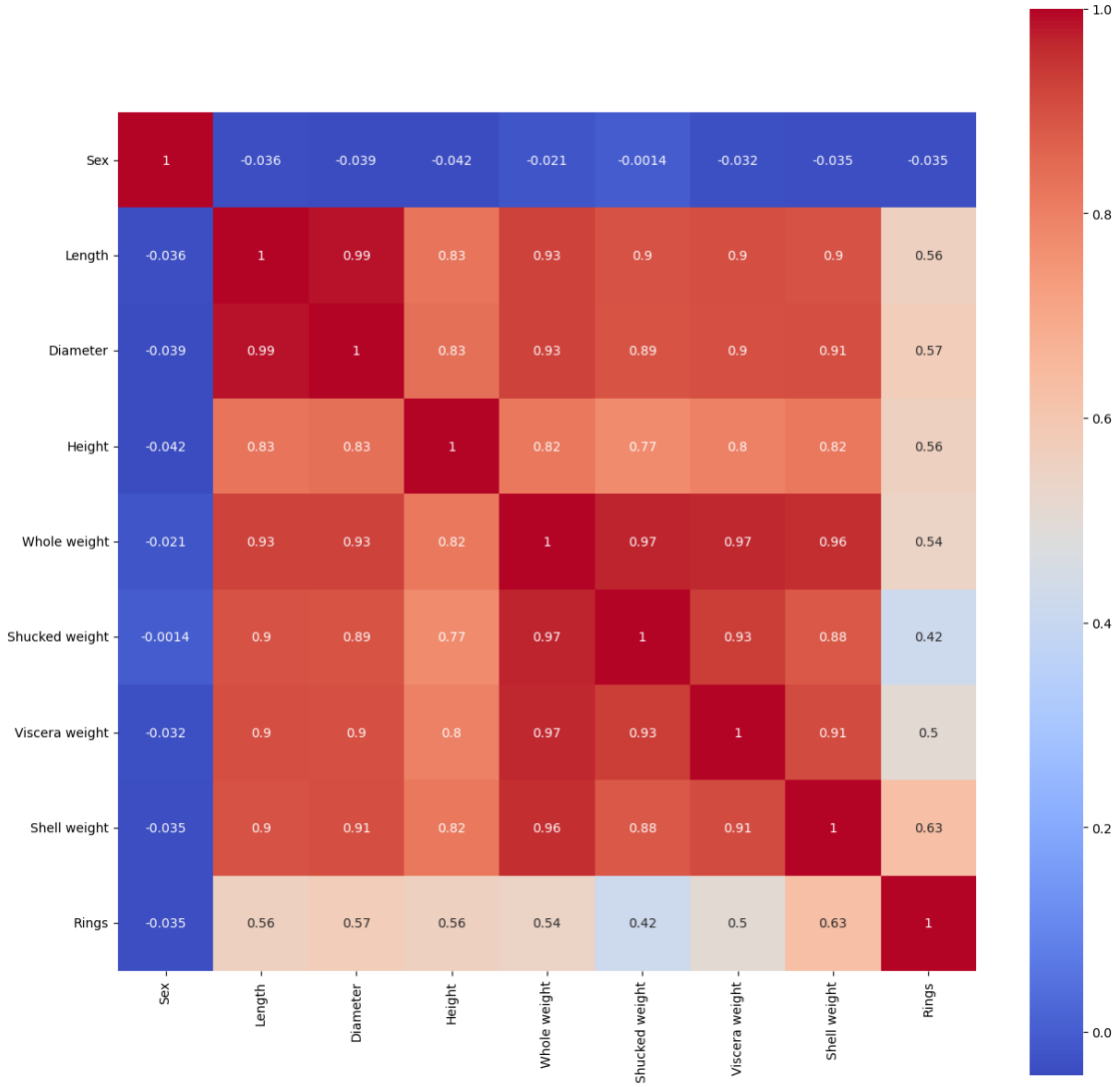
```
encoder.fit(data["Sex"])

# Apply the encoding to the "Sex" column
data["Sex"] = encoder.transform(data["Sex"])

nan_count = data.isna().sum().sum()
print(nan_count)
```

```
0
```

2.1 3.1 Creating Correlation Matrix

A correlation matrix is made, there appears to be highly correlated data so in order to remove it a threshold value is set to 0.95 meaning variables correlated greater then 0.95 will be removed.

[39]:
```
# Compute the correlation matrix
corr_matrix = data.corr()

# Plot the heatmap
plt.figure(figsize=(15, 15))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", square=True)
plt.show()
```



2.2 Remaining variables

The remaining variables are:

- Sex
- Diameter
- Height
- Shucked Weight
- Viscera Weight
- Shell Weight
- Rings

and the dropped variables are:

- length

- whole weight

[40]:
```
threshold = 0.95
to_drop = []

for i in range(corr_matrix.shape[0]):
    for j in range(0, i):
        if corr_matrix.iloc[i, j] >= threshold:
            to_drop.append(corr_matrix.columns[j])

# Get unique columns to drop
to_drop = list(set(to_drop))

# Drop features from the DataFrame
data = data.drop(to_drop, axis=1)
data.head()
```

[40]:

	Sex	Diameter	Height	Shucked weight	Viscera weight	Shell weight	Rings
0	2	0.365	0.095	0.2245	0.1010	0.150	15
1	2	0.265	0.090	0.0995	0.0485	0.070	7
2	0	0.420	0.135	0.2565	0.1415	0.210	9
3	2	0.365	0.125	0.2155	0.1140	0.155	10
4	1	0.255	0.080	0.0895	0.0395	0.055	7

2.2.1 Creating training and testing data

[41]:
```
# Update the feature matrix X
y = data["Rings"]
X = data.drop("Rings", axis=1).values

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,␣
 ↪random_state=123)

print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
Training set size: (3550, 6)
Test set size: (627, 6)
```

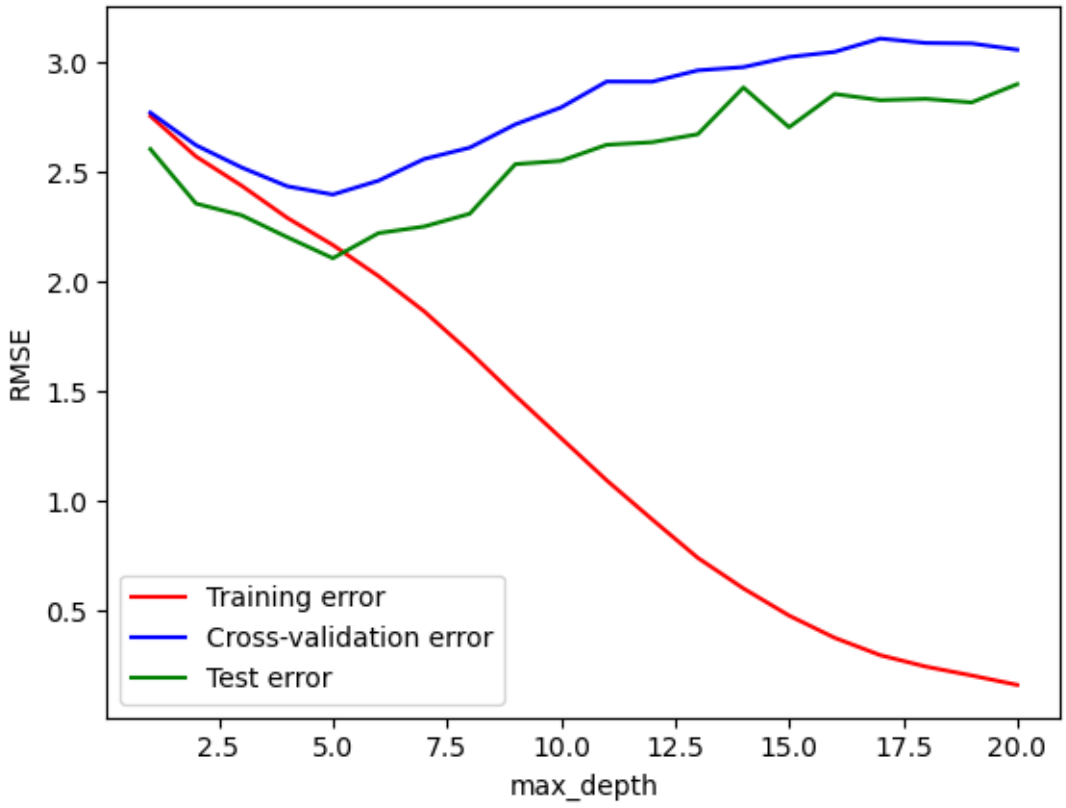
2.3 3.2 Creating Decision Tree Regressor

A decision tree regressor is created trained on the scaled data with a hyperparameter max_depth. The goal is to find the best max_depth.

The performance is evaluated by comparing the training error, cross-validation error, and test error.

Based on the plot the best max_depth value is 5.

[42]:
```
# Define max depth range to search over
max_depths = np.arange(1, 21)

# Train a Decision Tree Regression using GridSearchCV
tree_reg = DecisionTreeRegressor(random_state=123)
param_grid = {'max_depth': max_depths}
grid_search = GridSearchCV(tree_reg, param_grid, cv=3,␣
 ↪scoring='neg_root_mean_squared_error', return_train_score=True)
grid_search.fit(X_train_scaled, y_train)

# Extract the results
train_errors = -grid_search.cv_results_['mean_train_score']
cv_errors = -grid_search.cv_results_['mean_test_score']
test_errors = []

# Calculate test errors
for depth in max_depths:
    tree_reg = DecisionTreeRegressor(max_depth=depth, random_state=123)
    tree_reg.fit(X_train_scaled, y_train)
    y_pred = tree_reg.predict(X_test_scaled)
    test_error = mean_squared_error(y_test, y_pred, squared=False)
    test_errors.append(test_error)

# Plot the learning curves
plt.plot(max_depths, train_errors, 'r-', label='Training error')
plt.plot(max_depths, cv_errors, 'b-', label='Cross-validation error')
plt.plot(max_depths, test_errors, 'g-', label='Test error')
plt.xlabel('max_depth')
plt.ylabel('RMSE')
plt.legend()
plt.show()

# Find the best max_depth
best_max_depth = grid_search.best_params_['max_depth']
print(f"Best max_depth: {best_max_depth}")
```



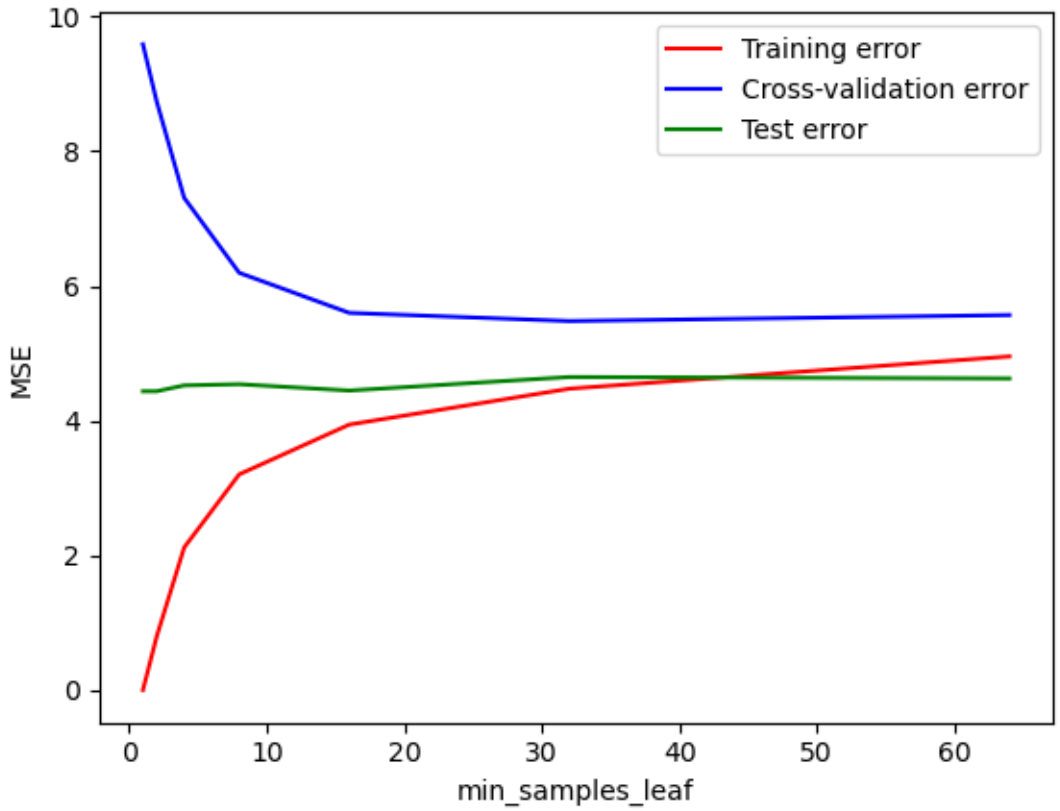

```
Best max_depth: 5
```

2.4 Creating decision tree for min_samples_leaf

Doing the same process but for min_samples_leaf the results conclude that 32 is the ideal size. The graph appears to plateau around 16 meaning higher values probably won't offer much improvement.

[43]:
```
# Define max depth range to search over
min_samples_leaf = [1, 2, 4, 8, 16, 32, 64]

# Train a Decision Tree Regression using GridSearchCV
tree_reg = DecisionTreeRegressor(random_state=123)
param_grid = {'min_samples_leaf': min_samples_leaf}
grid_search = GridSearchCV(tree_reg, param_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error', return_train_score=True)
grid_search.fit(X_train_scaled, y_train)

# Extract training errors and cross-validation errors
train_errors = -grid_search.cv_results_['mean_train_score']
cv_errors = -grid_search.cv_results_['mean_test_score']
test_errors = []
```

```
# Calculate test errors
for min_samples in min_samples_leaf:
    tree_reg = DecisionTreeRegressor(max_depth = 5,␣
 ↪min_samples_leaf=min_samples, random_state=123)
    tree_reg.fit(X_train_scaled, y_train)
    y_pred = tree_reg.predict(X_test_scaled)
    test_error = mean_squared_error(y_test, y_pred)
    test_errors.append(test_error)

# Plot the learning curves
plt.plot(min_samples_leaf, train_errors, 'r-', label='Training error')
plt.plot(min_samples_leaf, cv_errors, 'b-', label='Cross-validation error')
plt.plot(min_samples_leaf, test_errors, 'g-', label='Test error')
plt.xlabel('min_samples_leaf')
plt.ylabel('MSE')
plt.legend()
plt.show()

# Find the best max_depth
best_min_samples_leaf = grid_search.best_params_['min_samples_leaf']
print(f"Best min_samples_leaf: {best_min_samples_leaf}")
```



```
Best min_samples_leaf: 32
```

3 3.3 Random Forest Regressor

A random forest regressor was implemented with the following features:

n_estimators = 500 max_depth = 5 (previously found to be optimal) min_samples_leaf = 32 (previously found to be optimal)

The rmse for this regressor is 2.207

[44]:
```
# Create a Random Forest Regressor with 500 estimators
rf_regressor = RandomForestRegressor(n_estimators=500,␣
 ↪max_depth=best_k_max_depth, min_samples_leaf=best_min_samples_leaf,␣
 ↪random_state=123)

# Fit the model on the training data
rf_regressor.fit(X_train_scaled, y_train)

# Predict the test set results
y_pred = rf_regressor.predict(X_test_scaled)

# Round the predicted values to the nearest integer
y_pred_rounded = np.round(y_pred).astype(int)

# Calculate the RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred_rounded))
print(f"RMSE for the Random Forest Regressor: {rmse}")
```

```
RMSE for the Random Forest Regressor: 2.206630576421311
```

Experimenting with other hyperparameters: max samples, max features, bootstrap. The best values were determined to be

Best max samples = 0.2 Best max features = 5 Best bootstrap = True RMSE with best hyperparameters = 2.1712930609705405

[ ]:
```
# Define values to test for each hyperparameter
max_samples_values =  [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
max_features_values = list(range(1, 22))
bootstrap_values = [True]

# Initialize variables to store best hyperparameters and performance
best_k_max_samples = None
best_k_max_features = None
best_k_bootstrap = None
best_rmse = float('inf')
```

```
# Perform manual testing over hyperparameters
for max_samples in max_samples_values:
    for max_features in max_features_values:
        for bootstrap in bootstrap_values:
            # Train and evaluate model with current hyperparameters
            rf_regressor = RandomForestRegressor(n_estimators=500,␣
 ↪max_depth=best_k_max_depth, min_samples_leaf=best_min_samples_leaf,␣
 ↪max_samples=max_samples, max_features=max_features, bootstrap=bootstrap,␣
 ↪random_state=123)
            rf_regressor.fit(X_train_scaled, y_train)
            y_pred = rf_regressor.predict(X_test_scaled)
            y_pred_rounded = np.round(y_pred).astype(int)
            rmse = np.sqrt(mean_squared_error(y_test, y_pred_rounded))

            # Update best hyperparameters and performance if current␣
 ↪hyperparameters yield better performance
            if rmse < best_rmse:
                best_k_max_samples = max_samples
                best_k_max_features = max_features
                best_k_bootstrap = bootstrap
                best_rmse = rmse


best_rf_regressor = RandomForestRegressor(n_estimators=500,␣
 ↪max_depth=best_k_max_depth, min_samples_leaf=best_min_samples_leaf,␣
 ↪max_samples=best_k_max_samples, max_features=best_k_max_features,␣
 ↪bootstrap=best_k_bootstrap, random_state=123)
best_rf_regressor.fit(X_train_scaled, y_train)
y_pred = best_rf_regressor.predict(X_test_scaled)
y_pred_rounded = np.round(y_pred).astype(int)
rmse = np.sqrt(mean_squared_error(y_test, y_pred_rounded))
print(f"Best max samples = {best_k_max_samples}")
print(f"Best max features = {best_k_max_features}")
print(f"Best bootstrap = {best_k_bootstrap}")
print(f"RMSE with best hyperparameters = {rmse}")
```

3.1 3.4 Reducing Random Forest regressor

Dropping the features that have an importance less than 0.05 the result is removing sex and diameter. What's interesting is that setting the threshold to 0.1 would leave only feature left, shell_weight that is extremely important with a value of 0.73.

[ ]:
```
features = data.drop(columns=["Rings"])
# Train the Random Forest Regressor with the best hyperparameters
best_rf_regressor.fit(X_train_scaled, y_train)

# Obtain the feature importances
```

```
feature_importances = best_rf_regressor.feature_importances_
print("Feature importances:", feature_importances)

# Set the threshold for feature selection
threshold = 0.05

# Create a boolean mask for features above the threshold
mask = feature_importances > threshold

# Print retained and removed features
print("Retained features:", features.columns[mask])
print("Removed features:", features.columns[~mask])

# Calculate the total feature importance value retained
total_importance = np.sum(feature_importances[mask])
print("Total feature importance value retained:", total_importance)

# Trim the feature dimension using SelectFromModel
sfm = SelectFromModel(best_rf_regressor, threshold=threshold)
sfm.fit(X_train_scaled, y_train)

# Transform the training and test sets
X_train_reduced = sfm.transform(X_train_scaled)
X_test_reduced = sfm.transform(X_test_scaled)
```

3.2 Repeat Training

The training is repeated and the RMSE is calculated, a slight increase is observed: initial model = 2.1712930609705405, and reduced = 2.1782599906626965. This is expected as a very small percentage of the prediction accuracy is made up of the two removed variables, however keeping those variables makes the model more complex and isn't worth it keeping due to the little performance boost they provide especially since the difference is roughly 0.007 based on these results.

[ ]:
```
# Create a Random Forest Regressor with 500 estimators
rf_regressor = RandomForestRegressor(n_estimators=500,␣
 ↪max_depth=best_k_max_depth, min_samples_leaf=best_min_samples_leaf,␣
 ↪max_samples = best_k_max_samples, max_features = best_k_max_features,␣
 ↪bootstrap = True, random_state=123)

# Fit the model on the training data
rf_regressor.fit(X_train_reduced, y_train)

# Predict the test set results
y_pred = rf_regressor.predict(X_test_reduced)

# Round the predicted values to the nearest integer
y_pred_reduced_rounded = np.round(y_pred).astype(int)
```

```
# Calculate the RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred_reduced_rounded))
print(f"RMSE for the Random Forest Regressor: {rmse}")
```

3.3 Compare two Random Forest Regressors

The two graphs appear quite similar which is expected from the small RMSE differences, there appears to be more variation as Ring size increases.

[ ]:
```
# Calculate the prediction errors for the original and reduced-dimensional data
errors_original = np.abs(y_test - y_pred_rounded)
errors_reduced = np.abs(y_test - y_pred_reduced_rounded)

# Create a DataFrame with the true ring values and the prediction errors
error_data = pd.DataFrame({"Ring Value": y_test, "Error Original":␣
 ↪errors_original, "Error Reduced": errors_reduced})

# Calculate the average prediction error for each ring value
average_errors = error_data.groupby("Ring Value").mean()

# Plot the average prediction errors
plt.figure(figsize=(12, 6))
plt.plot(average_errors.index, average_errors["Error Original"],␣
 ↪label="Original Data")
plt.plot(average_errors.index, average_errors["Error Reduced"],␣
 ↪label="Reduced-Dimensional Data", linestyle="--")
plt.xlabel("Ring Value")
plt.ylabel("Average Prediction Error")
plt.legend()
plt.title("Average Prediction Error for Each Ring Value")
plt.show()
```

3.4 Bagging Regressor

Implement a Bagging regressor with 500 SVM regressors as the base estimators. Again, choose some reasonable hyperparameter values manually for the SVM regressors. For the Bagging regressor, you should use the same common hyperparameter values (e.g., max features, max samples, bootstrap, etc) as your Random Forest regressor. Train your Bagging regressor using the full-dimensional training set and test it using the full-dimensional test set. Report the RMSE of the predictions for the test set. Illustrate in a diagram the predicted ring values versus the ground truth ring values of all the test instances.

[ ]:
```
svm_regressor = SVR(kernel='rbf', C=1, gamma='scale')

bagging_regressor = BaggingRegressor(estimator=svm_regressor, n_estimators=500,␣
 ↪max_features = best_k_max_features,
                                     max_samples=best_k_max_samples,
```

```
                                        bootstrap=best_k_bootstrap,␣
 ↪random_state=123)

bagging_regressor.fit(X_train_scaled, y_train)

y_pred = bagging_regressor.predict(X_test_scaled)
y_pred_rounded = np.round(y_pred).astype(int)
rmse_bagging = np.sqrt(mean_squared_error(y_test, y_pred_rounded))
print(f"RMSE for Bagging regressor with SVM base estimator: {rmse_bagging}")
```

[ ]:
```
plt.scatter(y_test, y_pred)
plt.xlabel("True Ring Values")
plt.ylabel("Predicted Ring Values")
plt.title("True vs. Predicted Ring Values for Bagging Regressor with SVM Base␣
 ↪Estimators")
plt.show()
```

3.5 Comparison

Compare the performance of your first Random Forest regressor with the Bagging regressor.

Baes on the analysis the bagging regressor outperforms the random forest only slightly. The difference in the values is roughly 0.02.

[ ]:
```
print(f"RMSE for Random Forest regressor: {rmse}")  # This value was calculated␣
 ↪earlier
print(f"RMSE for Bagging regressor with SVM base estimators: {rmse_bagging}")
```