Data Science III with python (Class notes)

STAT 303-3

Lizhen Shi

2025-04-02

Table of contents

Preface

This book serves as the course notes for STAT 303 Sec20, Spring 2025 at Northwestern University. To enhance your understanding of the material, you are expected to read the textbook before using these notes.

It is an evolving resource designed to support the course's learning objectives. This edition builds upon the foundational work of Professor Arvind Krishna, whose contributions have provided a strong framework for this resource. We are deeply grateful for his efforts, which continue to shape the book's development.

Throughout the quarter, the content will be updated and refined in real time to enhance clarity, depth, and relevance. These modifications ensure alignment with current teaching objectives and methodologies.

As a living document, this book welcomes feedback, suggestions, and contributions from students, instructors, and the broader academic community. Your input helps improve its quality and effectiveness.

Thank you for being part of this journey—we hope this resource serves as a valuable guide in your learning.

Part I

Bias & Variance; KNN

1 Bias-variance tradeoff

Read section 2.2.2 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

In this chapter, we will show that a flexible model is likely to have high variance and low bias, while a relatively less flexible model is likely to have a high bias and low variance.

The examples considered below are motivated from the examples shown in the documentation of the bias_variance_decomp() function from the `mlxtend` library. We will first manually compute the bias and variance for understanding of the concept. Later, we will show application of the `bias_variance_decomp()` function to estimate bias and variance.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
sns.set(font_scale=1.35)
```

1.1 Simple model (Less flexible)

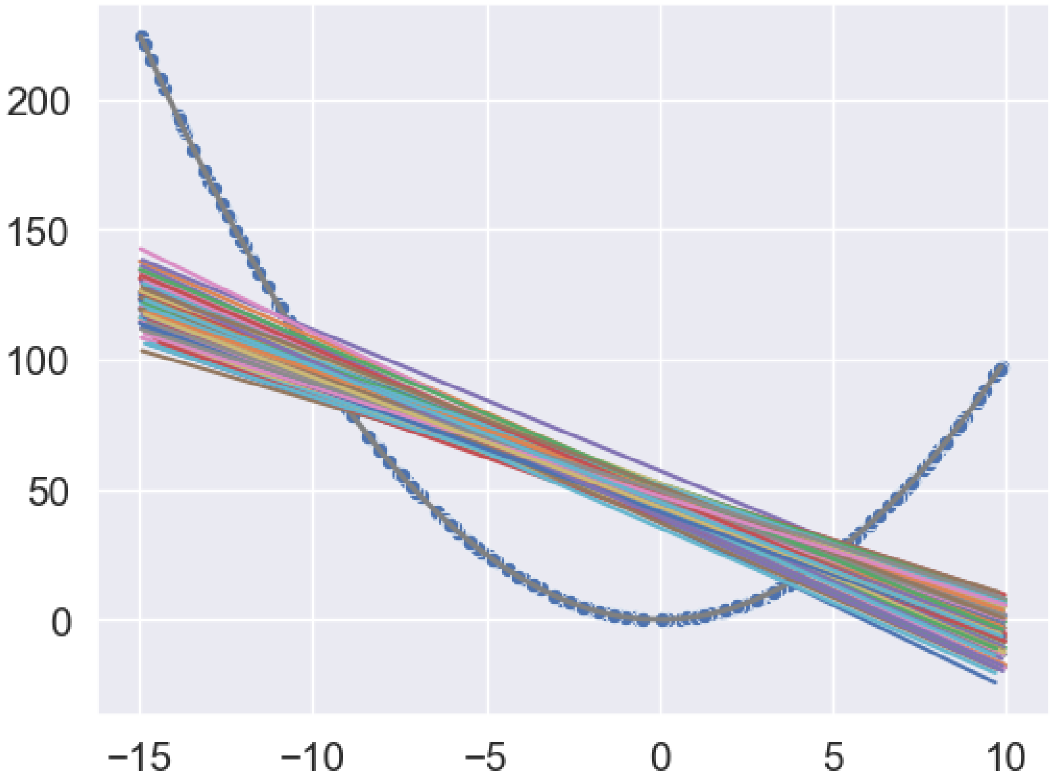
Let us consider a linear regression model as the less-flexible *(or relatively simple)* model.

We will first simulate the test dataset for which we will compute the bias and variance.

```
np.random.seed(101)

# Simulating predictor values of test data
xtest = np.random.uniform(-15, 10, 200)
```

```
# Assuming the true mean response is square of the predictor value
fxtest = xtest**2

# Simulating noiseless test response
ytest = fxtest

# We will find bias and variance using a linear regression model for prediction
model = LinearRegression()
```

```
# Visualizing the data and the true mean response
sns.scatterplot(x = xtest, y = ytest)
sns.lineplot(x = xtest, y = fxtest, color = 'grey', linewidth = 2)

# Initializing objects to store predictions and mean squared error
# of 100 models developed on 100 distinct training datasets samples
pred_test = []; mse_test = []

# Iterating over each of the 100 models
for i in range(100):
    np.random.seed(i)

    # Simulating the ith training data
    x = np.random.uniform(-15, 10, 200)
    fx = x**2
    y = fx + np.random.normal(0, 10, 200)

    # Fitting the ith model on the ith training data
    model.fit(x.reshape(-1,1), y)

    # Plotting the ith model
    sns.lineplot(x = x, y = model.predict(x.reshape(-1,1)))

    # Storing the predictions of the ith model on test data
    pred_test.append(model.predict(xtest.reshape(-1,1)))

    # Storing the mean squared error of the ith model on test data
    mse_test.append(mean_squared_error(model.predict(xtest.reshape(-1,1)), ytest))
```



The above plots show that the 100 models seem to have low variance, but high bias. Note that the bias is low only around a couple of points *(x = -10 & x = 5)*.

Let us compute the average squared bias over all the test data points.

```
mean_pred = np.array(pred_test).mean(axis = 0)
sq_bias = ((mean_pred - fxtest)**2).mean()
sq_bias
```

```
2042.104126728109
```

Let us compute the average variance over all the test data points.

```
mean_var = np.array(pred_test).var(axis = 0).mean()
mean_var
```

```
28.37397844429763
```

Let us compute the mean squared error over all the test data points.

```
np.array(mse_test).mean()
```

```
2070.4781051724062
```

Note that the mean squared error should be the same as the sum of squared bias and variance

The sum of squared bias and model variance is:

```
sq_bias + mean_var
```

```
2070.4781051724067
```

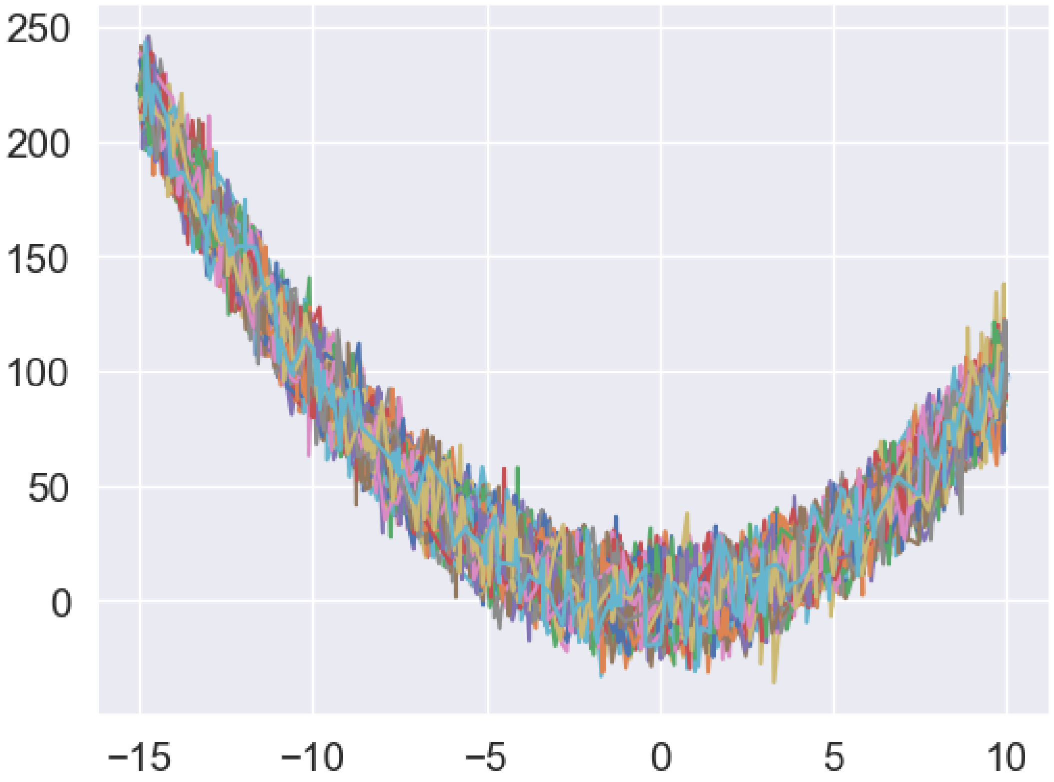
Note that this is exactly the same as the mean squared error computed above as we are developing a finite number of models, and making predictions on a finite number of test data points.

1.2 Complex model (more flexible)

Let us consider a decion tree as the more flexible model.

```
np.random.seed(101)
xtest = np.random.uniform(-15, 10, 200)
fxtest = xtest**2
ytest = fxtest
model = DecisionTreeRegressor()
```

```
sns.scatterplot(x = xtest, y = ytest)
sns.lineplot(x = xtest, y = fxtest, color = 'grey', linewidth = 2)
pred_test = []; mse_test = []
for i in range(100):
    np.random.seed(i)
    x = np.random.uniform(-15, 10, 200)
    fx = x**2
    y = fx + np.random.normal(0, 10, 200)
    model.fit(x.reshape(-1,1), y)
    sns.lineplot(x = x, y = model.predict(x.reshape(-1,1)))
    pred_test.append(model.predict(xtest.reshape(-1,1)))
    mse_test.append(mean_squared_error(model.predict(xtest.reshape(-1,1)), ytest))
```



The above plots show that the 100 models seem to have high variance, but low bias.

Let us compute the average squared bias over all the test data points.

```
mean_pred = np.array(pred_test).mean(axis = 0)
sq_bias = ((mean_pred - fxtest)**2).mean()
sq_bias
```

```
1.3117561629333938
```

Let us compute the average model variance over all the test data points.

```
mean_var = np.array(pred_test).var(axis = 0).mean()
mean_var
```

```
102.5226748977198
```

Let us compute the average mean squared error over all the test data points.

```
np.array(mse_test).mean()
```

```
103.83443106065317
```

Note that the above error is still the same as the sum of the squared bias, model variance and the irreducible error.

Note that the relatively more flexible model has a higher variance, but lower bias as compared to the less flexible linear model. This will typically be the case, but may not be true in all scenarios. We will discuss one such scenario later.

2 KNN

Read section 4.7.6 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.35)

from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV, cross_val_predict, KFold,

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer
from sklearn.metrics import root_mean_squared_error, r2_score
```

2.1 KNN for regression

```
# Load the dataset
car = pd.read_csv('Datasets/car.csv')

# Split the dataset into features and target variable
X = car.drop(columns=['price'])
y = car['price']

# split the dataset into training and testing sets
```

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```python
# extract the categorical columns and put them in a list
cat_cols = X.select_dtypes(include=['object']).columns.tolist()

# extract the numerical columns and put them in a list
num_cols = X.select_dtypes(include=['int64', 'float64']).columns.tolist()
```

```python
# First transform categorical variables
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', num_cols),  # Just pass numerical features through
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols)
    ])

# Create pipeline that scales all features together
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('scaler', StandardScaler()),  # Scale everything together
    ('knn', KNeighborsRegressor(n_neighbors=5))
])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)
# Predict on the test data
y_pred = pipeline.predict(X_test)
# Calculate RMSE
rmse = root_mean_squared_error(y_test, y_pred)
print(f"RMSE: {rmse:.2f}")
print(f"R² Score: {pipeline.score(X_test, y_test):.2f}")
```

```
RMSE: 4364.84
R² Score: 0.94
```

```python
# show the features in the numerical transformer
pipeline.named_steps['preprocessor'].transformers_[0][1].get_feature_names_out()
print("numerical features in the pipeline:", pipeline.named_steps['preprocessor'].transformer

# show the features in the categorical transformer
pipeline.named_steps['preprocessor'].transformers_[1][1].get_feature_names_out()
print("categorical features in the pipeline:", pipeline.named_steps['preprocessor'].transform
numerical features in the pipeline: ['year' 'mileage' 'tax' 'mpg' 'engineSize']
categorical features in the pipeline: ['brand_audi' 'brand_bmw' 'brand_ford' 'brand_hyundi'
 'brand_skoda' 'brand_toyota' 'brand_vauxhall' 'brand_vw'
 'model_ 6 Series' 'model_ 7 Series' 'model_ 8 Series' 'model_ A7'
 'model_ A8' 'model_ Agila' 'model_ Amarok' 'model_ Antara'
 'model_ Arteon' 'model_ Avensis' 'model_ Beetle' 'model_ CC'
 'model_ CLA Class' 'model_ CLK' 'model_ CLS Class' 'model_ Caddy'
 'model_ Caddy Life' 'model_ Caddy Maxi Life' 'model_ California'
 'model_ Camry' 'model_ Caravelle' 'model_ Combo Life' 'model_ Edge'
 'model_ Eos' 'model_ Fusion' 'model_ G Class' 'model_ GL Class'
 'model_ GLB Class' 'model_ GLS Class' 'model_ GT86' 'model_ GTC'
 'model_ Galaxy' 'model_ Getz' 'model_ Grand C-MAX'
 'model_ Grand Tourneo Connect' 'model_ Hilux' 'model_ I40' 'model_ I800'
 'model_ IQ' 'model_ IX20' 'model_ IX35' 'model_ Jetta' 'model_ KA'
 'model_ Kamiq' 'model_ Land Cruiser' 'model_ M Class' 'model_ M2'
 'model_ M3' 'model_ M4' 'model_ M5' 'model_ M6' 'model_ Mustang'
 'model_ PROACE VERSO' 'model_ Prius' 'model_ Puma' 'model_ Q8'
 'model_ R8' 'model_ RS3' 'model_ RS4' 'model_ RS5' 'model_ RS6'
 'model_ Rapid' 'model_ Roomster' 'model_ S Class' 'model_ S3' 'model_ S4'
 'model_ SLK' 'model_ SQ5' 'model_ SQ7' 'model_ Santa Fe' 'model_ Scala'
 'model_ Scirocco' 'model_ Shuttle' 'model_ Supra'
 'model_ Tiguan Allspace' 'model_ Tourneo Connect' 'model_ Tourneo Custom'
 'model_ V Class' 'model_ Verso' 'model_ Vivaro' 'model_ X-CLASS'
 'model_ X4' 'model_ X6' 'model_ X7' 'model_ Yeti' 'model_ Z3' 'model_ Z4'
 'model_ Zafira Tourer' 'model_ i3' 'model_ i8' 'transmission_Automatic'
 'transmission_Manual' 'transmission_Other' 'transmission_Semi-Auto'
 'fuelType_Diesel' 'fuelType_Electric' 'fuelType_Hybrid' 'fuelType_Other'
 'fuelType_Petrol']
```

2.2 Feature Scaling in KNN

Feature scaling is essential when using K-Nearest Neighbors (KNN) because the algorithm relies on calculating distances between data points. If features are measured on different scales (e.g., `mileage` in thousands and `mpg` in tens), the features with larger numeric ranges can dominate the distance calculations and distort the results.

To ensure that all features contribute equally, it’s important to **standardize or normalize** them before applying KNN. Common scaling techniques include:

- **Standardization** (zero mean, unit variance) using `StandardScaler`

- **Min-max scaling** to bring values into the `[0, 1]` range

Without scaling, KNN may produce biased or misleading predictions.
The example below illustrates how the same KNN model performs **without feature scaling**, highlighting the importance of preprocessing your data.

```
preprocessor_no_scaling = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', num_cols),  # Pass numerical features through without scaling
        ('cat', OneHotEncoder(handle_unknown='ignore'), cat_cols)  # Only one-hot encode cate
    ])

# Create pipeline without any scaling
pipeline_no_scaling = Pipeline(steps=[
    ('preprocessor', preprocessor_no_scaling),
    ('knn', KNeighborsRegressor(n_neighbors=5))
])

# Fit the pipeline
pipeline_no_scaling.fit(X_train, y_train)

# Evaluate
y_pred_no_scaling = pipeline_no_scaling.predict(X_test)

rmse_no_scaling = root_mean_squared_error(y_test, y_pred_no_scaling)
print(f"RMSE without scaling: {rmse_no_scaling:.2f}")
print(f"R² Score without scaling: {pipeline_no_scaling.score(X_test, y_test):.2f}")
```

```
RMSE without scaling: 13758.38
R² Score without scaling: 0.35
```

2.3 Hyperparameters in KNN

The most important hyperparameter in K-Nearest Neighbors (KNN) is ***k***, which determines the number of neighbors considered when making predictions. Tuning k helps balance the model's **bias and variance**:

- A **small *k*** (e.g., 1 or 3) can lead to **low bias but high variance**, making the model sensitive to noise in the training data.
- A **large *k*** results in **higher bias but lower variance**, producing smoother predictions that may underfit the data.

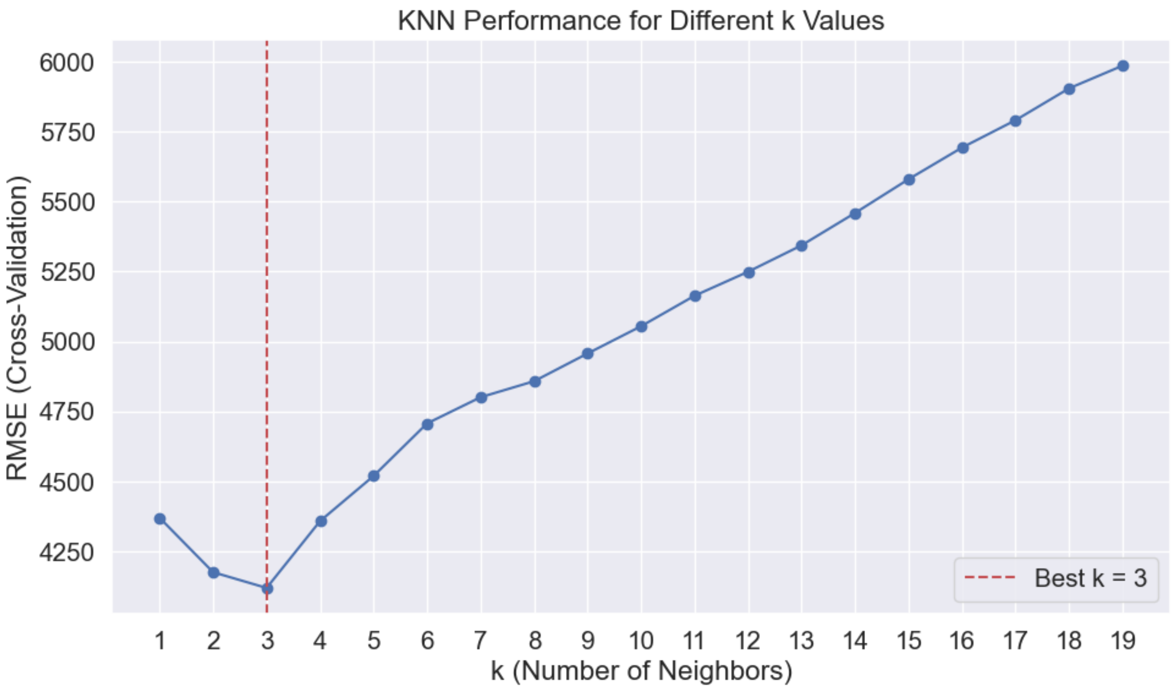
2.3.1 Tuning *k* in KNN

To find the optimal value of k, it's common to use **cross-validation**, which evaluates model performance on different subsets of the data. A popular tool for this is **`GridSearchCV`**, which automates the search process by testing multiple values of k using cross-validation behind the scenes. It selects the value of k that minimizes prediction error on unseen data—helping you achieve a good balance between underfitting and overfitting.

```
# Create parameter grid for k values
param_grid = {
    'knn__n_neighbors': list(range(1, 20))  # Test k values from 1 to 20
}

# Set up GridSearchCV
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='neg_root_mean_squared_error',  # Optimize for RMSE
    n_jobs=-1,  # Use all available cores
    verbose=1
)

# Fit grid search
print("Tuning k parameter...")
grid_search.fit(X_train, y_train)

# Get best parameters and results
best_k = grid_search.best_params_['knn__n_neighbors']
best_score = -grid_search.best_score_  # Convert back from negative RMSE

print(f"Best k: {best_k}")
print(f"Best CV RMSE: {best_score:.2f}")

# Evaluate on test set using best model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
test_rmse = root_mean_squared_error(y_test, y_pred)
test_r2 = r2_score(y_test, y_pred)

print(f"Test RMSE with k={best_k}: {test_rmse:.2f}")
print(f"Test R² Score with k={best_k}: {test_r2:.2f}")
```

```
Tuning k parameter...
Fitting 5 folds for each of 19 candidates, totalling 95 fits
Best k: 3
Best CV RMSE: 4117.42
Test RMSE with k=3: 4051.06
Test R² Score with k=3: 0.94
```

```python
# Plot performance across different k values
cv_results = grid_search.cv_results_
k_values = param_grid['knn__n_neighbors']
mean_rmse = -cv_results['mean_test_score']

plt.figure(figsize=(10, 6))
plt.plot(k_values, mean_rmse, marker='o')
plt.xlabel('k (Number of Neighbors)')
plt.ylabel('RMSE (Cross-Validation)')
plt.title('KNN Performance for Different k Values')
plt.grid(True)
plt.xticks(k_values)
plt.axvline(x=best_k, color='r', linestyle='--', label=f'Best k = {best_k}')
plt.legend()
plt.tight_layout()
plt.show()
```



The distances and the indices of the nearest K observations to each test observation can be obtained using the `kneighbors()` method.

```
# Get the KNN estimator from the pipeline
knn_estimator = best_model.named_steps['knn']

# Get indices of K-nearest neighbors for each test observation
neighbor_indices = knn_estimator.kneighbors(best_model.named_steps['preprocessor'].transform
                                            return_distance=False)
# neighbor_indices will contain the indices of the K nearest neighbors for each test observa
# Note: The indices are relative to the training set, not the test set.
# To get the actual neighbor observations, you can use these indices to index into the train
# For example, to get the actual neighbor observations for the first test observation:
neighbors = X_train.iloc[neighbor_indices[0]]
neighbors
```

	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize
4580	merc	V Class	2010	Automatic	259000	Diesel	540	30.8345	3.0
5651	merc	CLK	2003	Automatic	185000	Petrol	330	18.0803	4.3
3961	vw	Caravelle	2006	Manual	178000	Diesel	325	34.5738	2.5

2.3.2 Tuning Other KNN Hyperparameters

In addition to the number of neighbors (k), KNN has several other important hyperparameters that can significantly affect the model's performance. Fine-tuning these settings helps you get the most out of the algorithm. Key hyperparameters include:

- **`weights`**: Determines how the neighbors contribute to the prediction.
  - `'uniform'`: All neighbors are weighted equally (default).
  - `'distance'`: Closer neighbors have more influence.
  - Choosing `'distance'` can improve performance, especially when data points are unevenly distributed.
- **`metric`**: Defines the distance function used to measure similarity between data points.
  - `'minkowski'` (default) is a general-purpose metric that includes both Euclidean and Manhattan distances.
  - Other options include `'euclidean'`, `'manhattan'`, or even custom distance functions.
- **`p`**: Used when `metric='minkowski'`.
  - `p=2` gives **Euclidean distance** (standard for continuous features).
  - `p=1` gives **Manhattan distance** (useful when features are sparse or grid-based).
- **`algorithm`**: Controls the method used to compute nearest neighbors.
  - `'auto'`, `'ball_tree'`, `'kd_tree'`, or `'brute'`.
  - Most users can leave this as `'auto'`, which lets scikit-learn choose the best algorithm based on the data.

These hyperparameters can be tuned using `GridSearchCV` to find the combination that yields the best performance on validation data.

The model hyperparameters can be obtained using the `get_params()` method. Note that there are other hyperparameters to tune in addition to number of neighbors. However, the number of neighbours may be the most influential hyperparameter in most cases.

```
# Get the best model parameters
best_model.get_params()
```

```
{'memory': None,
 'steps': [('preprocessor',
   ColumnTransformer(transformers=[('num', 'passthrough',
                                    ['year', 'mileage', 'tax', 'mpg',
                                     'engineSize']),
                                   ('cat',
                                    OneHotEncoder(handle_unknown='ignore',
                                                  sparse_output=False),
                                    ['brand', 'model', 'transmission',
                                     'fuelType'])])),
  ('scaler', StandardScaler()),
  ('knn', KNeighborsRegressor(n_neighbors=3))],
 'transform_input': None,
 'verbose': False,
 'preprocessor': ColumnTransformer(transformers=[('num', 'passthrough',
                                  ['year', 'mileage', 'tax', 'mpg',
                                   'engineSize']),
                                 ('cat',
                                  OneHotEncoder(handle_unknown='ignore',
                                                sparse_output=False),
                                  ['brand', 'model', 'transmission',
                                   'fuelType'])]),
 'scaler': StandardScaler(),
 'knn': KNeighborsRegressor(n_neighbors=3),
 'preprocessor__force_int_remainder_cols': True,
 'preprocessor__n_jobs': None,
 'preprocessor__remainder': 'drop',
 'preprocessor__sparse_threshold': 0.3,
 'preprocessor__transformer_weights': None,
 'preprocessor__transformers': [('num',
   'passthrough',
   ['year', 'mileage', 'tax', 'mpg', 'engineSize']),
  ('cat',
   OneHotEncoder(handle_unknown='ignore', sparse_output=False),
   ['brand', 'model', 'transmission', 'fuelType'])],
 'preprocessor__verbose': False,
 'preprocessor__verbose_feature_names_out': True,
 'preprocessor__num': 'passthrough',
 'preprocessor__cat': OneHotEncoder(handle_unknown='ignore', sparse_output=False),
 'preprocessor__cat__categories': 'auto',
 'preprocessor__cat__drop': None,
 'preprocessor__cat__dtype': numpy.float64,
 'preprocessor__cat__feature_name_combiner': 'concat',
 'preprocessor__cat__handle_unknown': 'ignore',
 'preprocessor__cat__max_categories': None,
 'preprocessor__cat__min_frequency': None,
 'preprocessor__cat__sparse_output': False,
 'scaler__copy': True,
 'scaler__with_mean': True,
 'scaler__with_std': True,
 'knn__algorithm': 'auto',
 'knn__leaf_size': 30,
 'knn__metric': 'minkowski',
 'knn__metric_params': None,
 'knn__n_jobs': None,
 'knn__n_neighbors': 3,
 'knn__p': 2,
 'knn__weights': 'uniform'}
```

```python
# Extended parameter grid
param_grid = {
    'knn__n_neighbors': list(range(1, 20, 2)),  # Test odd k values from 1 to 19 (step=2 for
    'knn__weights': ['uniform', 'distance'],  # Uniform: equal weight; Distance: closer neig
    'knn__metric': ['euclidean', 'manhattan', 'minkowski'],  # Common distance metrics
    'knn__p': [1, 2]  # p=1 (Manhattan), p=2 (Euclidean) - only relevant for Minkowski
}

# Set up GridSearchCV
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='neg_root_mean_squared_error',  # Optimize for RMSE
    n_jobs=-1,  # Use all available cores
    verbose=1
)

# Fit grid search
print("Tuning KNN hyperparameters...")
grid_search.fit(X_train, y_train)

# Get best parameters and results
best_params = grid_search.best_params_
best_score = -grid_search.best_score_  # Convert negative RMSE to positive
```

```
# Display results
print("\nBest Parameters:")
for param, value in best_params.items():
    print(f"{param}: {value}")
print(f"Best CV RMSE: {best_score:.2f}")

# Evaluate on test set using best model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
test_rmse = root_mean_squared_error(y_test, y_pred)  # Calculate RMSE
print(f"Test RMSE: {test_rmse:.2f}")
```

```
Tuning KNN hyperparameters...
Fitting 5 folds for each of 120 candidates, totalling 600 fits

Best Parameters:
knn__metric: euclidean
knn__n_neighbors: 3
knn__p: 1
knn__weights: distance
Best CV RMSE: 4001.34
Test RMSE: 3826.94
```

The results for each cross-validation are stored in the `cv_results_` attribute.

```
pd.DataFrame(grid_search.cv_results_).head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_knn__metric	param_kn
0	0.033124	0.003042	0.127347	0.023598	euclidean	1
1	0.035615	0.010407	0.179835	0.013115	euclidean	1
2	0.027877	0.002536	0.148597	0.018612	euclidean	1
3	0.043631	0.016927	0.168392	0.027444	euclidean	1
4	0.043071	0.009615	0.184532	0.042681	euclidean	3

These results can be useful to see if other hyperparameter values are equally good.

```
pd.DataFrame(grid_search.cv_results_).sort_values(by = 'rank_test_score').head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_knn___metric	param_k
87	0.038193	0.010690	0.149261	0.050225	minkowski	3
5	0.047902	0.013181	0.185623	0.049865	euclidean	3
7	0.040595	0.005817	0.132290	0.009807	euclidean	3
51	0.034996	0.001900	0.744842	0.052065	manhattan	5
49	0.031465	0.004676	0.718503	0.057517	manhattan	5

The results show that the next two best hyperparameter values yield the same performance as the printed one

2.4 Hyperparameter Tuning

We used `GridSearchCV` to tune the hyperparameters of our KNN model above. Given a relatively simple set of hyperparameters and a limited number of combinations, this approach was sufficient to reduce the RMSE.

However, when the number of possible hyperparameter values grows large, `GridSearchCV` can become computationally expensive. In such cases, `RandomizedSearchCV` provides a more efficient alternative by sampling a fixed number of random combinations from the specified hyperparameter space. This makes it well-suited for scenarios with limited computational resources.

2.4.0.1 RandomizedSearchCV

```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
# Set up RandomizedSearchCV

# Define parameter distributions for randomized search
param_distributions = {
    'knn__n_neighbors': randint(1, 20),  # Random ints from 1 to 19
    'knn__weights': ['uniform', 'distance'],
    'knn__metric': ['euclidean', 'manhattan', 'minkowski'],
    'knn__p': [1, 2]  # Only relevant for Minkowski
}
```

```python
# Set up RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=pipeline,
    param_distributions=param_distributions,
    n_iter=30,  # Number of random combinations to try
    cv=5,
    scoring='neg_root_mean_squared_error',
    n_jobs=-1,
    random_state=42,
    verbose=1
)
```

```python
# Fit randomized search
print("Tuning KNN hyperparameters with RandomizedSearchCV...")
random_search.fit(X_train, y_train)

# Best results
best_params = random_search.best_params_
best_score = -random_search.best_score_

# Display results
print("\nBest Parameters (RandomizedSearchCV):")
for param, value in best_params.items():
    print(f"{param}: {value}")
print(f"Best CV RMSE: {best_score:.2f}")
```

```
Tuning KNN hyperparameters with RandomizedSearchCV...
Fitting 5 folds for each of 30 candidates, totalling 150 fits

Best Parameters (RandomizedSearchCV):
knn__metric: manhattan
knn__n_neighbors: 8
knn__p: 2
knn__weights: distance
Best CV RMSE: 4005.70
```

```python
# Evaluate on test set
best_model = random_search.best_estimator_
y_pred = best_model.predict(X_test)

# Calculate RMSE
```

```
test_rmse = root_mean_squared_error(y_test, y_pred)
print(f"Test RMSE: {test_rmse:.2f}")
```

```
Test RMSE: 3811.89
```

Why might `RandomizedSearchCV` outperform `GridSearchCV`?

Although `GridSearchCV` systematically evaluates all combinations of hyperparameter values from a predefined grid, it doesn't guarantee the best performance. In some cases, `RandomizedSearchCV` can actually perform better. Here's why:

- **Limited Grid Resolution**:
  `GridSearchCV` evaluates only the specific values you include in the grid. If the true optimal value lies between grid points, it may be missed entirely.
- **Broader Exploration**:
  `RandomizedSearchCV` samples from distributions (e.g., continuous or discrete ranges), allowing it to explore a wider range of hyperparameter values, including combinations not explicitly considered in a grid.

  In this case,

  - `list(range(1, 20, 2))` in `GridSearchCV`
  - But in `RandomizedSearchCV`, it samples from `randint(1, 20)`

  The best `n_neighbors` happens to be 11, only `RandomizedSearchCV` can find it unless you explicitly included it in your grid.
- **Efficiency in High Dimensions**:
  In high-dimensional search spaces, the number of combinations in a grid grows exponentially. `RandomizedSearchCV` remains efficient by sampling a fixed number of combinations, avoiding the "curse of dimensionality."
- **Better Use of Time Budget**:
  Given the same computational budget, `RandomizedSearchCV` may cover more diverse regions of the search space and stumble upon better-performing configurations.

In summary, `RandomizedSearchCV` is not only faster but can also lead to better models—especially when the hyperparameter space is large, continuous, or contains irrelevant parameters.

2.4.0.2 BayesSearchCV

In addition to these methods, **BayesSearchCV**, based on Bayesian optimization, provides a more intelligent approach to hyperparameter tuning. It models the performance landscape and selects hyperparameter combinations to evaluate based on past results, often requiring fewer evaluations to find optimal or near-optimal values. This makes `BayesSearchCV` a powerful option, especially when training models is costly.

```
# Step 1: Install scikit-optimize if not already installed
!pip install scikit-optimize
```

```
Collecting scikit-optimize
  Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl.metadata (9.7 kB)
Requirement already satisfied: joblib>=0.11 in c:\users\lsi8012\appdata\local\anaconda3\lib\s
Collecting pyaml>=16.9 (from scikit-optimize)
  Downloading pyaml-25.1.0-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: numpy>=1.20.3 in c:\users\lsi8012\appdata\local\anaconda3\lib\
Requirement already satisfied: scipy>=1.1.0 in c:\users\lsi8012\appdata\local\anaconda3\lib\s
Requirement already satisfied: scikit-learn>=1.0.0 in c:\users\lsi8012\appdata\local\anaconda
Requirement already satisfied: packaging>=21.3 in c:\users\lsi8012\appdata\roaming\python\pyt
Requirement already satisfied: PyYAML in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pa
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\lsi8012\appdata\local\anacond
Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl (107 kB)
   ---------------------------------------- 0.0/107.8 kB ? eta -:--:--
   ------------------------------------- -- 102.4/107.8 kB 5.8 MB/s eta 0:00:01
   ---------------------------------------- 107.8/107.8 kB 3.1 MB/s eta 0:00:00
Downloading pyaml-25.1.0-py3-none-any.whl (26 kB)
Installing collected packages: pyaml, scikit-optimize
Successfully installed pyaml-25.1.0 scikit-optimize-0.10.2
```

```
from skopt import BayesSearchCV
from skopt.space import Integer, Categorical
```

```
# Step 3: Define search space for Bayesian optimization
search_space = {
    'knn__n_neighbors': Integer(1, 19),  # Odd values will be sampled if needed
    'knn__weights': Categorical(['uniform', 'distance']),
    'knn__metric': Categorical(['euclidean', 'manhattan', 'minkowski']),
    'knn__p': Integer(1, 2)  # Used only when metric is minkowski
}
```

```python
# Step 4: Set up BayesSearchCV
bayes_search = BayesSearchCV(
    estimator=pipeline,
    search_spaces=search_space,
    n_iter=30,  # Number of different combinations to try
    scoring='neg_root_mean_squared_error',
    cv=5,
    n_jobs=-1,
    verbose=1,
    random_state=42
)
```

```python
# Step 5: Fit BayesSearchCV
print("Tuning KNN hyperparameters with Bayesian Optimization...")
bayes_search.fit(X_train, y_train)

# Get best parameters and best score
best_params = bayes_search.best_params_
best_score = -bayes_search.best_score_  # Convert negative RMSE to positive

# Display results
print("\nBest Parameters (Bayesian Optimization):")
for param, value in best_params.items():
    print(f"{param}: {value}")
print(f"Best CV RMSE: {best_score:.2f}")
```

```
Tuning KNN hyperparameters with Bayesian Optimization...
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(

Fitting 5 folds for each of 1 candidates, totalling 5 fits

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(
Fitting 5 folds for each of 1 candidates, totalling 5 fits
```

```
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\skopt\optimizer\optimizer.py:517:
  warnings.warn(
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits

Best Parameters (Bayesian Optimization):
knn__metric: manhattan
knn__n_neighbors: 8
knn__p: 2
knn__weights: distance
Best CV RMSE: 4005.70
```

```python
# Step 6: Evaluate on test set
best_model = bayes_search.best_estimator_
y_pred = best_model.predict(X_test)

# Calculate RMSE on test set
test_rmse = root_mean_squared_error(y_test, y_pred)
print(f"Test RMSE: {test_rmse:.2f}")
```

```
Test RMSE: 3811.89
```

3 Hyperparameter tuning

In this chapter we'll introduce several functions that help with tuning hyperparameters of a machine learning model.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score, cross_val_predict, \
cross_validate, GridSearchCV, RandomizedSearchCV, KFold, StratifiedKFold, RepeatedKFold, Repe
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, recall_score, mean_squared_error
from scipy.stats import uniform
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
import seaborn as sns
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import matplotlib.pyplot as plt
import warnings
from IPython import display
```

Let us read and pre-process data first. Then we'll be ready to tune the model hyperparameters. We'll use KNN as the model. Note that KNN has multiple hyperparameters to tune, such as number of neighbors, distance metric, weights of neighbours, etc.

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
predictors = ['mpg', 'engineSize', 'year', 'mileage']
X_train = train[predictors]
y_train = train['price']
X_test = test[predictors]
y_test = test['price']

# Scale
sc = StandardScaler()

sc.fit(X_train)
X_train_scaled = sc.transform(X_train)
X_test_scaled = sc.transform(X_test)
```

3.1 `GridSearchCV`

The function is used to compute the cross-validated score *(MSE, RMSE, accuracy, etc.)* over a grid of hyperparameter values. This helps avoid nested `for()` loops if multiple hyperparameter values need to be tuned.

```
# GridSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor() # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
    # the keys should be EXACTLY the same as the names of the model inputs
    # the values should be an array or list of hyperparam values you want to try out

# 30 K values x 2 weight settings x 3 metric settings = 180 different combinations in this g
grid = {'n_neighbors': np.arange(5, 151, 5), 'weights':['uniform', 'distance'],
        'metric': ['manhattan', 'euclidean', 'chebyshev']}
# 3) Create the Kfold object (Using RepeatedKFold will be more robust, but more expensive, u
# have the budget)
```

```
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)

# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within t
gcv = GridSearchCV(model, grid, cv = kfold, scoring = 'neg_root_mean_squared_error', n_jobs =

# Fit the models, and cross-validate
gcv.fit(X_train_scaled, y_train)
```

```
Fitting 5 folds for each of 180 candidates, totalling 900 fits
```

```
GridSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
             estimator=KNeighborsRegressor(), n_jobs=-1,
             param_grid={'metric': ['manhattan', 'euclidean', 'chebyshev'],
                         'n_neighbors': array([  5,  10,  15,  20,  25,  30,  35,  40,  45,
        70,  75,  80,  85,  90,  95, 100, 105, 110, 115, 120, 125, 130,
       135, 140, 145, 150]),
                         'weights': ['uniform', 'distance']},
             scoring='neg_root_mean_squared_error', verbose=10)
```

The optimal estimator based on cross-validation is:

```
gcv.best_estimator_
```

```
KNeighborsRegressor(metric='manhattan', n_neighbors=10, weights='distance')
```

The optimal hyperparameter values *(based on those considered in the grid search)* are:

```
gcv.best_params_
```

```
{'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}
```

The cross-validated root mean squared error for the optimal hyperparameter values is:

```
-gcv.best_score_
```

```
5740.928686723918
```

The RMSE on test data for the optimal hyperparameter values is:

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

```
5747.466851437544
```

Note that the error is further reduced as compared to the case when we tuned only one hyperparameter in the previous chatper. We must tune all the hyperparameters that can effect prediction accuracy, in order to get the most accurate model.

The results for each cross-validation are stored in the `cv_results_` attribute.

```
pd.DataFrame(gcv.cv_results_).head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_metric	param_n_neighb
0	0.011169	0.005060	0.011768	0.001716	manhattan	5
1	0.009175	0.001934	0.009973	0.000631	manhattan	5
2	0.008976	0.001092	0.012168	0.001323	manhattan	10
3	0.007979	0.000001	0.011970	0.000892	manhattan	10
4	0.006781	0.000748	0.012367	0.001017	manhattan	15

These results can be useful to see if other hyperparameter values are almost equally good.

For example, the next two best optimal values of the hyperparameter correspond to neighbors being 15 and 5 respectively. As the test error has a high variance, the best hyperparameter values need not necessarily be actually optimal.

```
pd.DataFrame(gcv.cv_results_).sort_values(by = 'rank_test_score').head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_metric	param_n_neighb
3	0.007979	0.000001	0.011970	0.000892	manhattan	10
5	0.009374	0.004829	0.013564	0.001850	manhattan	15
1	0.009175	0.001934	0.009973	0.000631	manhattan	5
7	0.007977	0.001092	0.017553	0.002054	manhattan	20
9	0.007777	0.000748	0.019349	0.003374	manhattan	25

Let us compute the RMSE on test data based on the 2nd and 3rd best hyperparameter values.

```
model = KNeighborsRegressor(n_neighbors=15, metric='manhattan', weights='distance').fit(X_tra
mean_squared_error(model.predict(X_test_scaled), y_test, squared = False)
```

```
5800.418957612656
```

```
model = KNeighborsRegressor(n_neighbors=5, metric='manhattan', weights='distance').fit(X_tra
mean_squared_error(model.predict(X_test_scaled), y_test, squared = False)
```

```
5722.4859230146685
```

We can see that the RMSE corresponding to the 3rd best hyperparameter value is the least. Due to variance in test errors, it may be a good idea to consider the set of top few best hyperparameter values, instead of just considering the best one.

3.2 RandomizedSearchCV()

In case of many possible values of hyperparameters, it may be comptaionally very expensive to use `GridSearchCV()`. In such cases, `RandomizedSearchCV()` can be used to compute the cross-validated score on a randomly selected subset of hyperparameter values from the specified grid. The number of values can be fixed by the user, as per the available budget.

```
# RandomizedSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor() # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
    # the keys should be EXACTLY the same as the names of the model inputs
    # the values should be an array or list of hyperparam values, or distribution of hyperpa


grid = {'n_neighbors': range(1, 500), 'weights':['uniform', 'distance'],
        'metric': ['minkowski'], 'p': uniform(loc=1, scale=10)} #We can specify a distributio
                                                                #for continuous hyperparamete

# 3) Create the Kfold object (Using RepeatedKFold will be more robust, but more expensive, u
# have the budget)
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
```

```
# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within th
gcv = RandomizedSearchCV(model, param_distributions = grid, cv = kfold, n_iter = 180, random_
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1, verbose = 10)

# Fit the models, and cross-validate
gcv.fit(X_train_scaled, y_train)
```

```
Fitting 5 folds for each of 180 candidates, totalling 900 fits
```

```
RandomizedSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
                   estimator=KNeighborsRegressor(), n_iter=180, n_jobs=-1,
                   param_distributions={'metric': ['minkowski'],
                                        'n_neighbors': range(1, 500),
                                        'p': <scipy.stats._distn_infrastructure.rv_continuous
                                        'weights': ['uniform', 'distance']},
                   random_state=10, scoring='neg_root_mean_squared_error',
                   verbose=10)
```

```
gcv.best_params_
```

```
{'metric': 'minkowski',
 'n_neighbors': 3,
 'p': 1.252639454318171,
 'weights': 'uniform'}
```

```
gcv.best_score_
```

```
-6239.171627183809
```

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

```
6176.533397589911
```

Note that in this example, `RandomizedSearchCV()` helps search for optimal values of the hyperparameter p over a continuous domain space. In this dataset, $p = 1$ seems to be the optimal value. However, if the optimal value was somewhere in the middle of a larger continuous domain space *(instead of the boundary of the domain space)*, and there were several other hyperparameters, some of which were not influencing the response *(effect sparsity)*, `RandomizedSearchCV()` is likely to be more effective in estimating the optimal value of the continuous hyperparameter.

The advantages of `RandomizedSearchCV()` over `GridSearchCV()` are:

1. `RandomizedSearchCV()` fixes the computational cost in case of large number of hyperparameters / large number of levels of individual hyperparameters. If there are n hyper parameters, each with 3 levels, the number of all possible hyperparameter values will be 3^n. The computational cost increase exponentially with increase in number of hyperparameters.

2. In case of a hyperparameter having continuous values, the distribution of the hyperparameter can be specified in `RandomizedSearchCV()`.

3. In case of effect sparsity of hyperparameters, i.e., if only a few hyperparameters significantly effect prediction accuracy, `RandomizedSearchCV()` is likely to consider more unique values of the influential hyperparameters as compared to `GridSearchCV()`, and is thus likely to provide more optimal hyperparameter values as compared to `GridSearchCV()`. The figure below shows effect sparsity where there are 2 hyperparameters, but only one of them is associated with the cross-validated score, Here, it is more likely that the optimal cross-validated score will be obtained by `RandomizedSearchCV()`, as it is evaluating the model on 9 unique values of the relevant hyperparameter, instead of just 3.

```
<IPython.core.display.Image object>
```

3.3 `BayesSearchCV()`

Unlike the grid search and random search, which treat hyperparameter sets independently, the Bayesian optimization is an informed search method, meaning that it learns from previous iterations. The number of trials in this approach is determined by the user.

- The function begins by computing the cross-validated score by randomly selecting a few hyperparameter values from the specified disttribution of hyperparameter values.
- Based on the data of hyperparameter values tested *(predictors)*, and the cross-validated score *(the response)*, a Gaussian process model is developed to estimate the cross-validated score & the uncertainty in the estimate in the entire space of the hyperparameter values

- A criterion that "explores" uncertain regions of the space of hyperparameter values *(where it is difficult to predict cross-validated score)*, and "exploits" promising regions of the space are of hyperparameter values *(where the cross-validated score is predicted to minimize)* is used to suggest the next hyperparameter value that will potentially minimize the cross-validated score
- Cross-validated score is computed at the suggested hyperparameter value, the Gaussian process model is updated, and the previous step is repeated, until a certain number of iterations specified by the user.

To summarize, instead of blindly testing the model for the specified hyperparameter values *(as in GridSearchCV())*, or randomly testing the model on certain hyperparameter values *(as in RandomizedSearchCV())*, `BayesSearchCV()` smartly tests the model for those hyperparameter values that are likely to reduce the cross-validated score. The algorithm becomes "smarter" as it "learns" more with increasing iterations.

Here is a nice blog, if you wish to understand more about the Bayesian optimization procedure.

```
# BayesSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor(metric = 'minkowski') # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
# the keys should be EXACTLY the same as the names of the model inputs
# the values should be the distribution of hyperparameter values. Lists and NumPy arrays can
# also be used

grid = {'n_neighbors': Integer(1, 500), 'weights': Categorical(['uniform', 'distance']),
       'p': Real(1, 10, prior = 'uniform')}

# 3) Create the Kfold object (Using RepeatedKFold will be more robust, but more expensive,
# use it if you have the budget)
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)

# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within
# the function
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)

# Fit the models, and cross-validate
# Sometimes the Gaussian process model predicting the cross-validated score suggests a
# "promising point" (i.e., set of hyperparameter values) for cross-validation that it has
# already suggested earlier. In such  a case a warning is raised, and the objective
# function (i.e., the cross-validation score) is computed at a randomly selected point
# (as in RandomizedSearchCV()). This feature helps the algorithm explore other regions of
# the hyperparameter space, rather than only searching in the promising regions. Thus, it
# balances exploration (of the hyperparameter space) with exploitation (of the promising
# regions of the hyperparameter space)

warnings.filterwarnings("ignore")
gcv.fit(X_train_scaled, y_train)
warnings.resetwarnings()
```

The optimal hyperparameter values *(based on Bayesian search)* on the provided distribution of hyperparameter values are:

```
gcv.best_params_
```

```
OrderedDict([('n_neighbors', 9),
             ('p', 1.0008321732366932),
             ('weights', 'distance')])
```

The cross-validated root mean squared error for the optimal hyperparameter values is:

```
-gcv.best_score_
```

```
5756.172382596493
```

The RMSE on test data for the optimal hyperparameter values is:

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

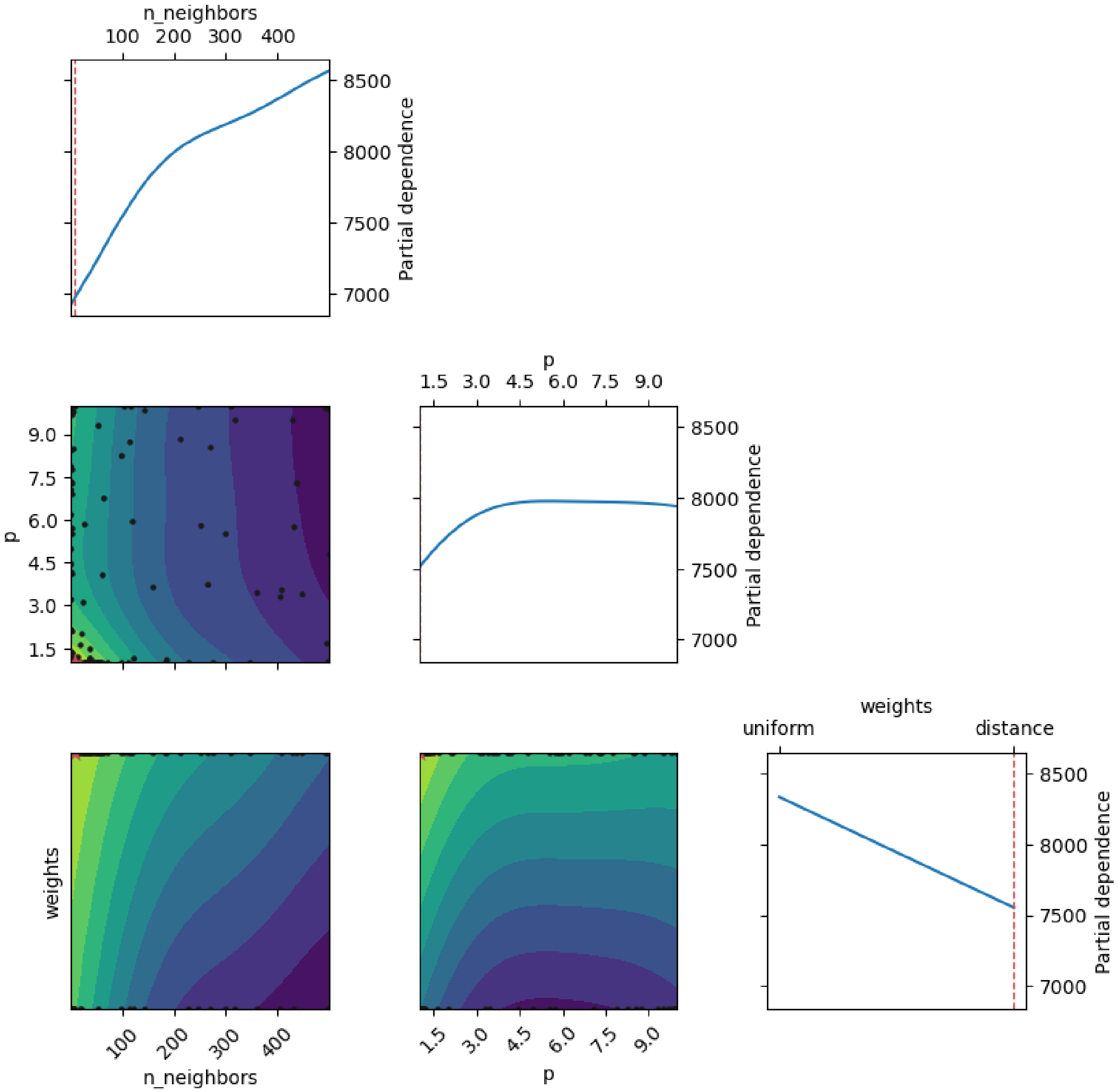
```
5740.432278861367
```

3.3.1 Diagonosis of cross-validated score optimization

Below are the partial dependence plots of the objective function *(i.e., the cross-validated score)*. The cross-validated score predictions are based on the most recently updated model *(i.e., the updated Gaussian Process model at the end of* ***n_iter*** *iterations specified by the user)* that predicts the cross-validated score.

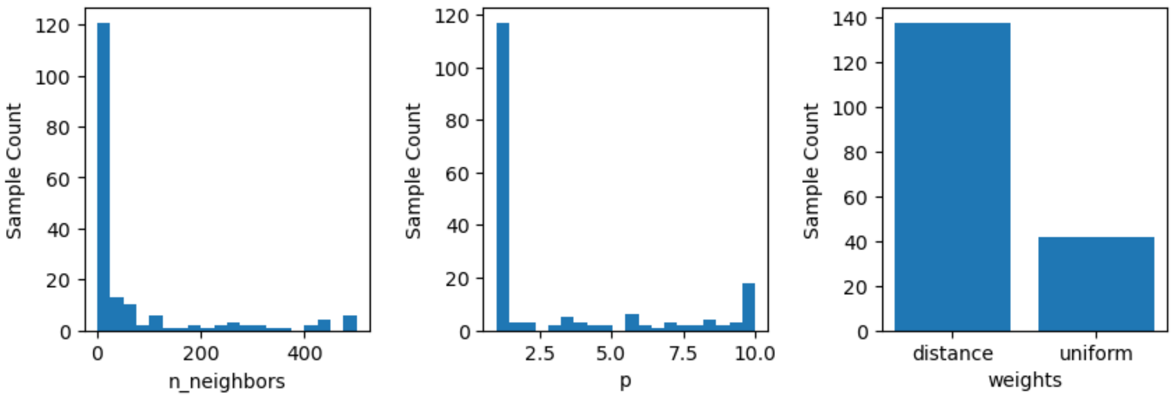
Check the `plot_objective()` documentation to interpret the plots.

```
plot_objective(gcv.optimizer_results_[0],
                   dimensions=["n_neighbors", "p", "weights"], size = 3)
plt.show();
```



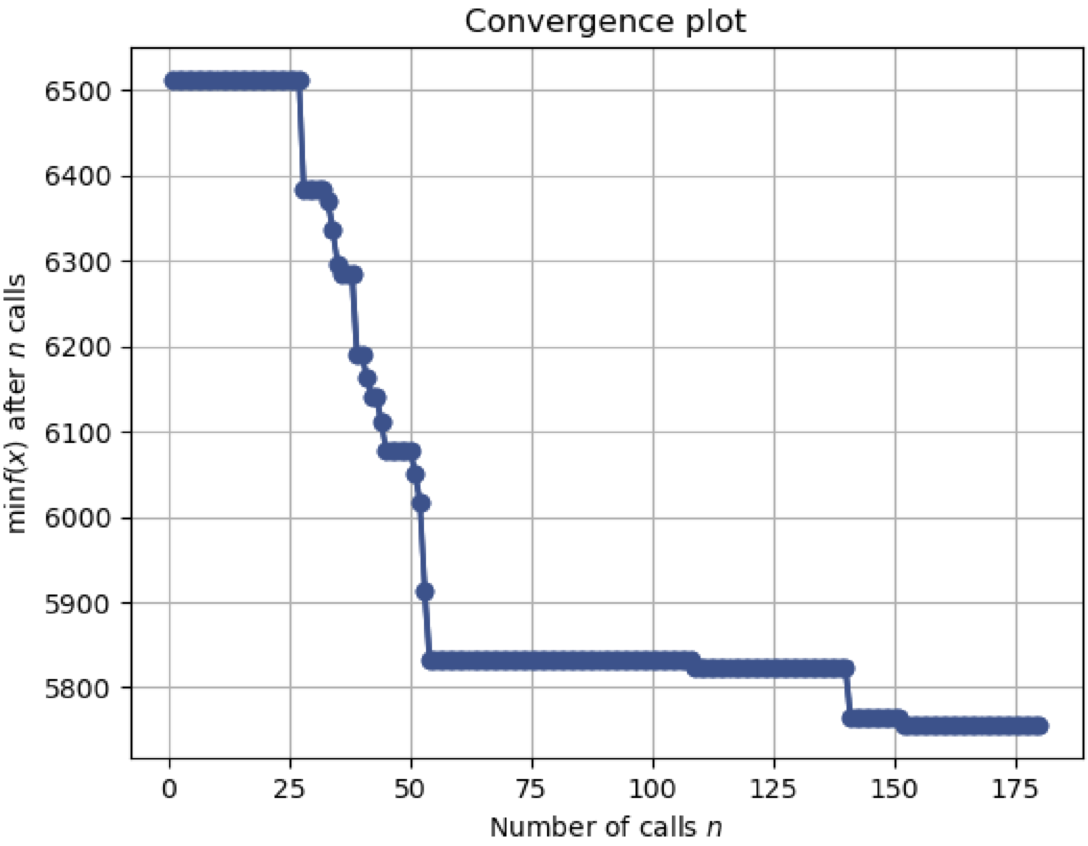
The frequence of individual hyperparameter values considered can also be visualized as below.

```
fig, ax = plt.subplots(1, 3, figsize = (10, 3))
plt.subplots_adjust(wspace=0.4)
plot_histogram(gcv.optimizer_results_[0], 0, ax = ax[0])
plot_histogram(gcv.optimizer_results_[0], 1, ax = ax[1])
plot_histogram(gcv.optimizer_results_[0], 2, ax = ax[2])
plt.show()
```



Below is the plot showing the minimum cross-validated score computed obtained until 'n' hyperparameter values are considered for cross-validation.

```
plot_convergence(gcv.optimizer_results_)
plt.show()
```



Note that the cross-validated error is close to the optmial value in the 53rd iteration itself.

The cross-validated error at the 53rd iteration is:

```
gcv.optimizer_results_[0]['func_vals'][53]
```

```
5831.87280274334
```

The hyperparameter values at the 53rd iterations are:

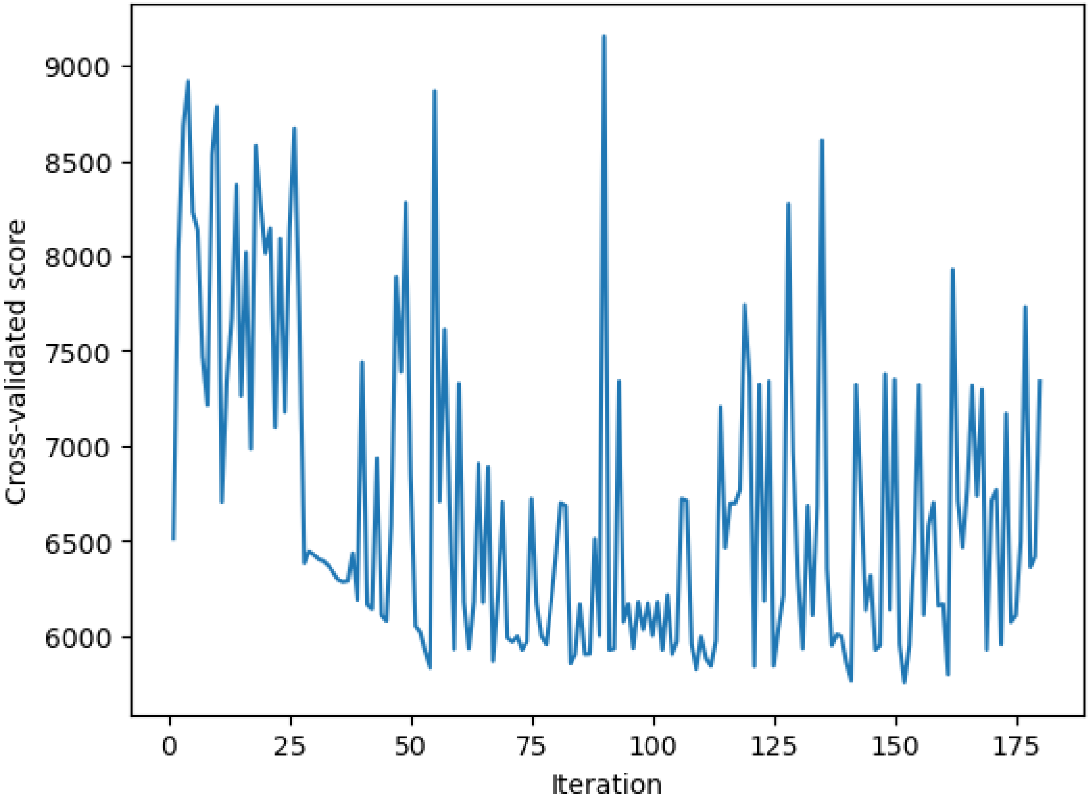
```
gcv.optimizer_results_[0]['x_iters'][53]
```

```
[15, 1.0, 'distance']
```

Note that this is the 2nd most optimal hyperparameter value based on `GridSearchCV()`.

Below is the plot showing the cross-validated score computed at each of the 180 hyperparameter values considered for cross-validation. The plot shows that the algorithm seems to explore new regions of the domain space, instead of just exploting the promising ones. There is a balance between exploration and exploitation for finding the optimal hyperparameter values that minimize the objective function *(i.e., the function that models the cross-validated score)*.

```
sns.lineplot(x = range(1, 181), y = gcv.optimizer_results_[0]['func_vals'])
plt.xlabel('Iteration')
plt.ylabel('Cross-validated score')
plt.show();
```



The advantages of `BayesSearchCV()` over `GridSearchCV()` and `RandomizedSearchCV()` are:

1. The Bayesian Optimization approach gives the benefit that we can give a much larger range of possible values, since over time we identify and exploit the most promising regions and discard the not so promising ones. Plain grid-search would burn computational resources to explore all regions of the domain space with the same granularity, even the not promising ones. Since we search much more effectively in Bayesian search, we can search over a larger domain space.

2. BayesSearch CV may help us identify the optimal hyperparameter value in fewer iterations if the Gaussian process model estimating the cross-validated score is relatively accurate. However, this is not certain. Grid and random search are completely uninformed by past evaluations, and as a result, often spend a significant amount of time evaluating “bad” hyperparameters.

3. BayesSearch CV is more reliable in cases of a large search space, where random selection may miss sampling values from optimal regions of the search space.

The disadvantages of `BayesSearchCV()` over `GridSearchCV()` and `RandomizedSearchCV()` are:

1. `BayesSearchCV()` has a cost of learning from past data, i.e., updating the model that predicts the cross-validated score after every iteration of evaluating the cross-validated score on a new hyperparameter value. This cost will continue to increase as more and more data is collected. There is no such cost in `GridSearchCV()` and `RandomizedSearchCV()` as there is no learning. This implies that each iteration of `BayesSearchCV()` will take a longer time than each iteration of `GridSearchCV()` / `RandomizedSearchCV()`. Thus, even if `BayesSearchCV()` finds the optimal hyperparameter value in fewer iterations, it may take more time than `GridSearchCV()` / `RandomizedSearchCV()` for the same.
2. The success of `BayesSearchCV()` depends on the predictions and associated uncertainty estimated by the Gaussian process (GP) model that predicts the cross-validated score. The GP model, although works well in general, may not be suitable for certain datasets, or may take a relatively large number of iterations to learn for certain datasets.

3.3.2 Live monitoring of cross-validated score

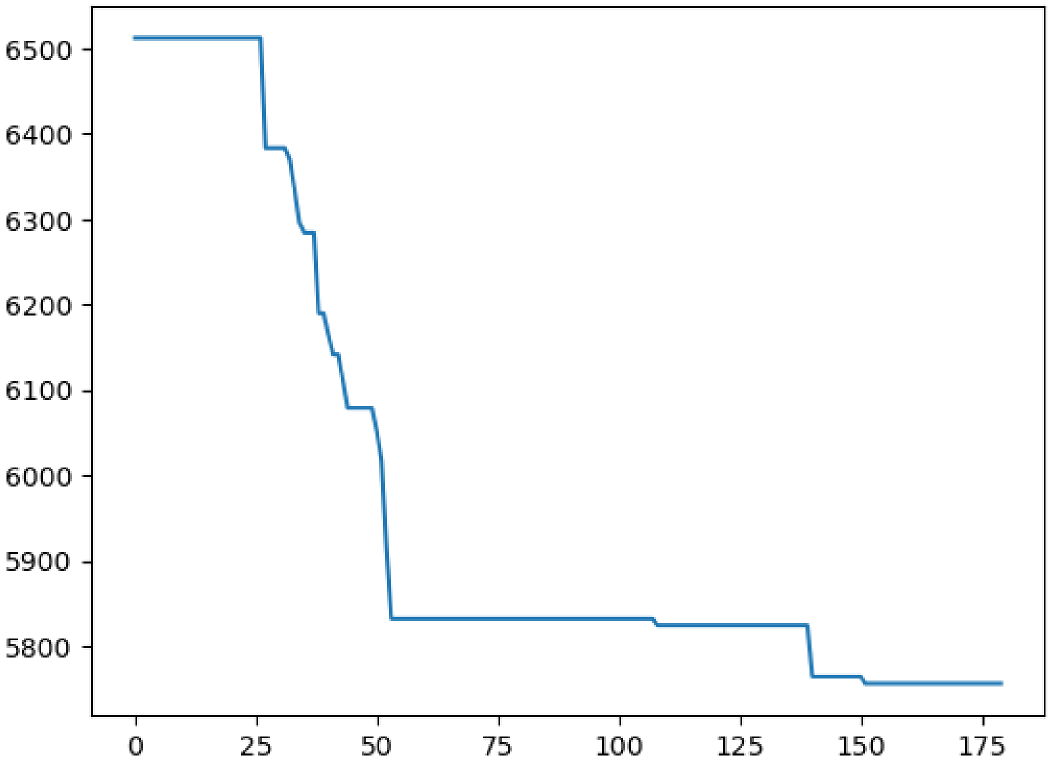
Note that it will be useful monitor the cross-validated score while the Bayesian Search CV code is running, and stop the code as soon as the desired accuracy is reached, or the optimal cross-validated score doesn't seem to improve. The `fit()` method of the `BayesSeaerchCV()` object has a `callback` argument that can be used as follows:

```
model = KNeighborsRegressor(metric = 'minkowski') # No inputs defined inside the model
grid = {'n_neighbors': Integer(1, 500), 'weights': Categorical(['uniform', 'distance']),
       'p': Real(1, 10, prior = 'uniform')}

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
```

```
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
```

```
gcv.fit(X_train_scaled, y_train, callback = monitor)
```

```
['n_neighbors', 'p', 'weights'] = [9, 1.0008321732366932, 'distance'] 5756.172382596493
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=KNeighborsRegressor(), n_iter=180, n_jobs=-1,
              random_state=10, scoring='neg_root_mean_squared_error',
              search_spaces={'n_neighbors': Integer(low=1, high=500, prior='uniform', transf
                             'p': Real(low=1, high=10, prior='uniform', transform='normalize
                             'weights': Categorical(categories=('uniform', 'distance'), prio
```

3.4 `cross_validate()`

We have used `cross_val_score()` and `cross_val_predict()` so far.

When can we use one over the other?

The function `cross_validate()` is similar to `cross_val_score()` except that it has the option to return multiple cross-validated metrics, instead of a single one.

Consider the heart disease classification problem, where the response is `target` *(whether the person has a heart disease or not)*.

```
data = pd.read_csv('Datasets/heart_disease_classification.csv')
data.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Let us pre-process the data.

```
# First, separate the response and the predictors
y = data['target']
X = data.drop('target', axis=1)
```

```
# Separate the data (X,y) into training and test

# Inputs:
    # data
    # train-test ratio
    # random_state for reproducible code

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=20, st

# stratify=y makes sure the class 0 to class 1 ratio in the training and test sets are kept
```

```
model = KNeighborsClassifier()
sc = StandardScaler()
sc.fit(X_train)
X_train_scaled = sc.transform(X_train)
X_test_scaled = sc.transform(X_test)
```

Suppose we want to take recall above a certain threshold with the highest precision possible. `cross_validate()` computes the cross-validated score for multiple metrics - rest is the same as `cross_val_score()`.

```
Ks = np.arange(10,200,10)

scores = []

for K in Ks:
    model = KNeighborsClassifier(n_neighbors=K) # Keeping distance uniform
    scores.append(cross_validate(model, X_train_scaled, y_train, cv=5, scoring = ['accuracy'
```

```
scores

# The output is now a list of dicts - easy to convert to a df

df_scores = pd.DataFrame(scores) # We need to handle test_recall and test_precision cols

df_scores['CV_recall'] = df_scores['test_recall'].apply(np.mean)
df_scores['CV_precision'] = df_scores['test_precision'].apply(np.mean)
df_scores['CV_accuracy'] = df_scores['test_accuracy'].apply(np.mean)

df_scores.index = Ks # We can set K values as indices for convenience


#df_scores
# What happens as K increases?
    # Recall increases (not monotonically)
    # Precision decreases (not monotonically)
# Why?
    # Check the class distribution in the data - more obs with class 1
    # As K gets higher, the majority class overrules (visualized in the slides)
    # More 1s means less FNs - higher recall
    # More 1s means more FPs - lower precision
# Would this be the case for any dataset?
    # NO!! Depends on what the majority class is!
```

Suppose we wish to have the maximum possible precision for at least 95% recall.

The optimal K will be:

```
df_scores.loc[df_scores['CV_recall'] > 0.95, 'CV_precision'].idxmax()
```

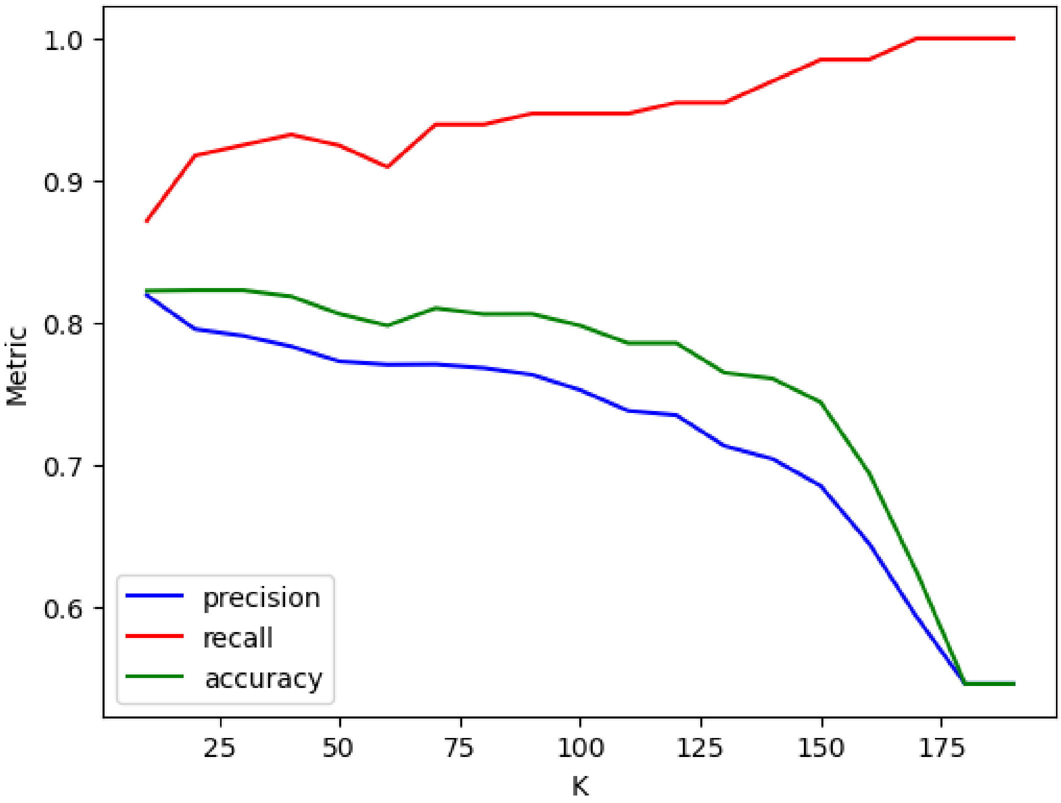
```
120
```

The cross-validated precision, recall and accuracy for the optimal K are:

```
df_scores.loc[120, ['CV_recall', 'CV_precision', 'CV_accuracy']]
```

```
CV_recall       0.954701
CV_precision    0.734607
CV_accuracy     0.785374
Name: 120, dtype: object
```

```
sns.lineplot(x = df_scores.index, y = df_scores.CV_precision, color = 'blue', label = 'precis
sns.lineplot(x = df_scores.index, y = df_scores.CV_recall, color = 'red', label = 'recall')
sns.lineplot(x = df_scores.index, y = df_scores.CV_accuracy, color = 'green', label = 'accura
plt.ylabel('Metric')
plt.xlabel('K')
plt.show()
```



Part II

Tree based models

4 Regression trees

Read section 8.1.1 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.35)

# import the decision tree regressor
from sklearn.tree import DecisionTreeRegressor, plot_tree, export_graphviz

# split the dataset into training and testing sets
from sklearn.model_selection import train_test_split


from sklearn.model_selection import cross_val_score, GridSearchCV, cross_val_predict, KFold

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer
from sklearn.metrics import root_mean_squared_error, r2_score, make_scorer
```

We will use the same dataset as in the KNN model for regression trees.

```
# Load the dataset
car = pd.read_csv('Datasets/car.csv')
car.head()
```

	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	vw	Beetle	2014	Manual	55457	Diesel	30	65.3266	1.6	7490
1	vauxhall	GTC	2017	Manual	15630	Petrol	145	47.2049	1.4	10998
2	merc	G Class	2012	Automatic	43000	Diesel	570	25.1172	3.0	44990
3	audi	RS5	2019	Automatic	10	Petrol	145	30.5593	2.9	51990
4	merc	X-CLASS	2018	Automatic	14000	Diesel	240	35.7168	2.3	28990

4.1 Native Support for Missing Values

Starting with **scikit-learn version 1.3**, classical tree-based models in `scikit-learn` have added **native support for missing values**, which simplifies preprocessing and improves model robustness:

- `DecisionTreeClassifier` supports missing values as of **version 1.3.0**

- `RandomForestClassifier` adds support in **version 1.4.0**

This means you no longer need to impute missing values manually before training these models.

To take advantage of this feature, first check your `scikit-learn` version:

```python
import sklearn
print(sklearn.__version__)
```

```
1.6.1
```

If your version is below `1.4.0`, you can upgrade by running:

```python
# pip install --upgrade scikit-learn
```

```python
# Make a copy of the original dataset
car_missing = car.copy()

# Randomly add missing values
# Inject missing values into 10% of the 'mileage' column
car_missing.loc[car_missing.sample(frac=0.1, random_state=42).index, 'mileage'] = np.nan
# Inject missing values into 10% of the 'fuelType' and 'engineSize' columns

car_missing.loc[car_missing.sample(frac=0.1, random_state=42).index, 'fuelType'] = np.nan
car_missing.loc[car_missing.sample(frac=0.1, random_state=42).index, 'engineSize'] = np.nan
car_missing.isna().sum()
```

```
brand             0
model             0
year              0
transmission      0
mileage         763
fuelType        763
tax               0
mpg               0
engineSize      763
price             0
dtype: int64
```

```
# Split the car_missing dataset into features and target
X_missing = car_missing.drop(columns=['price'])
y_missing = car_missing['price']
```

4.1.1 Build a regression tree using mileage as the solo predictor

```
# Use only 'mileage' as the feature
X_mileage = X_missing[['mileage']]
y_mileage = y_missing
```

```
# Create a DecisionTreeRegressor model
reg_tree = DecisionTreeRegressor(random_state=42)

# Fit the model to the data
reg_tree.fit(X_mileage, y_mileage)
```

```
DecisionTreeRegressor(random_state=42)
```

```
# Predict the target variable using the model
y_pred = reg_tree.predict(X_mileage)

# Calculate the RMSE and R² score
rmse = np.sqrt(np.mean((y_missing - y_pred) ** 2))
r2 = r2_score(y_missing, y_pred)
print(f"RMSE: {rmse:.2f}")
print(f"R²: {r2:.2f}")
```

```
RMSE: 8797.85
R²: 0.71
```

4.2 Building regression trees

```
X = car.drop(columns=['price'])
y = car['price']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

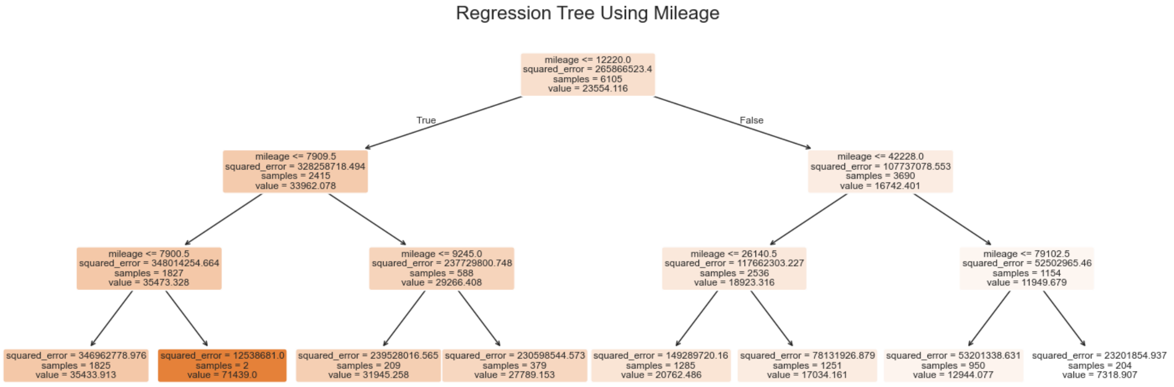
4.2.1 Using only mileage feature

```
# Use only 'mileage' as the feature
X_train_mileage = X_train[['mileage']]
X_test_mileage = X_test[['mileage']]

# Create a DecisionTreeRegressor model
reg_tree = DecisionTreeRegressor(random_state=42, max_depth=3)

# Fit the model to the training data
reg_tree.fit(X_train_mileage, y_train)

# Predict the target variable using the model
y_pred = reg_tree.predict(X_test_mileage)

# Calculate the RMSE and R² score
rmse = np.sqrt(np.mean((y_test - y_pred) ** 2))
r2 = r2_score(y_test, y_pred)
print(f"RMSE using only mileage predictor: {rmse:.2f}")
print(f"R² using only mileage predictor: {r2:.2f}")
```

```
RMSE using only mileage predictor: 14437.80
R² using only mileage predictor: 0.29
```

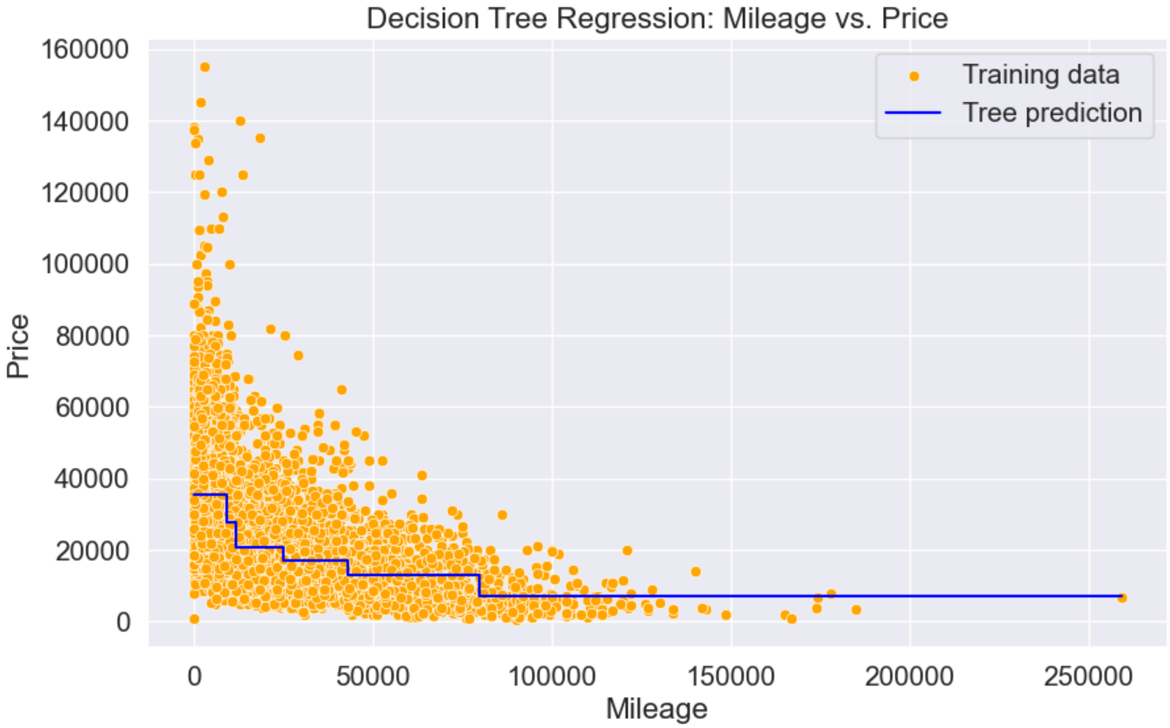
Let’s visualize the tree structure

```
# Plot the tree
plt.figure(figsize=(18, 6))
plot_tree(reg_tree, feature_names=['mileage'], filled=True, rounded=True)
plt.title("Regression Tree Using Mileage")
plt.show()
```



Let's visualize how mileage is used in the decision tree below:

```
# Create evenly spaced mileage values within the range of training data
Xtest = np.linspace(X_train_mileage['mileage'].min(), X_train_mileage['mileage'].max(), 100)

# Convert Xtest to a DataFrame with the correct column name
Xtest_df = pd.DataFrame(Xtest, columns=['mileage'])

# Predict using the DataFrame instead of NumPy array
ytest_pred = reg_tree.predict(Xtest_df)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_train_mileage['mileage'], y=y_train, color='orange', label='Training data

# Step plot to reflect piecewise constant predictions
plt.step(Xtest_df['mileage'], ytest_pred, color='blue', label='Tree prediction', where='mid']

plt.xlabel("Mileage")
plt.ylabel("Price")
plt.title("Decision Tree Regression: Mileage vs. Price")
plt.legend()
plt.show()
```



All cars falling within the same terminal node have the same predicted price, which is seen as flat line segments in the above model curve.

4.2.2 Using mileage and brand as predictors

```python
X_train.head()
```

	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize
216	vw	Scirocco	2016	Manual	41167	Diesel	20	55.2654	2.0
4381	merc	CLS Class	2018	Semi-Auto	12078	Diesel	145	47.7624	2.9
6891	hyundi	Santa Fe	2019	Automatic	623	Diesel	145	43.0887	2.2
421	hyundi	IX35	2014	Manual	37095	Diesel	145	53.4862	1.7
505	ford	Edge	2016	Semi-Auto	15727	Diesel	160	49.0741	2.0

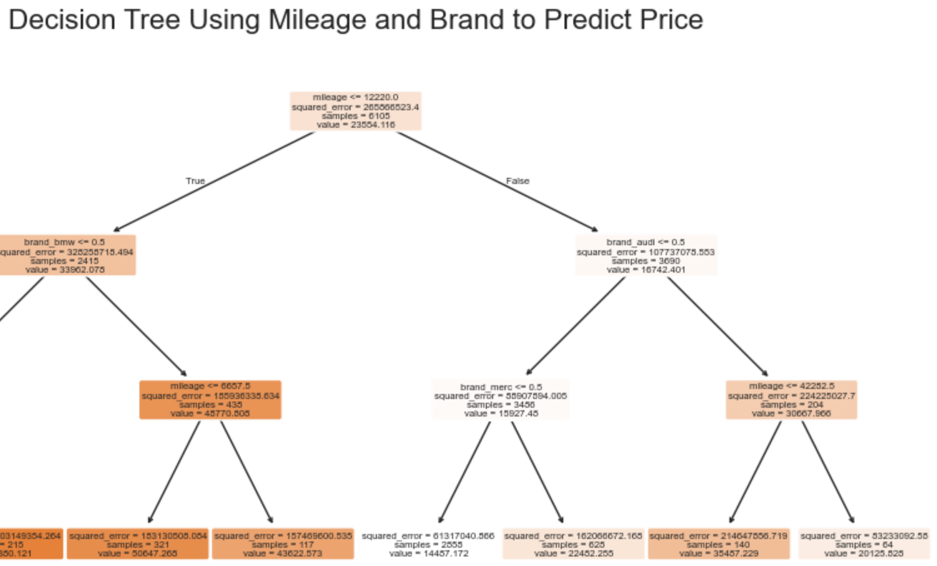
```python
# Select features and target
X_train_tree = X_train[['mileage', 'brand']]

X_test_tree = X_test[['mileage', 'brand']]
```

```python
# One-hot encode the categorical variable 'brand'
X_train_tree_encoded = pd.get_dummies(X_train_tree, columns=['brand'])
X_test_tree_encoded = pd.get_dummies(X_test_tree, columns=['brand'])
```

```python
model = DecisionTreeRegressor(max_depth=3, random_state=42)
model.fit(X_train_tree_encoded, y_train)
```

```
DecisionTreeRegressor(max_depth=3, random_state=42)
```

```python
plt.figure(figsize=(12, 6))
plot_tree(model, feature_names=X_train_tree_encoded.columns, filled=True, rounded=True)
plt.title("Decision Tree Using Mileage and Brand to Predict Price")
plt.show()
```



```python
# Predict the target variable using the model
y_pred_tree = model.predict(X_test_tree_encoded)

# Calculate the RMSE and R² score
rmse_tree = np.sqrt(np.mean((y_test - y_pred_tree) ** 2))
r2_tree = r2_score(y_test, y_pred_tree)
print(f"RMSE using mileage and brand predictor: {rmse_tree:.2f}")
print(f"R² using mileage and brand predictor: {r2_tree:.2f}")
```

```
# Compare the performance of the two models
print(f"RMSE using only mileage predictor: {rmse:.2f}")
print(f"RMSE using mileage and brand predictor: {rmse_tree:.2f}")
# The RMSE using mileage and brand predictor is lower than using only mileage predictor.
# This indicates that adding the brand feature improves the model's performance
```

```
RMSE using mileage and brand predictor: 12531.44
R² using mileage and brand predictor: 0.46
RMSE using only mileage predictor: 14437.80
RMSE using mileage and brand predictor: 12531.44
```

4.2.3 Using all predictors

Now that we've explored a single predictor (`mileage`) and added a second predictor (`brand`), let's take it a step further and use **all available features** to build a more robust model.

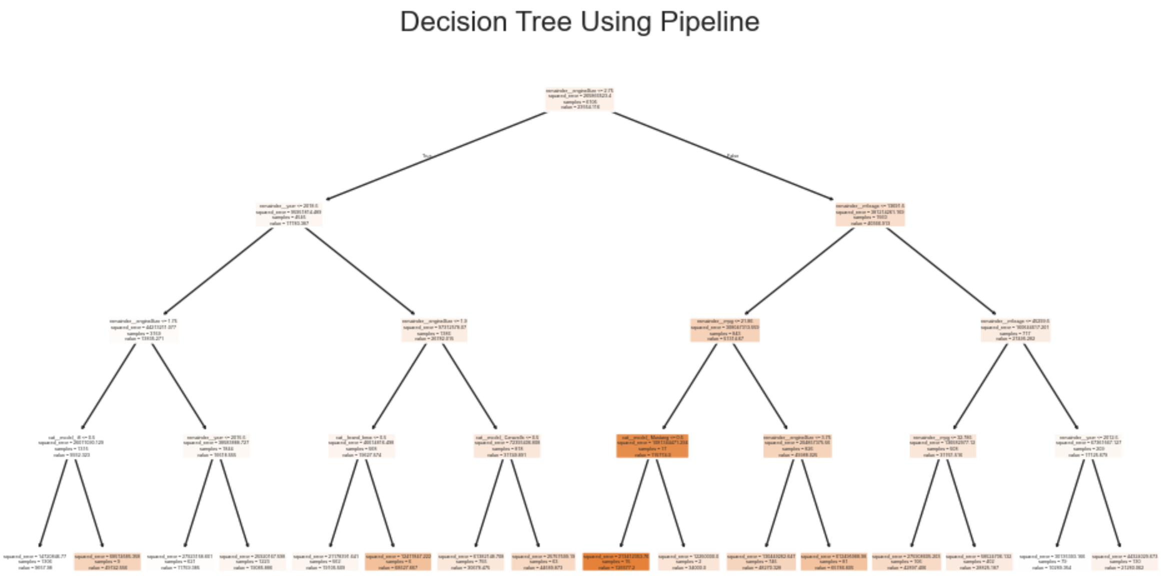
We'll construct a **pipeline** that handles necessary preprocessing steps (e.g., categorical encoding) and fits a **Decision Tree Regressor** in a streamlined and reproducible way.

```
# extract the categorical columns and put them in a list
categorical_feature = X.select_dtypes(include=['object']).columns.tolist()

# extract the numerical columns and put them in a list
numerical_feature = X.select_dtypes(include=['int64', 'float64']).columns.tolist()


# Create a ColumnTransformer to handle encoding
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_feature)
    ],
    remainder='passthrough',  # Keep numerical feature (mileage) unchanged
    force_int_remainder_cols=False
)

# Create the pipeline
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', DecisionTreeRegressor(max_depth=4, random_state=42))
])
```

```
# Usage:
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)

# Calculate the RMSE and R² score
rmse_pipeline = np.sqrt(np.mean((y_test - y_pred) ** 2))
r2_pipeline = r2_score(y_test, y_pred)
print(f"RMSE using pipeline: {rmse_pipeline:.2f}")
print(f"R² using pipeline: {r2_pipeline:.2f}")
```

```
RMSE using pipeline: 8186.34
R² using pipeline: 0.77
```

```
# let's visuzalize the decision tree

plt.figure(figsize=(12, 6))
plot_tree(pipeline.named_steps['regressor'], feature_names=pipeline.named_steps['preprocesso
plt.title("Decision Tree Using Pipeline")
plt.show()
```



4.3 Key Hyperparameters in Decision Tree

In regression trees, **model complexity** is controlled by hyperparameters, tuning them is crucial for balancing **underfitting** and **overfitting**.

4.3.1 Underfitting

- The model is **too simple** to capture patterns in the data.
- High bias, low variance.
- Often caused by:
    - Shallow trees (`max_depth` is too small)
    - Too strict constraints (`min_samples_split` or `min_samples_leaf` is too high)

4.3.2 Overfitting

- The model is **too complex** and learns noise from the training data.
- Low bias, high variance.
- Often caused by:
    - Deep trees with many splits
    - Very small `min_samples_leaf` or `min_samples_split`

Below are the most commonly used hyperparameters:

4.3.3 `max_depth`

- Controls the **maximum depth** of the tree.
- If `None`, the tree will expand until all leaves are pure or contain fewer than `min_samples_split` samples.
- Controls overfitting (deep trees → overfit, shallow trees → underfit)
- Typical values: 3 to 20 (start with lower values).

4.3.4 `min_samples_split`

- The **minimum number of samples** required to split an internal node.
- higher values → simpler trees (reducing overfitting)

4.3.5 min_samples_leaf

- The minimum number of samples required to be at a leaf node.
- Setting this to a higher number can smooth the model by reducing variance.

4.3.6 max_features

- Number of features to consider when looking for the best split.
- Can be set to:
  - "auto" or None: use all features
  - "sqrt": use the square root of the number of features
  - "log2": use log base 2

```
# Define your parameter grid with pipeline step prefix
param_grid = {
    'regressor__max_depth': [3, 5, 7, 10, None],
    'regressor__min_samples_split': [2, 5, 10],
    'regressor__min_samples_leaf': [1, 2, 4],
    'regressor__max_features': ['sqrt', None]
}

# Create custom scorer for RMSE
rmse_scorer = make_scorer(lambda y_true, y_pred: root_mean_squared_error(y_true, y_pred),
                          greater_is_better=False)
# Create GridSearchCV object
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    scoring={
        'RMSE': rmse_scorer,
        'R2': 'r2'
    },
    refit='R2',
    cv=5,
    n_jobs=-1,
    verbose=1
)

# Fit the grid search to training data
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 270 candidates, totalling 1350 fits
```

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(force_int_remainder_cols=False,
                                                          remainder='passthrough',
                                                          transformers=[('cat',
                                                                         OneHotEncoder(handl
                                                                         ['brand',
                                                                          'model',
                                                                          'transmission',
                                                                          'fuelType'])])),
                                       ('regressor',
                                        DecisionTreeRegressor(max_depth=4,
                                                              random_state=42))]),
             n_jobs=-1,
             param_grid={'regressor__ccp_alpha': [0.001, 0.01, 0.1],
                         'regressor__max_depth': [3, 5, 7, 10, None],
                         'regressor__max_features': ['sqrt', None],
                         'regressor__min_samples_leaf': [1, 2, 4],
                         'regressor__min_samples_split': [2, 5, 10]},
             refit='R2',
             scoring={'R2': 'r2',
                      'RMSE': make_scorer(<lambda>, greater_is_better=False, response_method=
             verbose=1)
```

The `GridSearchCV` setup evaluates both **RMSE** and **R²** during cross-validation.

- **R²** is used to **select the best model** and is also used to **refit** the model on the entire training set.
- **RMSE** is computed during the process for evaluation purposes, but it is **not used to determine** the best model.

This allows for more comprehensive model assessment while still optimizing based on a single selected metric.

```
# Get best estimator and predictions
best_model = grid_search.best_estimator_
y_pred_tuned = best_model.predict(X_test)

# Calculate metrics for tuned model
rmse_tuned = root_mean_squared_error(y_test, y_pred_tuned)
r2_tuned = r2_score(y_test, y_pred_tuned)
```

```
print("\n=== Best Parameters ===")
print(grid_search.best_params_)
print("\n=== Tuned Model Performance ===")
print(f"RMSE (Tuned): {rmse_tuned:.2f}")
print(f"R² (Tuned): {r2_tuned:.2f}")
print(f"Improvement in R²: {(r2_tuned - r2_pipeline):.2%}")
```

```
=== Best Parameters ===
{'regressor__ccp_alpha': 0.001, 'regressor__max_depth': None, 'regressor__max_features': None

=== Tuned Model Performance ===
RMSE (Tuned): 4726.17
R² (Tuned): 0.92
Improvement in R²: 15.23%
```

```
print("\n=== Best Parameters ===")
print(grid_search.best_params_)
print("\n=== Tuned Model Performance ===")
print(f"RMSE (Tuned): {rmse_tuned:.2f}")
print(f"R² (Tuned): {r2_tuned:.2f}")
print(f"Improvement in R²: {(r2_tuned - r2_pipeline):.2%}")
```

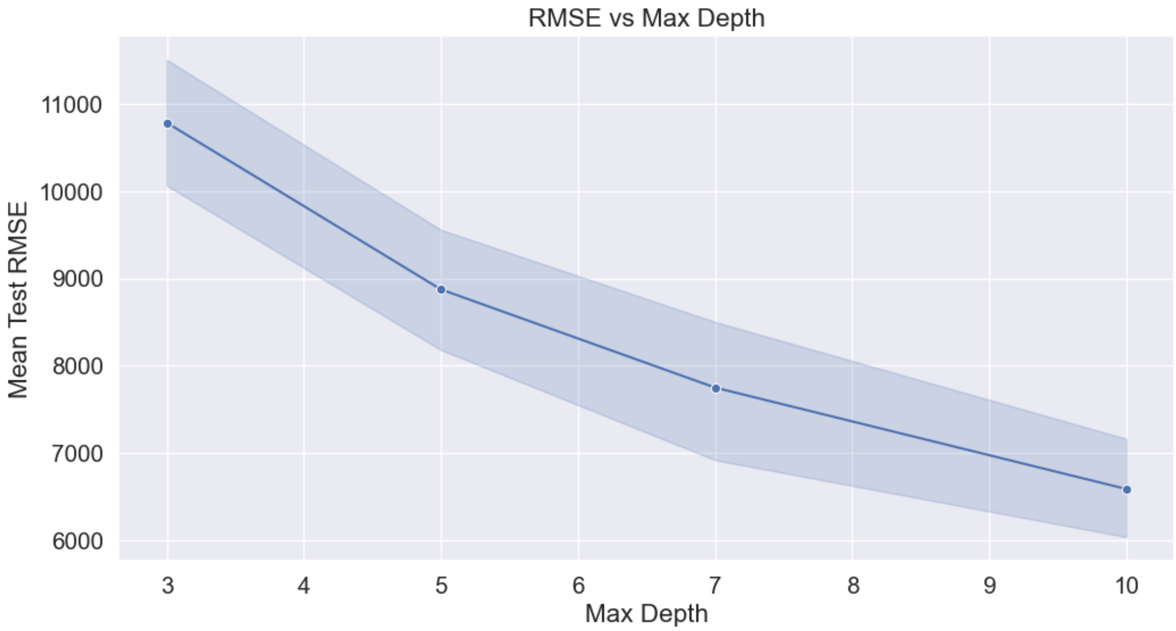
```
=== Best Parameters ===
{'regressor__ccp_alpha': 0.001, 'regressor__max_depth': None, 'regressor__max_features': None

=== Tuned Model Performance ===
RMSE (Tuned): 4726.17
R² (Tuned): 0.92
Improvement in R²: 15.23%
```

`GridSearchCV` improves the r squared from 0.77 to 0.92, increased by 15.23%, Let us visualize the mean squared error based on the hyperparameter values. We'll use the cross validation results stored in the cv_results_ attribute of the GridSearchCV fit() object.

```
#Detailed results of k-fold cross validation
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results.head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_regressor__ccp_alpha	p
0	0.075184	0.013049	0.009999	0.001052	0.001	3
1	0.078505	0.014115	0.010804	0.001051	0.001	3
2	0.077370	0.011081	0.010873	0.000946	0.001	3
3	0.036274	0.036182	0.010617	0.000345	0.001	3
4	0.018702	0.003564	0.012120	0.004090	0.001	3

```
# Plotting the RMSE for different max_depth values
plt.figure(figsize=(12, 6))
sns.lineplot(data=cv_results, x='param_regressor__max_depth', y=np.abs(cv_results['mean_test_
plt.xlabel('Max Depth')
plt.ylabel('Mean Test RMSE')
plt.title('RMSE vs Max Depth');
```

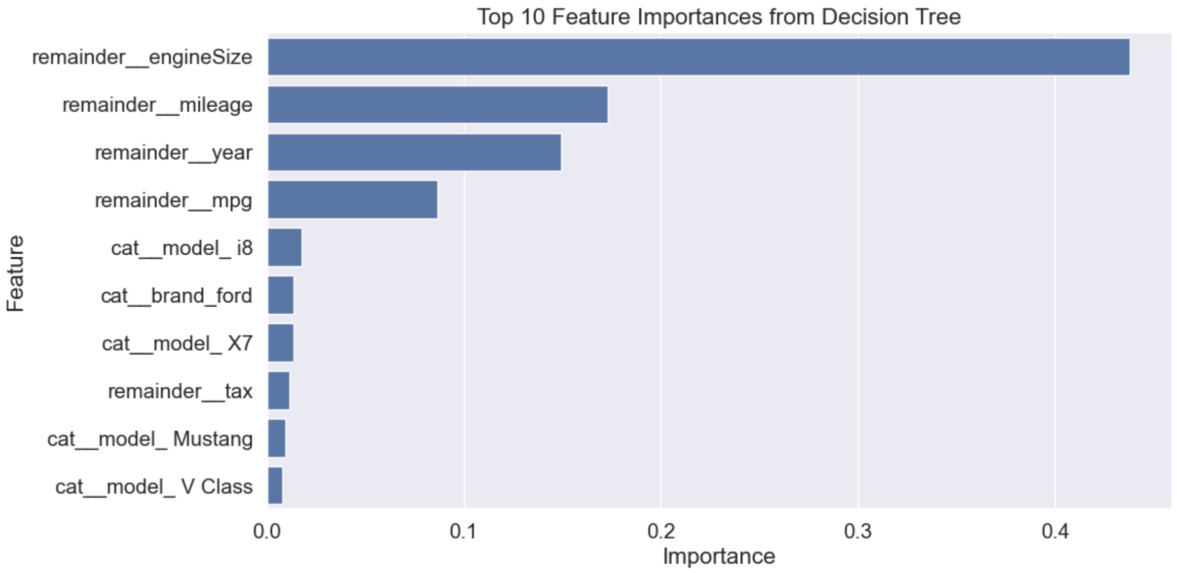


4.3.7 Output feature importance

```
# Get feature importances and names
feature_importances = best_model.named_steps['regressor'].feature_importances_
feature_names = best_model.named_steps['preprocessor'].get_feature_names_out()
```

```
# Create DataFrame and select top 10
feature_importance_df = (
    pd.DataFrame({'Feature': feature_names, 'Importance': feature_importances})
    .sort_values(by='Importance', ascending=False)
    .head(10)  # Keep only top 10 features
)

# Print top 10 features
print("=== Top 10 Feature Importances ===")
print(feature_importance_df)

# Plot top 10 features
plt.figure(figsize=(12, 6))
sns.barplot(data=feature_importance_df, x='Importance', y='Feature')
plt.title('Top 10 Feature Importances from Decision Tree')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

```
=== Top 10 Feature Importances ===
                   Feature  Importance
112  remainder__engineSize    0.437921
109     remainder__mileage    0.173215
108        remainder__year    0.149303
111         remainder__mpg    0.086922
98          cat__model_ i8    0.017971
2          cat__brand_ford    0.013837
92          cat__model_ X7    0.013477
110         remainder__tax    0.011502
60     cat__model_ Mustang    0.009517
86     cat__model_ V Class    0.008147
```



4.4 Cost-Complexity Pruning (`ccp_alpha`)

Cost-complexity pruning is a post-pruning technique used to reduce the size of a decision tree by removing sections that provide little to no improvement in prediction accuracy. It helps prevent **overfitting** and improves **model generalization**.

4.4.1 Key Idea

Each subtree in a decision tree has an associated **cost-complexity score**:

$ R_ (T) = R(T) + |T| $

- $ R(T) $: Total training error of the tree (T)
- $ |T| $: Number of leaf nodes in the tree
- *alpha* (**ccp_alpha**): Complexity parameter that penalizes tree size

As *alpha* increases, the tree is **pruned more aggressively**.

4.4.2 Parameter: `ccp_alpha` in scikit-learn

- Available in `DecisionTreeRegressor` and `DecisionTreeClassifier`
- Default: `ccp_alpha = 0.0` (no pruning)
- Increasing `ccp_alpha` encourages simpler trees by penalizing extra leaf nodes

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, mean_squared_error
import numpy as np

# Define your parameter grid with pipeline step prefix
param_grid = {
    'regressor__ccp_alpha': [0.0, 0.001, 0.01, 0.1]
}

# Create custom scorer for RMSE
rmse_scorer = make_scorer(lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred))
                          greater_is_better=False)

# Create GridSearchCV object
grid_search_ccp = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    scoring={
        'RMSE': rmse_scorer,
        'R2': 'r2'
    },
    refit='R2',
    cv=5,
    n_jobs=-1,
    verbose=1
)

# Fit the grid search to training data
grid_search_ccp.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
```

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(force_int_remainder_cols=False,
                                                          remainder='passthrough',
                                                          transformers=[('cat',
                                                                         OneHotEncoder(handl
                                                                         ['brand',
                                                                          'model',
                                                                          'transmission',
                                                                          'fuelType'])])),
                                       ('regressor',
                                        DecisionTreeRegressor(max_depth=4,
                                                              random_state=42))]),
             n_jobs=-1,
             param_grid={'regressor__ccp_alpha': [0.0, 0.001, 0.01, 0.1]},
             refit='R2',
             scoring={'R2': 'r2',
                      'RMSE': make_scorer(<lambda>, greater_is_better=False, response_method=
             verbose=1)
```

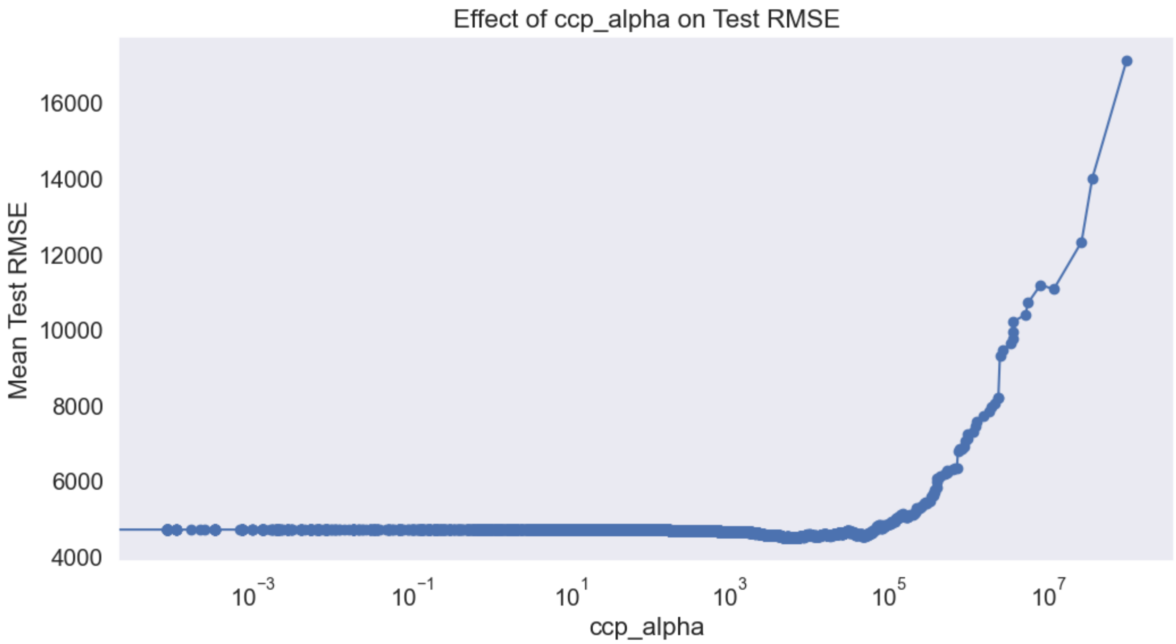
```
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)

X_train_encoded = encoder.fit_transform(X_train[categorical_feature])
X_test_encoded = encoder.transform(X_test[categorical_feature])

# Convert the encoded features back to DataFrame
X_train_encoded_df = pd.DataFrame(X_train_encoded, columns=encoder.get_feature_names_out(cate
X_test_encoded_df = pd.DataFrame(X_test_encoded, columns=encoder.get_feature_names_out(catego

# Concatenate the encoded features with the original numerical features
X_train_final = pd.concat([X_train_encoded_df, X_train[numerical_feature].reset_index(drop=Tr
X_test_final = pd.concat([X_test_encoded_df, X_test[numerical_feature].reset_index(drop=True)

# Check the final shape of the training and testing sets
print("Training set shape:", X_train_final.shape)
print("Testing set shape:", X_test_final.shape)
# Check the first few rows of the final training set
X_train_final.head()
# Check the first few rows of the final testing set
```

```
Training set shape: (6105, 113)
Testing set shape: (1527, 113)
```

	brand_audi	brand_bmw	brand_ford	brand_hyundi	brand_merc	brand_skoda	brand_toyota
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	1.0	0.0	0.0
2	0.0	0.0	0.0	1.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	1.0	0.0	0.0	0.0	0.0

```
model = DecisionTreeRegressor(random_state = 1)#model without any restrictions
path= model.cost_complexity_pruning_path(X_train_final,y_train)# Compute the pruning path du
```

```
# Extract the effective alphas and the corresponding performance metrics
ccp_alphas = path.ccp_alphas
impurities = path.impurities
# Create a DataFrame to store the results
ccp_results = pd.DataFrame({'ccp_alpha': ccp_alphas, 'impurity': impurities})
# Fit the model for each alpha value and calculate the mean test score
mean_test_scores = []
for alpha in ccp_alphas:
    model = DecisionTreeRegressor(random_state=1, ccp_alpha=alpha)
    model.fit(X_train_final, y_train)
    y_pred = model.predict(X_test_final)
    mean_test_scores.append(np.sqrt(mean_squared_error(y_test, y_pred)))
# Add the mean test scores to the DataFrame
ccp_results['mean_test_score'] = mean_test_scores
# Plot the results
plt.figure(figsize=(12, 6))
plt.plot(ccp_results['ccp_alpha'], ccp_results['mean_test_score'], marker='o')
plt.xlabel('ccp_alpha')
plt.ylabel('Mean Test RMSE')
plt.title('Effect of ccp_alpha on Test RMSE')
plt.xscale('log')
plt.grid()
```


Effect of ccp_alpha on Test RMSE
Mean Test RMSE
16000
14000
12000
10000
8000
6000
4000
10^{-3}
10^{-1}
10^{1}
10^{3}
10^{5}
10^{7}
ccp_alpha

5 Classification trees

Read section 8.1.2 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

Import libraries

```
# %load ../standard_import.txt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from IPython.display import Image

from sklearn.model_selection import train_test_split, cross_val_score
from six import StringIO
from sklearn.tree import export_graphviz, DecisionTreeClassifier, plot_tree
from sklearn.metrics import confusion_matrix, accuracy_score

%matplotlib inline
```

```
# load the dataset
heart_df  = pd.read_csv('datasets/heart_disease_classification.csv')
print(heart_df .shape)
heart_df .head()
```

```
(303, 14)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

```
# print out target distribution
heart_df.target.value_counts()
```

```
target
1    165
0    138
Name: count, dtype: int64
```

```
# split the x and y data
X = heart_df.drop(columns=['target'])
y = heart_df.target

# split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

5.1 Building a Classification Tree

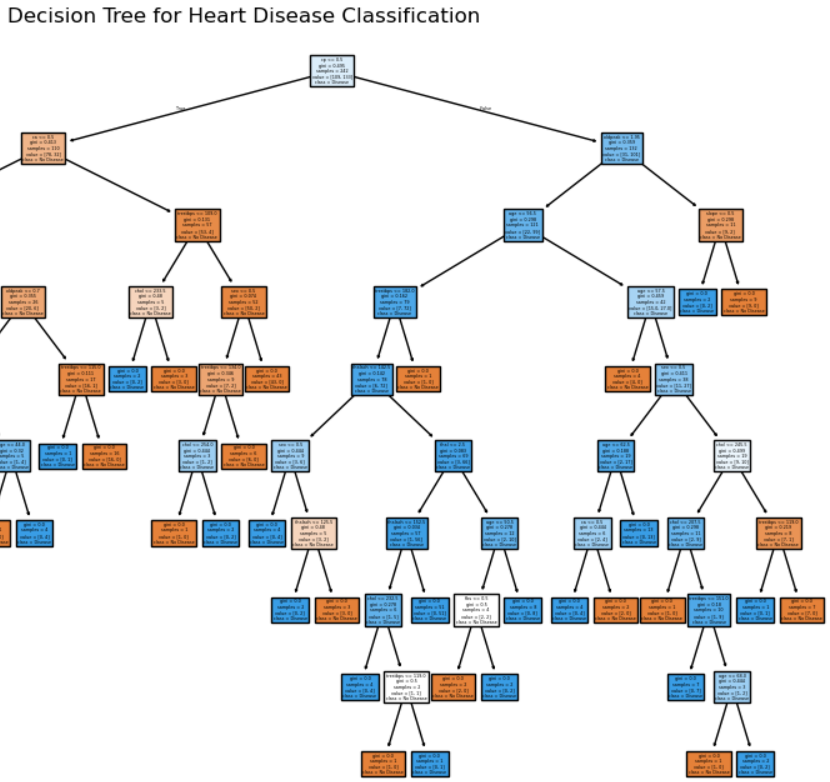
We will build a classification tree to predict whether a person has heart disease, using the default parameters of the decision tree classifier.

```
# create a decision tree classifier
tree = DecisionTreeClassifier(random_state=42)

# fit the model to the training data
tree.fit(X_train, y_train)

# make predictions on the test data
y_pred = tree.predict(X_test)

# calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Test Accuracy: {accuracy:.2f}')

# train accuracy
```

```python
y_train_pred = tree.predict(X_train)
train_accuracy = accuracy_score(y_train, y_train_pred)
print(f'Train Accuracy: {train_accuracy:.2f}')
```

```
Test Accuracy: 0.75
Train Accuracy: 1.00
```

```python
# plot the decision tree
plt.figure(figsize=(12, 8))
plot_tree(tree, filled=True, feature_names=X.columns, class_names=['No Disease', 'Disease'])
plt.title('Decision Tree for Heart Disease Classification');
```



```python
# get the number of leaves in the tree
num_leaves = tree.get_n_leaves()
print(f'Number of leaves: {num_leaves}')

# get the depth of the tree
```

```
tree_depth = tree.get_depth()
print(f'Depth of the tree: {tree_depth}')
```

```
Number of leaves: 41
Depth of the tree: 9
```

Clearly, the model is overfitting, as indicated by a training accuracy of 100% and a much lower test accuracy of 75%.
Next, we will explore different strategies to address and reduce overfitting.

5.2 Pre-pruning: Hyperparameters Tuning

Maximum depth of tree (`max_depth`) - Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

Minimum samples for a node split (`min_samples_split`) - Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting. - Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.

Minimum samples for a terminal node (`min_samples_leaf`) - Defines the minimum samples (or observations) required in a terminal node or leaf. - Used to control over-fitting similar to `min_samples_split`. - Generally lower values should be chosen for imbalanced class problems because the regions in which the minority class will be in majority will be very small.

Maximum number of terminal nodes (`max_leaf_nodes`) - The maximum number of terminal nodes or leaves in a tree.

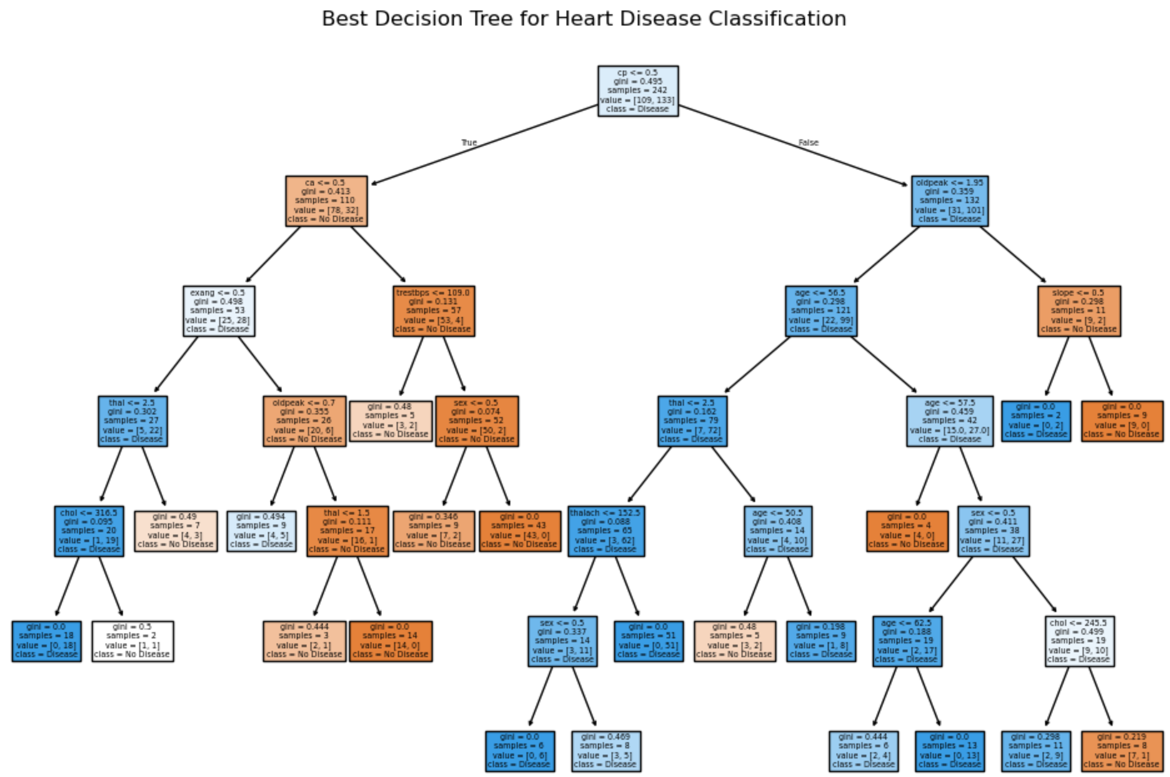
```
# hyperparameter tuning

from sklearn.model_selection import GridSearchCV

# define the parameter grid
param_grid = {
    'max_depth': list(range(1, 9)) + [None],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4]
}

# create a grid search object
grid_search = GridSearchCV(estimator=tree, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2
# fit the grid search to the training data
grid_search.fit(X_train, y_train)
# print the best parameters
print("Best parameters found: ", grid_search.best_params_)

# print the best score
print("Best score: ", grid_search.best_score_)
# get the best estimator
best_tree = grid_search.best_estimator_

# make predictions on the test data with the best estimator
y_pred_best = best_tree.predict(X_test)
# calculate the accuracy of the best estimator
best_accuracy = accuracy_score(y_test, y_pred_best)
print(f'Best Test Accuracy: {best_accuracy:.2f}')
```

```
Fitting 5 folds for each of 135 candidates, totalling 675 fits
Best parameters found:  {'max_depth': 6, 'min_samples_leaf': 2, 'min_samples_split': 10}
Best score:  0.7687074829931972
Best Test Accuracy: 0.85
```

```
# print out the best tree depth and number of leaves
best_num_leaves = best_tree.get_n_leaves()
print(f'Best Number of leaves: {best_num_leaves}')

best_tree_depth = best_tree.get_depth()
print(f'Best Depth of the tree: {best_tree_depth}')
```

```
Best Number of leaves: 21
Best Depth of the tree: 6
```

```
# plot the best decision tree
plt.figure(figsize=(12, 8))
plot_tree(best_tree, filled=True, feature_names=X.columns, class_names=['No Disease', 'Diseas
plt.title('Best Decision Tree for Heart Disease Classification')
plt.show()
```


Best Decision Tree for Heart Disease Classification
cp <= 0.5 gini = 0.495 samples = 242 value = [109, 133] class = Disease
True
False
ca <= 0.5 gini = 0.413 samples = 110 value = [78, 32] class = No Disease
oldpeak <= 1.95 gini = 0.359 samples = 132 value = [31, 101] class = Disease
exang <= 0.5 gini = 0.498 samples = 53 value = [25, 28] class = Disease
trestbps <= 109.0 gini = 0.131 samples = 57 value = [53, 4] class = No Disease
age <= 56.5 gini = 0.298 samples = 121 value = [22, 99] class = Disease
slope <= 0.5 gini = 0.298 samples = 11 value = [9, 2] class = No Disease
thal <= 2.5 gini = 0.302 samples = 27 value = [5, 22] class = Disease
oldpeak <= 0.7 gini = 0.355 samples = 26 value = [20, 6] class = No Disease
gini = 0.48 samples = 5 value = [3, 2] class = No Disease
sex <= 0.5 gini = 0.074 samples = 52 value = [50, 2] class = No Disease
thal <= 2.5 gini = 0.162 samples = 79 value = [7, 72] class = Disease
age <= 57.5 gini = 0.459 samples = 42 value = [15.0, 27.0] class = Disease
gini = 0.0 samples = 2 value = [0, 2] class = Disease
gini = 0.0 samples = 9 value = [9, 0] class = No Disease
chol <= 316.5 gini = 0.095 samples = 20 value = [1, 19] class = Disease
gini = 0.49 samples = 7 value = [4, 3] class = No Disease
gini = 0.494 samples = 9 value = [4, 5] class = Disease
thal <= 1.5 gini = 0.111 samples = 17 value = [16, 1] class = No Disease
gini = 0.346 samples = 9 value = [7, 2] class = No Disease
gini = 0.0 samples = 43 value = [43, 0] class = No Disease
thalach <= 152.5 gini = 0.088 samples = 65 value = [3, 62] class = Disease
age <= 50.5 gini = 0.408 samples = 14 value = [4, 10] class = Disease
gini = 0.0 samples = 4 value = [4, 0] class = No Disease
sex <= 0.5 gini = 0.411 samples = 38 value = [11, 27] class = Disease
gini = 0.0 samples = 18 value = [0, 18] class = Disease
gini = 0.5 samples = 2 value = [1, 1] class = No Disease
gini = 0.444 samples = 3 value = [2, 1] class = No Disease
gini = 0.0 samples = 14 value = [14, 0] class = No Disease
sex <= 0.5 gini = 0.337 samples = 14 value = [3, 11] class = Disease
gini = 0.0 samples = 51 value = [0, 51] class = Disease
gini = 0.48 samples = 5 value = [3, 2] class = No Disease
gini = 0.198 samples = 9 value = [1, 8] class = Disease
age <= 62.5 gini = 0.188 samples = 19 value = [2, 17] class = Disease
chol <= 245.5 gini = 0.499 samples = 19 value = [9, 10] class = Disease
gini = 0.0 samples = 6 value = [0, 6] class = Disease
gini = 0.469 samples = 8 value = [3, 5] class = Disease
gini = 0.444 samples = 6 value = [2, 4] class = Disease
gini = 0.0 samples = 13 value = [0, 13] class = Disease
gini = 0.298 samples = 11 value = [2, 9] class = Disease
gini = 0.219 samples = 8 value = [7, 1] class = No Disease

5.2.1 Gini or entropy

```
# define the parameter grid
param_grid_metric = {
    'max_depth': list(range(1, 9)) + [None],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4],
    # adding the criterion parameter to the grid search
    'criterion': ['gini', 'entropy']
}

# create a grid search object
grid_search_metric = GridSearchCV(estimator=tree, param_grid=param_grid_metric, cv=5, n_jobs=
# fit the grid search to the training data
grid_search_metric.fit(X_train, y_train)
# print the best parameters
print("Best parameters found: ", grid_search_metric.best_params_)
```

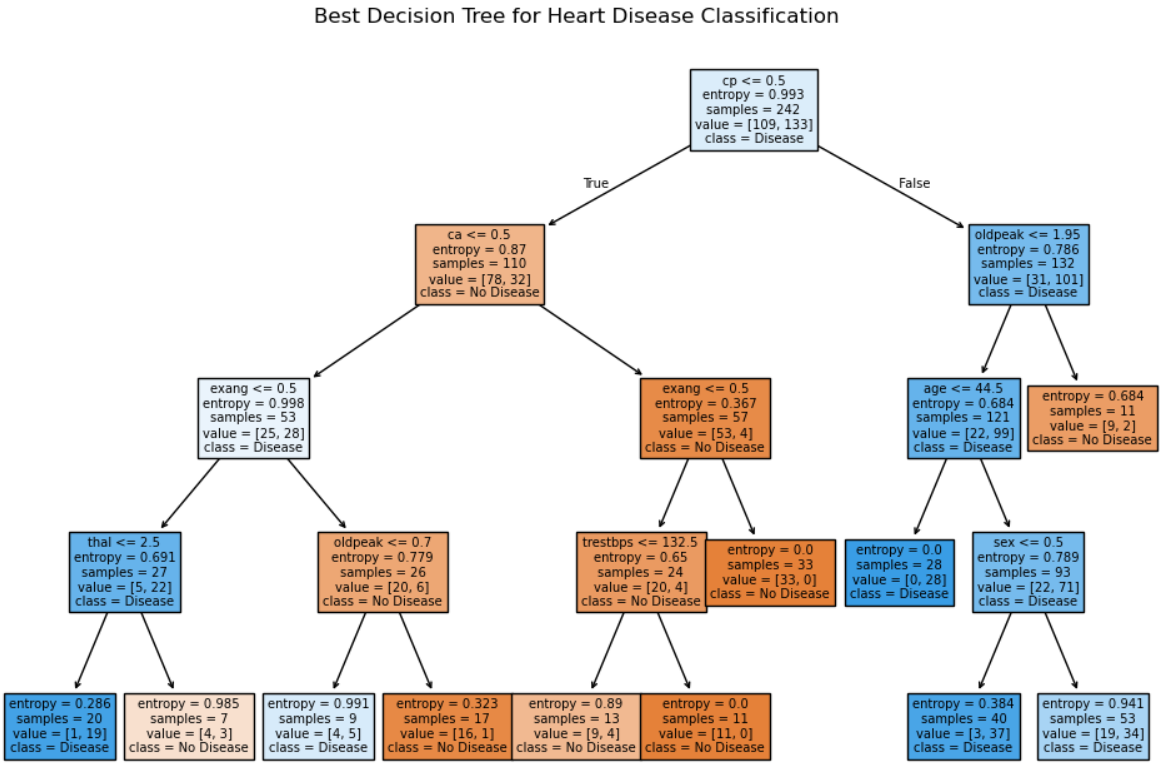
```
# print the best score
print("Best score: ", grid_search_metric.best_score_)
# get the best estimator
best_tree = grid_search_metric.best_estimator_

# make predictions on the test data with the best estimator
y_pred_best = best_tree.predict(X_test)
# calculate the accuracy of the best estimator
best_accuracy = accuracy_score(y_test, y_pred_best)
print(f'Best Test Accuracy: {best_accuracy:.2f}')
```

```
Fitting 5 folds for each of 270 candidates, totalling 1350 fits
Best parameters found:  {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_
Best score:  0.7811224489795918
Best Test Accuracy: 0.85
```

```
# print out the best tree depth and number of leaves
best_num_leaves = best_tree.get_n_leaves()
print(f'Best Number of leaves: {best_num_leaves}')

best_tree_depth = best_tree.get_depth()
print(f'Best Depth of the tree: {best_tree_depth}')
```

```
Best Number of leaves: 11
Best Depth of the tree: 4
```

```
# plot the best decision tree
plt.figure(figsize=(12, 8))
plot_tree(best_tree, filled=True, feature_names=X.columns, class_names=['No Disease', 'Diseas
plt.title('Best Decision Tree for Heart Disease Classification')
plt.show()
```



Both criteria aim to minimize impurity in splits, in pratice, they often lead to comparable performance in decision trees, even though their mathematical forulations differ

- **Gini**: Faster to compute (no logarithms) and often used for large datasets. so it is a good default. May produce slightly more complex trees.
- **Entropy**: Slower but aligns with information theory. Prefers splits that balance node sizes, leading to more interpretable trees.

5.3 Post-pruning: Cost complexity pruning

Post-pruning, on the other hand, allows the decision tree to grow to its full extent and then prunes it back to reduce complexity. This approach first builds a complete tree and then removes or collapses branches that don't significantly contribute to the model's performance. One common post-pruning technique is called Cost-Complexity Pruning.

5.3.1 step 1: calculate the cost complexity pruning path

```python
path = tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

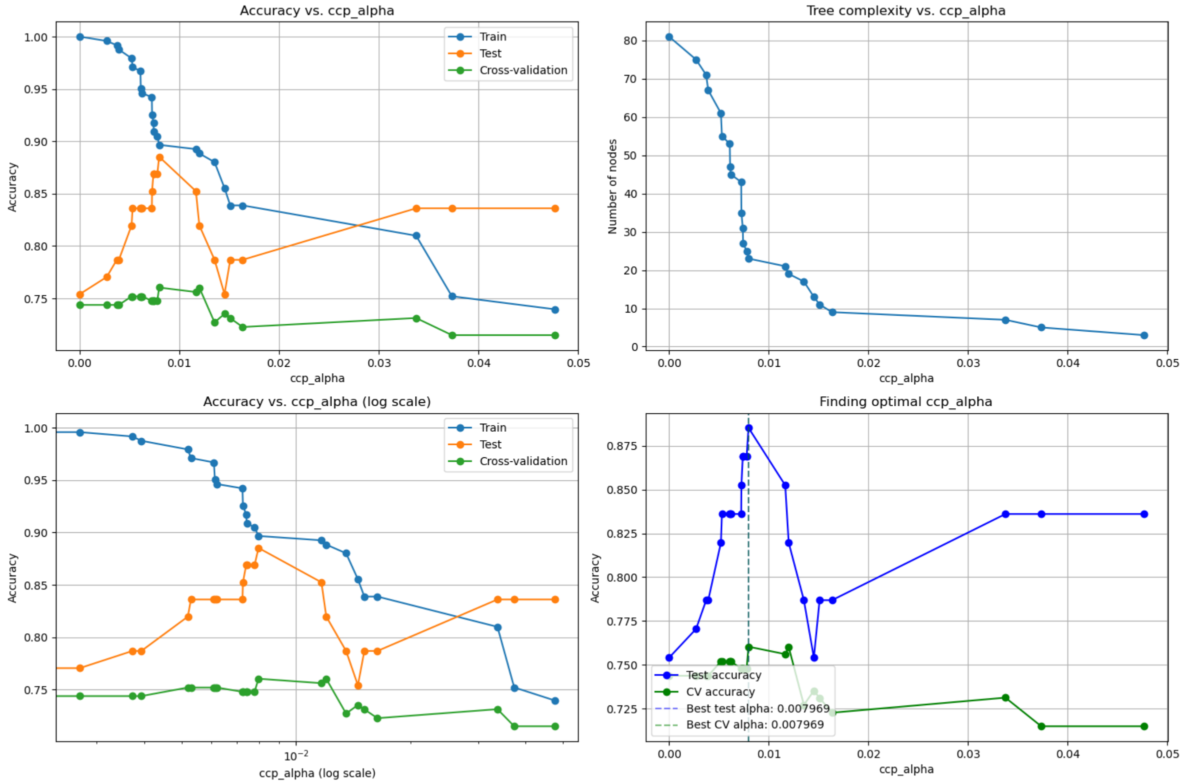
5.3.2 step 2: Create trees with different ccp_alpha values and evaluate their performance

```python
# We'll skip the last alpha which would produce a single-node tree
alphas = ccp_alphas[:-1]

# Create empty lists to store the results
train_scores = []
test_scores = []
cv_scores = []
node_counts = []

# For each alpha value, fit a tree and evaluate
for alpha in alphas:
    # Create and train the model
    clf = DecisionTreeClassifier(ccp_alpha=alpha, random_state=42)
    clf.fit(X_train, y_train)

    # Record scores
    train_scores.append(accuracy_score(y_train, clf.predict(X_train)))
    test_scores.append(accuracy_score(y_test, clf.predict(X_test)))

    # Cross-validation score for robustness
    cv_score = cross_val_score(clf, X_train, y_train, cv=5).mean()
    cv_scores.append(cv_score)

    # Record tree complexity
    node_counts.append(clf.tree_.node_count)
```

5.3.3 Step 3: Visualize the results

```
# Step 3: Visualize the results
fig, ax = plt.subplots(2, 2, figsize=(15, 10))

# Plot accuracy vs alpha
ax[0, 0].plot(alphas, train_scores, marker='o', label='Train')
ax[0, 0].plot(alphas, test_scores, marker='o', label='Test')
ax[0, 0].plot(alphas, cv_scores, marker='o', label='Cross-validation')
ax[0, 0].set_xlabel('ccp_alpha')
ax[0, 0].set_ylabel('Accuracy')
ax[0, 0].set_title('Accuracy vs. ccp_alpha')
ax[0, 0].legend()
ax[0, 0].grid(True)

# Plot number of nodes vs alpha
ax[0, 1].plot(alphas, node_counts, marker='o')
ax[0, 1].set_xlabel('ccp_alpha')
ax[0, 1].set_ylabel('Number of nodes')
ax[0, 1].set_title('Tree complexity vs. ccp_alpha')
ax[0, 1].grid(True)

# Log scale for better visualization of small alpha values
ax[1, 0].plot(alphas, train_scores, marker='o', label='Train')
ax[1, 0].plot(alphas, test_scores, marker='o', label='Test')
ax[1, 0].plot(alphas, cv_scores, marker='o', label='Cross-validation')
ax[1, 0].set_xlabel('ccp_alpha (log scale)')
ax[1, 0].set_ylabel('Accuracy')
ax[1, 0].set_title('Accuracy vs. ccp_alpha (log scale)')
ax[1, 0].set_xscale('log')
ax[1, 0].legend()
ax[1, 0].grid(True)

# Find best alpha based on test score
best_test_idx = np.argmax(test_scores)
best_test_alpha = alphas[best_test_idx]
best_test_acc = test_scores[best_test_idx]

# Find best alpha based on CV score (more robust)
best_cv_idx = np.argmax(cv_scores)
best_cv_alpha = alphas[best_cv_idx]
best_cv_acc = cv_scores[best_cv_idx]
```

```
# Plot highlighting best points
ax[1, 1].plot(alphas, test_scores, 'b-', marker='o', label='Test accuracy')
ax[1, 1].plot(alphas, cv_scores, 'g-', marker='o', label='CV accuracy')
ax[1, 1].axvline(x=best_test_alpha, color='blue', linestyle='--', alpha=0.5,
                label=f'Best test alpha: {best_test_alpha:.6f}')
ax[1, 1].axvline(x=best_cv_alpha, color='green', linestyle='--', alpha=0.5,
                label=f'Best CV alpha: {best_cv_alpha:.6f}')
ax[1, 1].set_xlabel('ccp_alpha')
ax[1, 1].set_ylabel('Accuracy')
ax[1, 1].set_title('Finding optimal ccp_alpha')
ax[1, 1].legend(loc='lower left')
ax[1, 1].grid(True)

plt.tight_layout()
plt.show()

# Print the optimal alpha values and corresponding metrics
print(f"Best alpha based on test score: {best_test_alpha:.6f} (Accuracy: {best_test_acc:.4f}
print(f"Best alpha based on CV score: {best_cv_alpha:.6f} (Accuracy: {best_cv_acc:.4f}, Nodes
```

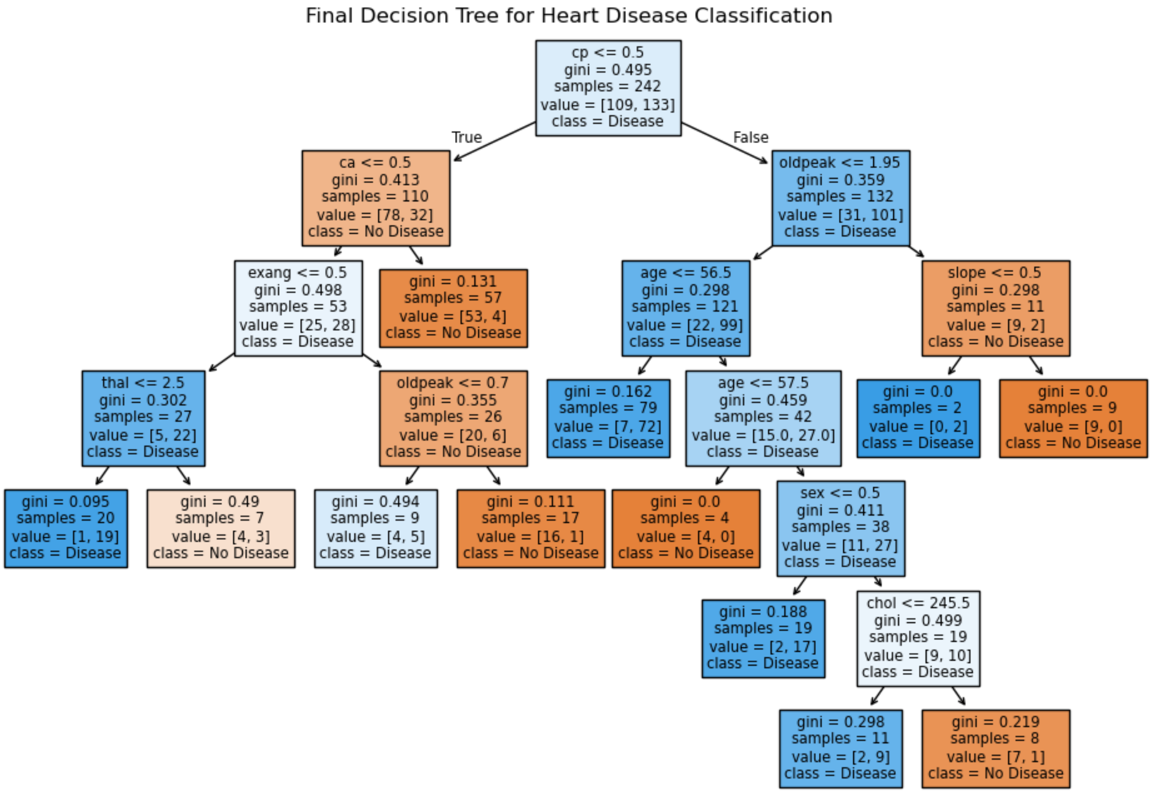


5.3.4 Step 4: Create the final model with the optimal alpha

```
# Using CV-based alpha as it's more robust against overfitting
final_model = DecisionTreeClassifier(ccp_alpha=best_cv_alpha, random_state=42)
final_model.fit(X_train, y_train)

# Evaluate the final model
train_acc = accuracy_score(y_train, final_model.predict(X_train))
test_acc = accuracy_score(y_test, final_model.predict(X_test))

print(f"\nFinal model performance:")
print(f"Training accuracy: {train_acc:.4f}")
print(f"Test accuracy: {test_acc:.4f}")
print(f"Tree nodes: {final_model.tree_.node_count}")
print(f"Tree depth: {final_model.get_depth()}")
```

```
Best alpha based on test score: 0.007969 (Accuracy: 0.8852, Nodes: 23)
Best alpha based on CV score: 0.007969 (Accuracy: 0.7604, Nodes: 23)

Final model performance:
Training accuracy: 0.8967
Test accuracy: 0.8852
Tree nodes: 23
Tree depth: 6
```

```
# plot the final decision tree
plt.figure(figsize=(12, 8))
plot_tree(final_model, filled=True, feature_names=X.columns, class_names=['No Disease', 'Dise
plt.title('Final Decision Tree for Heart Disease Classification')
plt.show()
```



Post-pruning can potentially create more optimal trees, as it considers the entire tree structure before making pruning decisions. However, it can be more computationally expensive.

Both approaches aim to find a balance between model complexity and performance, with the goal of creating a model that generalizes well to unseen data. The choice between pre-pruning and post-pruning (or a combination of both) often depends on the specific dataset, the problem at hand, and of course, computational resources available.

5.4 Feature Importance in Decision Trees

Decision tree algorithms, such as Classification and Regression Trees (CART), compute **feature importance** scores based on how much each feature contributes to reducing the splitting criterion (e.g., **Gini impurity** or **entropy**).

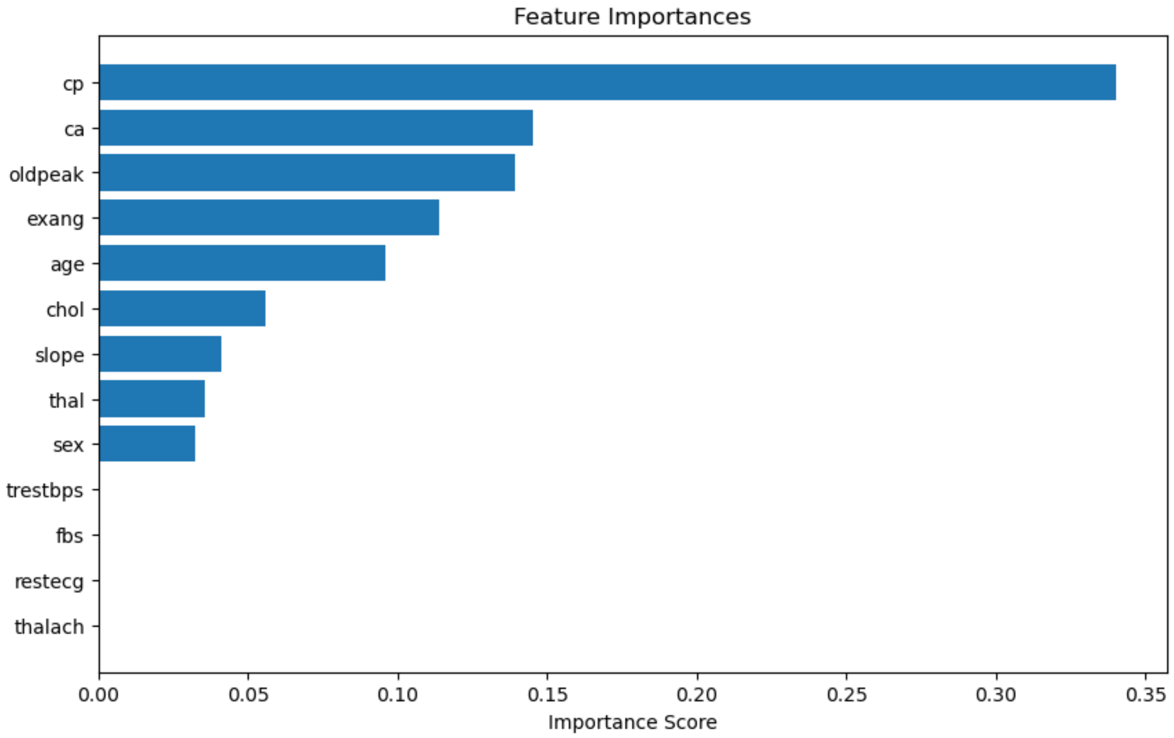
This methodology extends naturally to **ensemble models** like **Random Forests** and **Gradient Boosting**, which average feature importance across all trees in the ensemble.

Once a model is trained, the relative importance of each feature can be accessed using the `.feature_importances_` attribute. These scores indicate how valuable each feature was in constructing the decision rules that led to the model's predictions.

```
importances  = final_model.feature_importances_

# Create a DataFrame for easier visualization
feature_importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

# Display the DataFrame
feature_importance_df
```

	Feature	Importance
2	cp	0.340102
11	ca	0.145273
9	oldpeak	0.139508
8	exang	0.113871
0	age	0.095885
4	chol	0.056089
10	slope	0.041242
12	thal	0.035531
1	sex	0.032499
3	trestbps	0.000000
5	fbs	0.000000
6	restecg	0.000000
7	thalach	0.000000

```
plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'])
plt.xlabel("Importance Score")
plt.title("Feature Importances")
plt.gca().invert_yaxis()  # Most important feature at the top
plt.show()
```



5.5 Using Bagging to Combat Overfitting

Bagging, short for **Bootstrap Aggregating**, is an effective way to reduce overfitting in decision trees. It works by training multiple decision trees—each on a different bootstrap sample of the data—and then aggregating their predictions.

Each individual decision tree is a **weak learner** and prone to overfitting, but by combining many such trees, bagging produces a **strong learner** with **lower variance** and improved generalization performance.

The number of trees to include in the ensemble is specified by the `n_estimators` hyperparameter.

```
# using tree bagging to fit the data

from sklearn.ensemble import BaggingClassifier

bagging = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100, random_state=0)
bagging.fit(X_train, y_train)
```

```
y_pred_train = bagging.predict(X_train)
cm = pd.DataFrame(confusion_matrix(y_train, y_pred_train).T, index=['No', 'Yes'], columns=['
print(cm)

y_pred = bagging.predict(X_test)
cm = pd.DataFrame(confusion_matrix(y_test, y_pred).T, index=['No', 'Yes'], columns=['No', 'Y

print(cm)

#print out the accuracy on test set and training set
print("Train Accuracy is ", accuracy_score(y_train,y_pred_train)*100)
print("Test Accuracy is ", accuracy_score(y_test,y_pred)*100)
```

```
      No  Yes
No   109    0
Yes    0  133
     No  Yes
No   25    5
Yes   4   27
Train Accuracy is  100.0
Test Accuracy is  85.24590163934425
```

5.5.1 Takeaway:

Bagging reduces variance but does not affect bias.
By averaging the predictions of multiple high-variance models (like decision trees), bagging stabilizes the model and improves generalization, while the overall bias remains unchanged.

6 Bagging

Read section 8.2.1 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

6.1 Bagging

Bagging, short for **Bootstrap Aggregating**, is an effective way to reduce overfitting in decision trees. It works by training multiple decision trees—each on a different bootstrap sample of the data—and then aggregating their predictions.

Each individual decision tree is a **weak learner** and prone to overfitting, but by combining many such trees, bagging produces a **strong learner** with **lower variance** and improved generalization performance.

The number of trees to include in the ensemble is specified by the `n_estimators` hyperparameter.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.35)

# import the decision tree regressor
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier, plot_tree, export_gra
from sklearn.ensemble import BaggingRegressor,BaggingClassifier

# split the dataset into training and testing sets
from sklearn.model_selection import train_test_split


from sklearn.model_selection import cross_val_score, GridSearchCV, cross_val_predict, KFold
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer
from sklearn.metrics import root_mean_squared_error, r2_score, make_scorer, accuracy_score
```

6.2 Bagging regression trees

Let's revisit the same dataset used for building a single regression tree and explore whether we can further improve its performance using bagging

```
# Load the dataset
car = pd.read_csv('Datasets/car.csv')
car.head()
```

	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	vw	Beetle	2014	Manual	55457	Diesel	30	65.3266	1.6	7490
1	vauxhall	GTC	2017	Manual	15630	Petrol	145	47.2049	1.4	10998
2	merc	G Class	2012	Automatic	43000	Diesel	570	25.1172	3.0	44990
3	audi	RS5	2019	Automatic	10	Petrol	145	30.5593	2.9	51990
4	merc	X-CLASS	2018	Automatic	14000	Diesel	240	35.7168	2.3	28990

```
X = car.drop(columns=['price'])
y = car['price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# extract the categorical columns and put them in a list
categorical_feature = X.select_dtypes(include=['object']).columns.tolist()

# extract the numerical columns and put them in a list
numerical_feature = X.select_dtypes(include=['int64', 'float64']).columns.tolist()
```

```
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)

X_train_encoded = encoder.fit_transform(X_train[categorical_feature])
X_test_encoded = encoder.transform(X_test[categorical_feature])

# Convert the encoded features back to DataFrame
```

```
X_train_encoded_df = pd.DataFrame(X_train_encoded, columns=encoder.get_feature_names_out(cate
X_test_encoded_df = pd.DataFrame(X_test_encoded, columns=encoder.get_feature_names_out(catego

# Concatenate the encoded features with the original numerical features
X_train_final = pd.concat([X_train_encoded_df, X_train[numerical_feature].reset_index(drop=T
X_test_final = pd.concat([X_test_encoded_df, X_test[numerical_feature].reset_index(drop=True)
```

Below is the best model obtained by tuning the hyperparameters of a single decision tree.

```
# fit the decision tree regressor
tree_reg = DecisionTreeRegressor(max_depth=None, min_samples_leaf=1, min_samples_split=5, ran
tree_reg.fit(X_train_final, y_train)

# make predictions on the test set
y_pred = tree_reg.predict(X_test_final)
# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# print the RMSE and R^2 score, keep the decimal points to 2
print("Test RMSE:", round(rmse, 2))
print("Test R^2 score:", round(r2, 2))
```

```
Test RMSE: 4726.17
Test R^2 score: 0.92
```

Next, let's apply bagging to the optimized decision tree using 10 base estimators (default setting) and evaluate whether the performance improves.

```
#Bagging the results of 10 decision trees to predict car price
bagging_reg = BaggingRegressor(estimator=tree_reg, random_state=1,
                        n_jobs=-1).fit(X_train_final, y_train)

# make predictions on the test set
y_pred_bagging = bagging_reg.predict(X_test_final)

# calculate the RMSE and R^2 score
rmse_bagging = root_mean_squared_error(y_test, y_pred_bagging)
r2_bagging = r2_score(y_test, y_pred_bagging)

print("Test RMSE with Bagging pruned trees:", round(rmse_bagging, 2))
print("Test R^2 score with Bagging pruned trees:", round(r2_bagging, 2))


# training RMSE and R^2 score
y_pred_train_bagging = bagging_reg.predict(X_train_final)

# calculate the RMSE and R^2 score
rmse_train_bagging = root_mean_squared_error(y_train, y_pred_train_bagging)
r2_train_bagging = r2_score(y_train, y_pred_train_bagging)

print("Train RMSE with Bagging pruned trees:", round(rmse_train_bagging, 2))
print("Train R^2 score with Bagging pruned trees:", round(r2_train_bagging, 2))
```

```
Test RMSE with Bagging pruned trees: 3806.7
Test R^2 score with Bagging pruned trees: 0.95
Train RMSE with Bagging pruned trees: 1868.7
Train R^2 score with Bagging pruned trees: 0.99
```

Compared to the optimized single decision tree, bagging with pruned trees significantly improves performance—reducing the RMSE from 4726.17 to 3806.7 and increasing the R^2 score from 0.92 to 0.95.

Since bagging helps reduce variance, and the default setting for a single decision tree allows it to grow to full depth—often overfitting the training data—can we simply use the default tree as the base estimator for bagging?

```
#Bagging the results of 10 decision trees with the default parameters to predict car price
bagging_reg = BaggingRegressor(random_state=1,
                        n_jobs=-1).fit(X_train_final, y_train)

# make predictions on the test set
y_pred_bagging = bagging_reg.predict(X_test_final)

# calculate the RMSE and R^2 score
rmse_bagging = root_mean_squared_error(y_test, y_pred_bagging)
r2_bagging = r2_score(y_test, y_pred_bagging)

print("Test RMSE with Bagging unpruned trees:", round(rmse_bagging, 2))
print("Test R^2 score with Bagging unpruned trees:", round(r2_bagging, 2))

# training RMSE and R^2 score
y_pred_train_bagging = bagging_reg.predict(X_train_final)
```

```
# calculate the RMSE and R^2 score
rmse_train_bagging = root_mean_squared_error(y_train, y_pred_train_bagging)
r2_train_bagging = r2_score(y_train, y_pred_train_bagging)

print("Train RMSE with Bagging unpruned trees:", round(rmse_train_bagging, 2))
print("Train R^2 score with Bagging unpruned trees:", round(r2_train_bagging, 2))
```

```
Test RMSE with Bagging unpruned trees: 3758.1
Test R^2 score with Bagging unpruned trees: 0.95
Train RMSE with Bagging unpruned trees: 1501.04
Train R^2 score with Bagging unpruned trees: 0.99
```

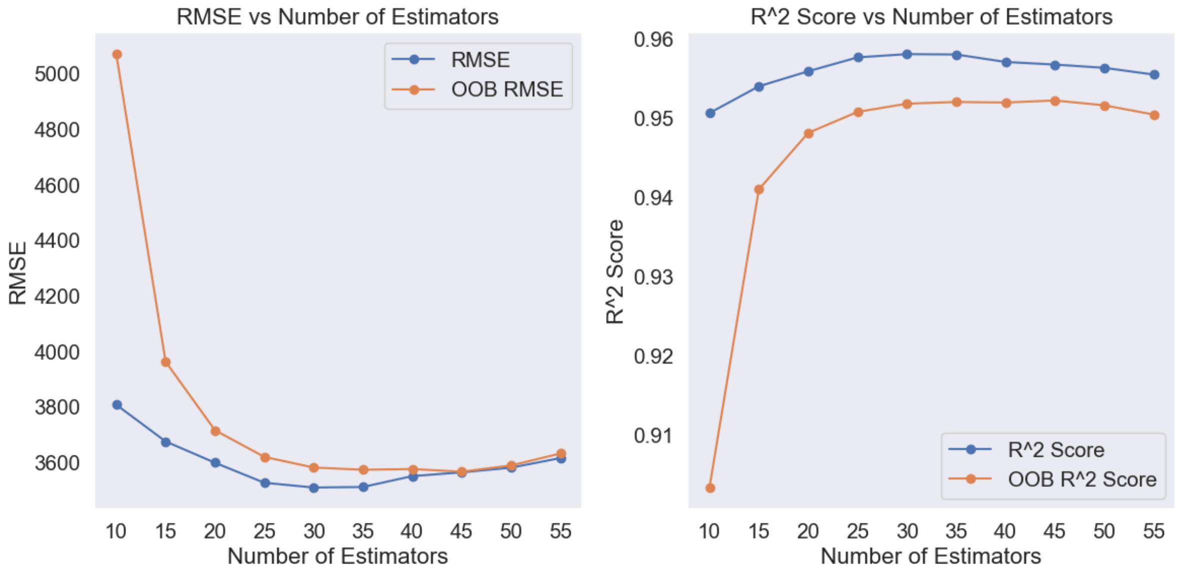
With the default setting, bagging using unpruned trees achieves an even lower RMSE of 3758.1 while maintaining the same R² score of 0.95.
This suggests that when using bagging, we don't necessarily need to tune the hyperparameters of the base decision tree—bagging itself effectively combats overfitting by reducing variance, much like hyperparameter tuning does.

6.3 Model Performance vs. Number of Trees

To better understand how the number of base estimators affects the performance of a bagging model, we evaluate the test RMSE and R² score across different numbers of trees.
This analysis helps us determine whether adding more trees continues to improve performance or if the model reaches a performance plateau.

```
# explore how the number of estimators affects the performance of the model, output both oob
n_estimators = [ 10, 15, 20, 25,30, 35, 40, 45, 50, 55]
rmse_scores = []
r2_scores = []
oob_scores = []
oob_rmse_scores = []
# iterate through the number of estimators and fit the model
for n in n_estimators:
    bagging_reg = BaggingRegressor(estimator=tree_reg, n_estimators=n, oob_score=True, rando
                        n_jobs=-1).fit(X_train_final, y_train)
    y_pred_bagging = bagging_reg.predict(X_test_final)
    rmse_scores.append(np.sqrt(np.mean((y_test - y_pred_bagging) ** 2)))
    r2_scores.append(r2_score(y_test, y_pred_bagging))
    oob_scores.append(bagging_reg.oob_score_)
    oob_rmse_scores.append(np.sqrt(np.mean((y_train - bagging_reg.oob_prediction_) ** 2)))

# plot the RMSE and R^2 scores against the number of estimators
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(n_estimators, rmse_scores, marker='o')
plt.plot(n_estimators, oob_rmse_scores, marker='o')
plt.legend(['RMSE', 'OOB RMSE'])
plt.title('RMSE vs Number of Estimators')
plt.xlabel('Number of Estimators')
plt.ylabel('RMSE')
plt.xticks(n_estimators)
plt.grid()
plt.subplot(1, 2, 2)
plt.plot(n_estimators, r2_scores, marker='o')
plt.plot(n_estimators, oob_scores, marker='o')
plt.legend(['R^2 Score', 'OOB R^2 Score'])
plt.title('R^2 Score vs Number of Estimators')
plt.xlabel('Number of Estimators')
plt.ylabel('R^2 Score')
plt.xticks(n_estimators)
plt.grid()
plt.tight_layout()
plt.show()
```

```
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\ensemble\_bagging.py:1315
  warn(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\ensemble\_bagging.py:1315
  warn(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\ensemble\_bagging.py:1315
  warn(
```



Quick Takeaway

- **Increasing the number of estimators improves performance initially**, as shown by the decreasing RMSE and increasing R^2 scores on both the test set and OOB estimates.
- **Performance stabilizes around 30–35 estimators**, beyond which further increases lead to minimal gains.
- **OOB estimates become more reliable** as the number of trees grows, with OOB scores closely tracking the test performance after around 30 estimators.
- **OOB RMSE is consistently higher** and **OOB R^2 is consistently lower** than their test counterparts when the ensemble is small, highlighting the **instability of OOB estimates with few trees**.
- Choosing `n_estimators` in the range of **30 to 40** offers a good trade-off between model accuracy and computational cost.

```python
# get the number of estimators that gives the best RMSE score
best_rmse_index = np.argmin(rmse_scores)
best_rmse_n_estimators = n_estimators[best_rmse_index]
best_rmse_value = rmse_scores[best_rmse_index]
print("Best number of estimators for RMSE:", best_rmse_n_estimators)
print("Best RMSE value:", round(best_rmse_value, 2))

# get the number of estimators that gives the best R^2 score
best_r2_index = np.argmax(r2_scores)
best_r2_n_estimators = n_estimators[best_r2_index]
best_r2_value = r2_scores[best_r2_index]
```

```
print("Best number of estimators for R^2 score:", best_r2_n_estimators)
print("Best R^2 score:", round(best_r2_value, 2))

# get the number of estimators that gives the best OOB RMSE score
best_oob_rmse_index = np.argmin(oob_rmse_scores)
best_oob_rmse_n_estimators = n_estimators[best_oob_rmse_index]
best_oob_rmse_value = oob_rmse_scores[best_oob_rmse_index]
print("Best number of estimators for OOB RMSE:", best_oob_rmse_n_estimators)
print("Best OOB RMSE value:", round(best_oob_rmse_value, 2))
```

```
Best number of estimators for RMSE: 30
Best RMSE value: 3508.31
Best number of estimators for R^2 score: 30
Best R^2 score: 0.96
Best number of estimators for OOB RMSE: 45
Best OOB RMSE value: 3565.29
```

6.4 Bagging Hyperparameter Tuning

To further improve the performance of our bagging model, we can tune key hyperparameters such as the number of base estimators (`n_estimators`), the maximum number of features considered at each split (`max_features`), and the size of each bootstrap sample (`max_samples`). By systematically exploring different combinations of these parameters, we aim to find the optimal settings that enhance the model's predictive accuracy while maintaining generalizability.

```
# hyperparameter tuning using GridSearchCV
param_grid = {
    'n_estimators': [10, 20, 30, 40, 50],
    'max_samples': [0.5, 0.75, 1.0],
    'max_features': [0.5, 0.75, 1.0],
    'bootstrap': [True, False],
    'bootstrap_features': [True, False]
}

bagging_reg_grid = BaggingRegressor(random_state=42, n_jobs=-1)
grid_search = GridSearchCV(bagging_reg_grid, param_grid, cv=5, scoring='neg_mean_squared_erro
grid_search.fit(X_train_final, y_train)

# get the best parameters and the best score
```

```
best_params = grid_search.best_params_
best_score = np.sqrt(-grid_search.best_score_)
print("Best parameters:", best_params)
print("Best RMSE cv score:", round(best_score, 2))

# make predictions on the test set using the best parameters
best_bagging_reg = grid_search.best_estimator_
y_pred_best_bagging = best_bagging_reg.predict(X_test_final)

# calculate the RMSE and R^2 score
rmse_best_bagging = root_mean_squared_error(y_test, y_pred_best_bagging)
r2_best_bagging = r2_score(y_test, y_pred_best_bagging)
print("Test RMSE with best Bagging model:", round(rmse_best_bagging, 2))
print("Test R^2 score with best Bagging model:", round(r2_best_bagging, 2))

# training RMSE and R^2 score
y_pred_train_best_bagging = best_bagging_reg.predict(X_train_final)

# calculate the RMSE and R^2 score
rmse_train_best_bagging = root_mean_squared_error(y_train, y_pred_train_best_bagging)
r2_train_best_bagging = r2_score(y_train, y_pred_train_best_bagging)
print("Train RMSE with best Bagging model:", round(rmse_train_best_bagging, 2))
print("Train R^2 score with best Bagging model:", round(r2_train_best_bagging, 2))
```

```
Best parameters: {'bootstrap': True, 'bootstrap_features': False, 'max_features': 0.75, 'max_
Best RMSE cv score: 3288.03
Test RMSE with best Bagging model: 3348.45
Test R^2 score with best Bagging model: 0.96
Train RMSE with best Bagging model: 1411.13
Train R^2 score with best Bagging model: 0.99
```

After simultaneously tuning multiple hyperparameters of the bagging model, including `n_estimators`, `max_features`, and `max_samples`, we achieved the best performance:

- **Test RMSE with best Bagging model:** 3348.45
- **Test R² score with best Bagging model:** 0.96

This demonstrates that careful tuning of bagging-specific parameters can lead to further improvements beyond using default or even optimized single decision trees.

6.5 Bagging Doesn't Reduce Bias

Bagging is an effective ensemble method for reducing **variance**, especially when using high-variance models like deep decision trees. However, it does **not reduce bias**.

To demonstrate this, we first fit a **shallow decision tree** with `max_depth=2`, which severely underfits the data due to its limited capacity. Then, we apply **bagging** using 10 such shallow trees (default setting) as base estimators.

Since each tree has high bias, the aggregated predictions from bagging still inherit that bias. In our results, both the single shallow tree and the bagged version yield similar (and poor) performance in terms of RMSE and R^2 on both the training and test sets.

This experiment shows that if your base model is too simple to capture the underlying patterns in the data, bagging will not help. To improve performance in such cases, we need to use more expressive base models or consider methods like **boosting**, which are better suited to reducing both bias and variance.

```
# Single shallow decision tree (underfitting)
shallow_tree = DecisionTreeRegressor(max_depth=2, random_state=1)
shallow_tree.fit(X_train_final, y_train)

# Predict and evaluate on test set
y_pred_single = shallow_tree.predict(X_test_final)
rmse_single = root_mean_squared_error(y_test, y_pred_single)
r2_single = r2_score(y_test, y_pred_single)

# Predict and evaluate on training set
y_pred_train_single = shallow_tree.predict(X_train_final)
rmse_train_single = root_mean_squared_error(y_train, y_pred_train_single)
r2_train_single = r2_score(y_train, y_pred_train_single)

print("Single Shallow Tree - Test RMSE:", round(rmse_single, 2))
print("Single Shallow Tree - Test R^2:", round(r2_single, 2))
print("Single Shallow Tree - Train RMSE:", round(rmse_train_single, 2))
print("Single Shallow Tree - Train R^2:", round(r2_train_single, 2))


# Bagging with 10 shallow trees
bagging_shallow = BaggingRegressor(estimator=DecisionTreeRegressor(max_depth=2, random_state=
                                    random_state=1,
                                    n_jobs=-1)
bagging_shallow.fit(X_train_final, y_train)
```

```
# Predict and evaluate on test set
y_pred_bagging = bagging_shallow.predict(X_test_final)
rmse_bagging = root_mean_squared_error(y_test, y_pred_bagging)
r2_bagging = r2_score(y_test, y_pred_bagging)

# Predict and evaluate on training set
y_pred_train_bagging = bagging_shallow.predict(X_train_final)
rmse_train_bagging = root_mean_squared_error(y_train, y_pred_train_bagging)
r2_train_bagging = r2_score(y_train, y_pred_train_bagging)

print("Bagged Shallow Trees - Test RMSE:", round(rmse_bagging, 2))
print("Bagged Shallow Trees - Test R^2:", round(r2_bagging, 2))
print("Bagged Shallow Trees - Train RMSE:", round(rmse_train_bagging, 2))
print("Bagged Shallow Trees - Train R^2:", round(r2_train_bagging, 2))
```

```
Single Shallow Tree - Test RMSE: 11084.97
Single Shallow Tree - Test R^2: 0.58
Single Shallow Tree - Train RMSE: 10314.42
Single Shallow Tree - Train R^2: 0.6
Bagged Shallow Trees - Test RMSE: 10894.92
Bagged Shallow Trees - Test R^2: 0.6
Bagged Shallow Trees - Train RMSE: 10114.07
Bagged Shallow Trees - Train R^2: 0.62
```

What you should observe:

- Both models will likely have low R^2 and high RMSE.
- Bagging may slightly reduce variance (narrower spread) but won't fix the high bias from using a shallow model.

6.6 Bagging classification trees

Let's revisit the same dataset used for building a single classification tree.
When using the default settings, the tree tends to **overfit** the data, as shown here.

In that notebook, we addressed the overfitting issue using both **pre-pruning** and **post-pruning** techniques.
Now, we'll explore an alternative approach—**bagging**—to reduce overfitting and improve model performance.

```
# load the dataset
heart_df  = pd.read_csv('datasets/heart_disease_classification.csv')
print(heart_df .shape)
heart_df .head()
```

```
(303, 14)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

```
# split the x and y data
X_cls = heart_df.drop(columns=['target'])
y_cls = heart_df.target

# split the data into train and test sets, add _cls to the variable names
X_train_cls, X_test_cls, y_train_cls, y_test_cls = train_test_split(X_cls, y_cls, test_size=
```

```
# using tree bagging to fit the data

bagging = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100, random_state=0)
bagging.fit(X_train_cls, y_train_cls)

y_pred_train_cls = bagging.predict(X_train_cls)
y_pred_cls = bagging.predict(X_test_cls)


#print out the accuracy on test set and training set
print("Train Accuracy is ", accuracy_score(y_train_cls,y_pred_train_cls)*100)
print("Test Accuracy is ", accuracy_score(y_test_cls,y_pred_cls)*100)
```

```
Train Accuracy is  100.0
Test Accuracy is  85.24590163934425
```

7 Bagging (addendum)

This notebook provides examples to:

1. Compare tuning bagging hyperparameters with OOB validation and k-fold cross-validation.
2. Compare bagging tuned models with untuned models.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score,train_test_split, KFold, GridSearchCV, Pa
RandomizedSearchCV, RepeatedKFold
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.ensemble import BaggingRegressor,BaggingClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_score
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score, mean_squared
from skopt import BayesSearchCV
from skopt.space import Real, Integer, Categorical
from skopt.plots import plot_convergence, plot_histogram, plot_objective
from IPython import display
import itertools as it

#Libraries for visualizing trees
from sklearn.tree import export_graphviz, export_text
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
import warnings
```

```
#Using the same datasets as in linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

1. Tree without tuning
2. Tree performance improves with tuning
3. Bagging tuned tree
4. Bagging untuned tree - better, how?
5. Tuning bagged model - OOB
6. Tuning bagged model - BayesSearchCV
7. warm start
8. Bagging KNN - no need to tune number of neighbors

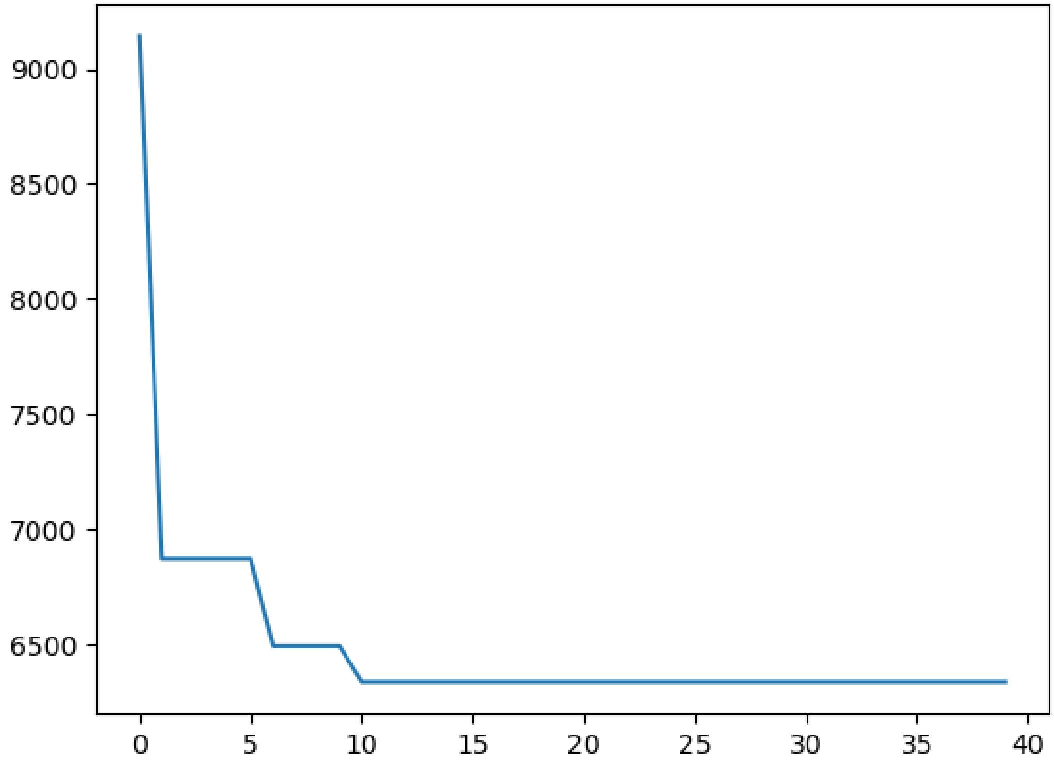
7.1 Tree without tuning

```
model = DecisionTreeRegressor()
cv = KFold(n_splits=5, shuffle=True, random_state=1)
-np.mean(cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv = cv))
```

```
7056.960817154941
```

```
param_grid = {'max_depth': Integer(2, 30)}
gcv = BayesSearchCV(model, search_spaces = param_grid, cv = cv, n_iter = 40, random_state = 1
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
['max_depth'] = [10] 6341.1481858990355
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=DecisionTreeRegressor(), n_iter=40, n_jobs=-1,
              random_state=10, scoring='neg_root_mean_squared_error',
              search_spaces={'max_depth': Integer(low=2, high=30, prior='uniform', transform=
```

7.2 Performance of tree improves with tuning

```
model = DecisionTreeRegressor(max_depth=10)
cv = KFold(n_splits=5, shuffle=True, random_state=1)
-np.mean(cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv = cv))
```

```
6442.494300778735
```

7.3 Bagging tuned trees

```
model = BaggingRegressor(DecisionTreeRegressor(max_depth = 10), oob_score=True, n_estimators
mean_squared_error(model.oob_prediction_, y, squared = False)
```

```
5354.357809020438
```

7.4 Bagging untuned trees

```
model = BaggingRegressor(DecisionTreeRegressor(), oob_score=True, n_estimators = 100).fit(X,
mean_squared_error(model.oob_prediction_, y, squared = False)
```

```
5248.720845665685
```

Why is bagging tuned trees worse than bagging untuned trees?

In the tuned tree here, the reduction in variance by controlling maximum depth resulted in an increas in bias of indivudual trees. Bagging trees only reduces the variance, but not the bias of the indivudal trees. Thus, bagging high bias models will result in a high-bias model, while bagging high variance models may result in a low variance model if the models are not highly correlated.

Bagging tuned models may provide a better performance as compared to bagging untuned models if the reduction in variance of the individual models is high enough to overshadow the increase in bias, and increase in pairwise correlation of the individual models.

7.5 Tuning bagged model - OOB

```
param_grid1 = {'max_samples': [0.25, 0.5, 0.75, 1.0],
             'max_features': [2, 3, 4],
             'bootstrap_features': [True, False]}
param_grid2 = {'max_samples': [0.25, 0.5, 0.75, 1.0],
             'max_features': [1],
              'bootstrap_features': [False]}
param_list1 = list(it.product(*[values for key, values in param_grid1.items()]))
param_list2 = list(it.product(*[values for key, values in param_grid2.items()]))
param_list = param_list1 + param_list2
```

```
oob_score_pr = []
for pr in param_list:
    model = BaggingRegressor(DecisionTreeRegressor(), max_samples=pr[0], max_features=pr[1],
                            bootstrap_features=pr[2], n_jobs = -1, oob_score=True, n_estimat
    oob_score_pr.append(mean_squared_error(model.oob_prediction_, y, squared=False))
```

What is the benefit of OOB validation to tune hyperparameters in bagging?

It is much cheaper than k-fold cross-validation, as only $1/k$ of the models are trained with OOB validation as compared to k-fold cross-validation. However, the cost of training individual models is lower in k-fold cross-validation as models are trained on a smaller dataset. Typically, OOB will be faster than k-fold cross-validation. The higher the value of k, the more faster OOB validation will be as compared to k-fold cross-validation.

7.6 Tuning without k-fold cross-validation

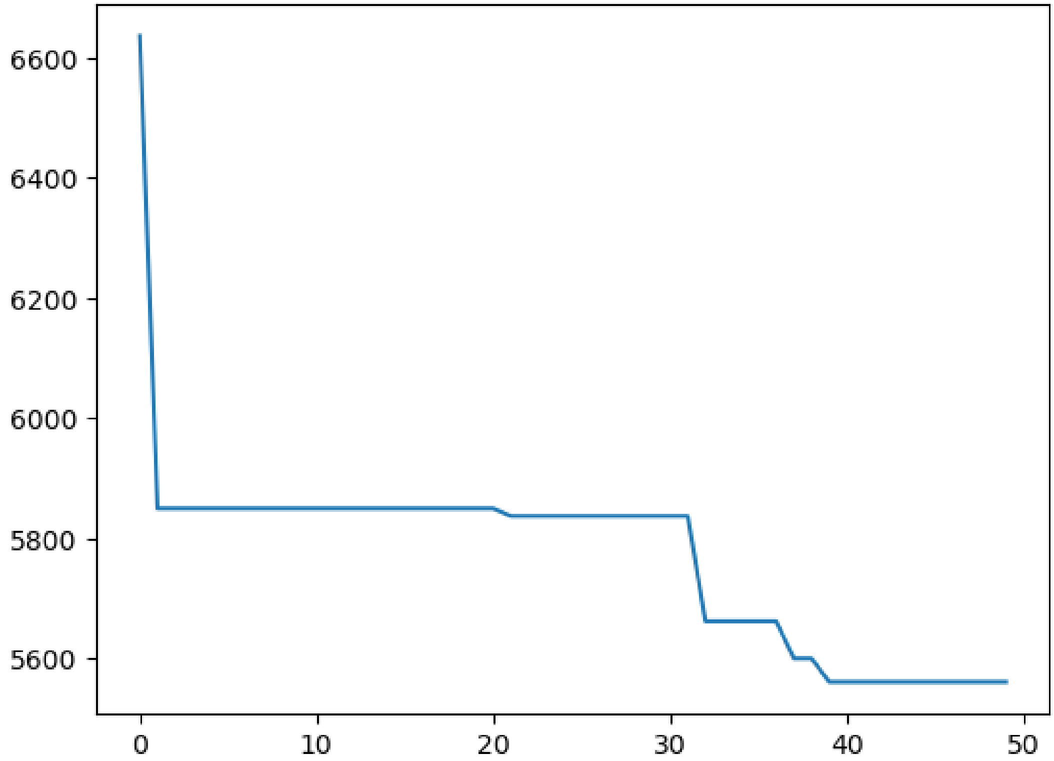
When hyperparameters can be tuned with OOB validation, what is the benefit of using k-fold cross-validation?

1. Hyperparameters cannot be tuned over continuous spaces with OOB validation.
2. OOB score is not computed if samping is done without replacement *(`bootstrap = False`)*. Thus, for tuning the `bootstrap` hyperparameter, k-fold cross-validation will need to be used.

```python
def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
```

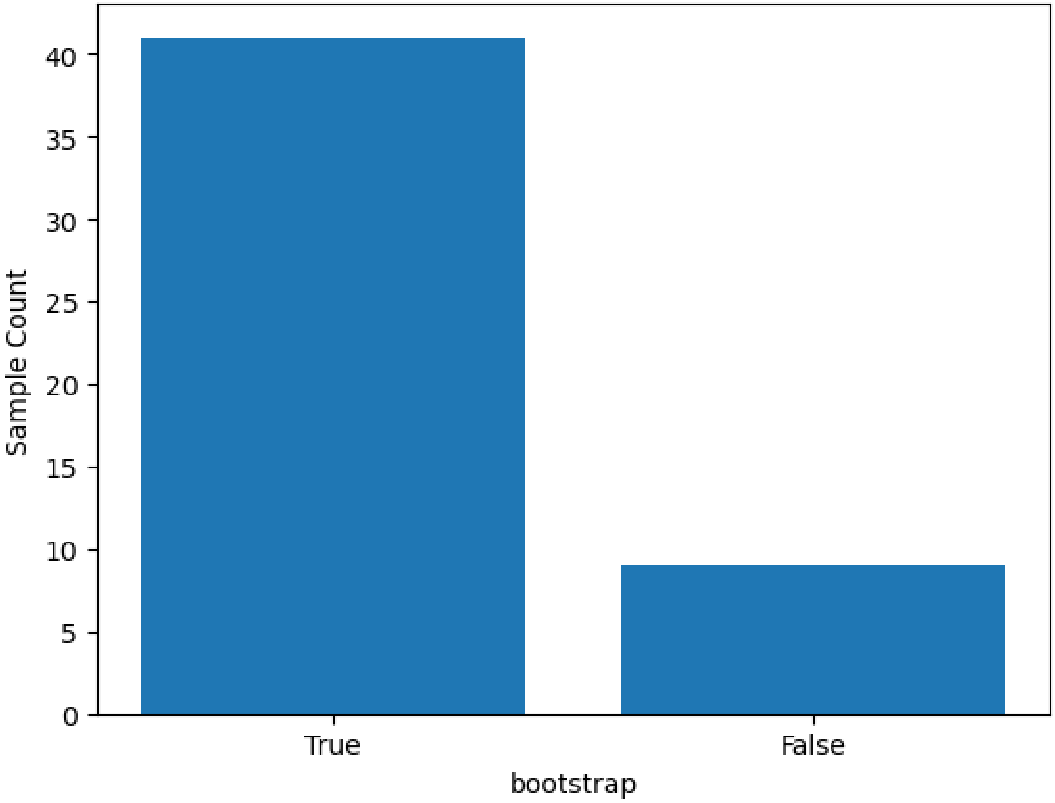
```python
param_grid = {'max_samples': Real(0.2, 1.0),
             'max_features': Integer(1, 4),
             'bootstrap_features': [True, False],
              'bootstrap': [True, False]}
gcv = BayesSearchCV(BaggingRegressor(DecisionTreeRegressor(), bootstrap=False),
                    search_spaces = param_grid, cv = cv, n_jobs = -1,
                  scoring='neg_root_mean_squared_error')

paras = list(gcv.search_spaces.keys())
paras.sort()

gcv.fit(X, y, callback=monitor)
```

```
['bootstrap', 'bootstrap_features', 'max_features', 'max_samples'] = [True, False, 4, 0.80613
```

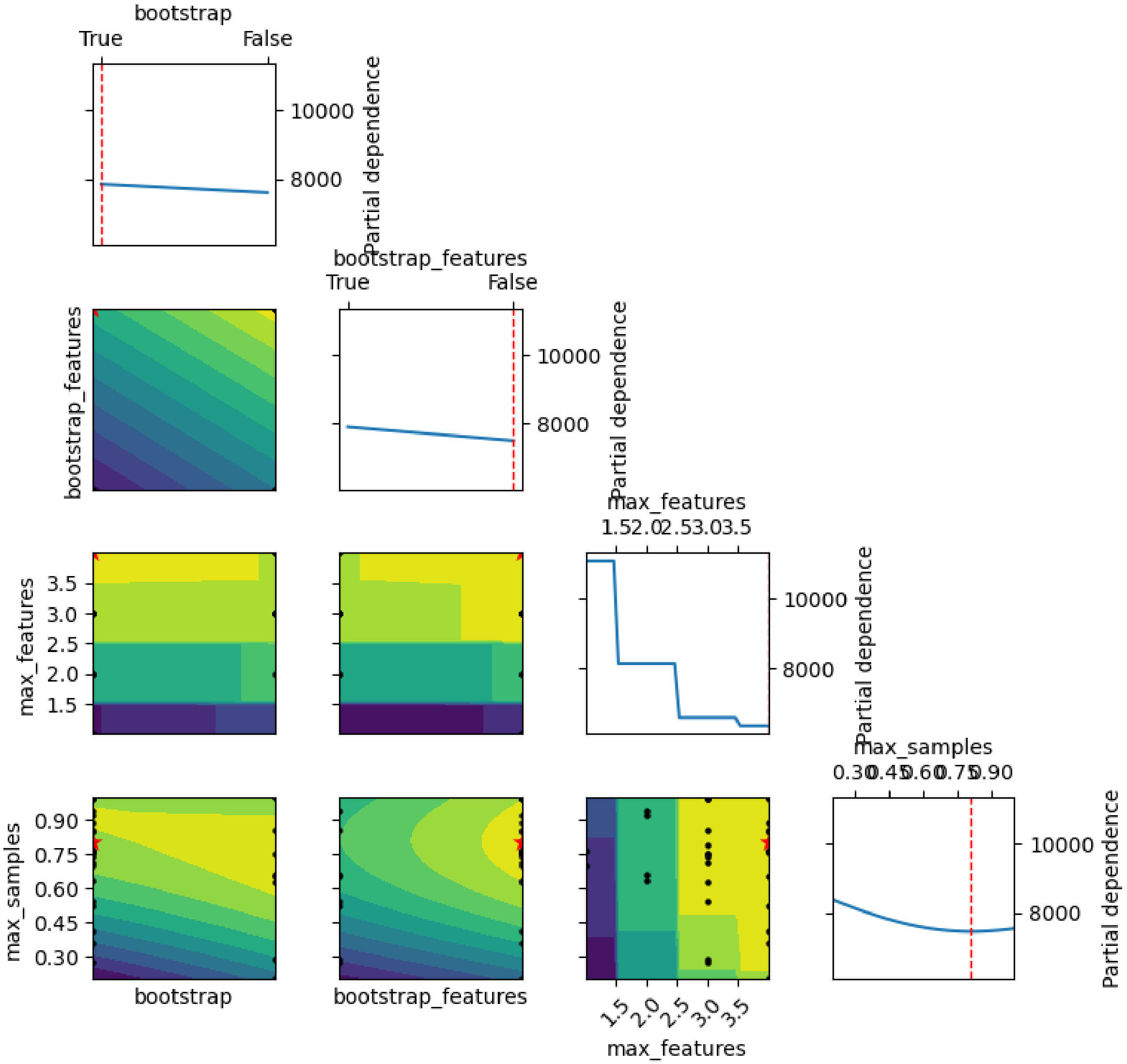


```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=BaggingRegressor(bootstrap=False,
                                         estimator=DecisionTreeRegressor()),
              n_jobs=-1, scoring='neg_root_mean_squared_error',
              search_spaces={'bootstrap': [True, False],
                             'bootstrap_features': [True, False],
                             'max_features': Integer(low=1, high=4, prior='uniform', transfo
                             'max_samples': Real(low=0.2, high=1.0, prior='uniform', transfo
```

```
plot_histogram(gcv.optimizer_results_[0],0)
```



```
plot_objective(gcv.optimizer_results_[0])
```



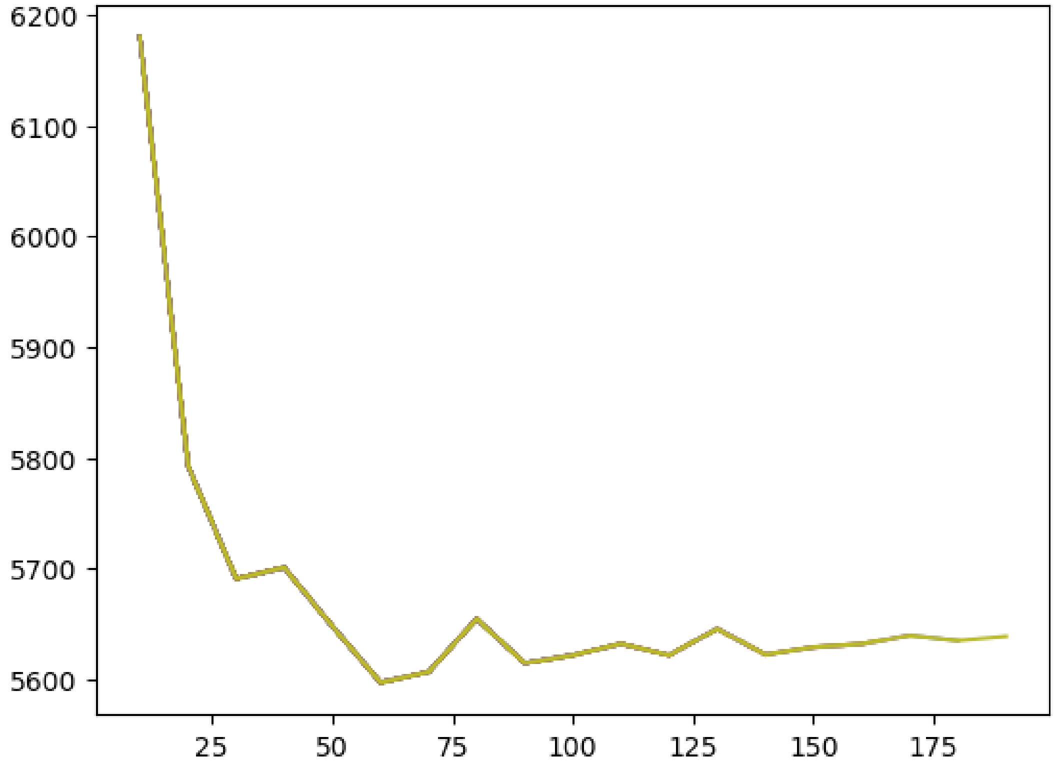
7.7 warm start

What is the purpose of warm_start?

The purpose of `warm_start` is to avoid developing trees from scratch, and incrementally add trees to monitor the validation error. However, note that OOB score is not computed with `warm_start`. Thus, a validation set approach will need to be adopted to tune number of trees.

A cheaper approach to tune number of estimators is to just use trial and error, and stop increasing once the cross-validation error / OOB error / validation set error stabilizes.

```
model = BaggingRegressor(DecisionTreeRegressor(), oob_score=False, n_estimators = 5,
                        warm_start=True).fit(X, y)
rmse = []
for i in range(10, 200, 10):
    model.n_estimators = i
    model.fit(X, y)
    rmse.append(mean_squared_error(model.predict(Xtest), ytest, squared=False))
    sns.lineplot(x = range(10, i+1, 10), y = rmse)
```



7.8 Bagging KNN

Should we bag a tuned KNN model or an untuned one?

```
from sklearn.preprocessing import StandardScaler
```

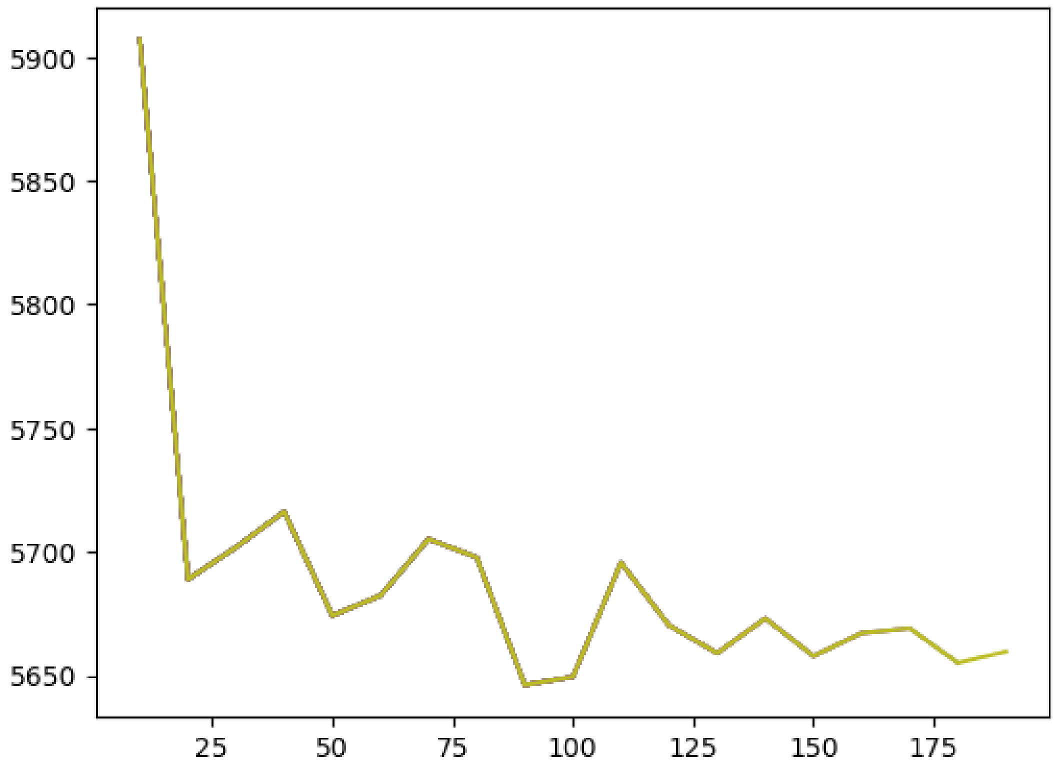
```
model = KNeighborsRegressor(n_neighbors=9) # optimal neigbors
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
-np.mean(cross_val_score((model), X_scaled, y, cv = cv,
                         scoring='neg_root_mean_squared_error', n_jobs = -1))
```

```
6972.997277781689
```

```
model = KNeighborsRegressor(n_neighbors=1)
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
-np.mean(cross_val_score(BaggingRegressor(model), X_scaled, y, cv = cv,
                         scoring='neg_root_mean_squared_error', n_jobs = -1))
```

```
6254.305462266355
```

```
model = BaggingRegressor(DecisionTreeRegressor(), n_estimators=5, warm_start=True)
model.fit(X, y)
rmse = []
for i in range(10, 200,10):
    model.n_estimators = i
    model.fit(X, y)
    rmse.append(mean_squared_error(model.predict(Xtest), ytest, squared=False))
    sns.lineplot(x = range(10, i + 1, 10), y = rmse)
```


5900
5850
5800
5750
5700
5650
25
50
75
100
125
150
175

8 Random Forest

Read section 8.2.2 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid
from sklearn.ensemble import BaggingRegressor,BaggingClassifier,RandomForestRegressor,Random
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_scor
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score

import itertools as it

#Libraries for visualizing trees
from sklearn.tree import export_graphviz
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
import warnings
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
```

```
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

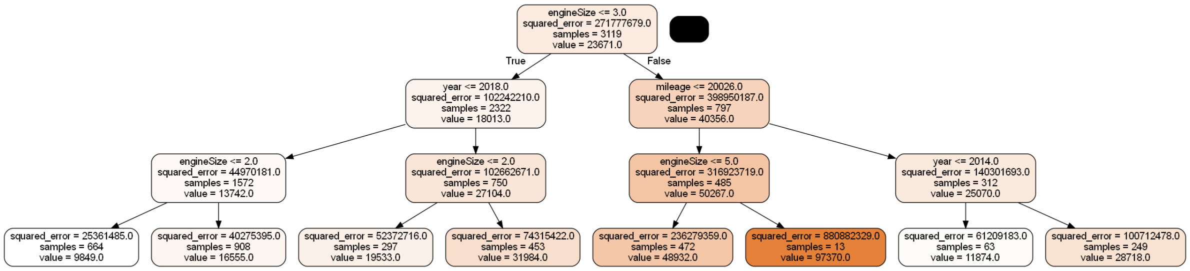
	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

Let us make a bunch of small trees with bagging, so that we can visualize and see if they are being dominated by a particular predictor or predictor(s).

```
#Bagging the results of 10 decision trees to predict car price
model = BaggingRegressor(estimator=DecisionTreeRegressor(max_depth=3), n_estimators=10, rando
                        n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 bagged trees, one at a tim
dot_data = StringIO()
export_graphviz(model.estimators_[0], out_file=dot_data,
                filled=True, rounded=True,
                feature_names =['mileage','mpg','year','engineSize'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



Each of the 10 bagged trees seems to be dominated by the `engineSize` predictor, thereby making the trees highly correlated. Average of highly correlated random variables has a higher variance than the average of lesser correlated random variables. Thus, highly correlated trees will tend to have a relatively high prediction variance despite averaging their predictions.

```
#Feature importance can be found by averaging the feature importance in individual trees
feature_importances = np.mean([
    tree.feature_importances_ for tree in model.estimators_
], axis=0)
feature_importances
```

```
array([0.13058631, 0.03965966, 0.22866077, 0.60109325])
```

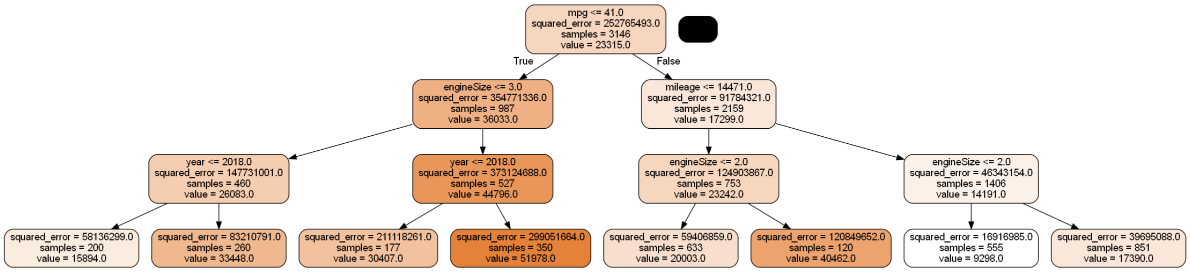
We can see that `engineSize` has the highest importance among predictors, supporting the visualization that it dominates the trees.

8.1 Random Forest for regression

Now, let us visualize small trees with the random forest algorithm to see if a predictor dominates all the trees.

```
#Averaging the results of 10 decision trees, while randomly considering sqrt(4)=2 predictors
#to split, to predict car price
model = RandomForestRegressor(n_estimators=10, random_state=1,max_features="sqrt",max_depth=
                        n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 random forest trees, one a
dot_data = StringIO()
export_graphviz(model.estimators_[4], out_file=dot_data,
                filled=True, rounded=True,
                feature_names =['mileage','mpg','year','engineSize'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



As two of the four predictors are randomly selected for splitting each node, `engineSize` no longer seems to dominate the trees. This will tend to reduce correlation among trees, thereby reducing the prediction variance, which in turn will tend to improve prediction accuracy.

```
#Averaging the results of 10 decision trees, while randomly considering sqrt(4)=2 predictors
#to split, to predict car price
model = RandomForestRegressor(n_estimators=10, random_state=1,max_features="sqrt",
                        n_jobs=-1).fit(X, y)
```

```
model.feature_importances_
```

```
array([0.16370584, 0.35425511, 0.18552673, 0.29651232])
```

Note that the feature importance of `engineSize` is reduced in random forests (as compared to bagged trees), and it no longer dominates the trees.

```
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
5856.022395768459
```

The RMSE is similar to that obtained by bagging. We will discuss the comparison later.

8.1.1 Model accuracy vs number of trees

How does the model accuracy vary with the number of trees?

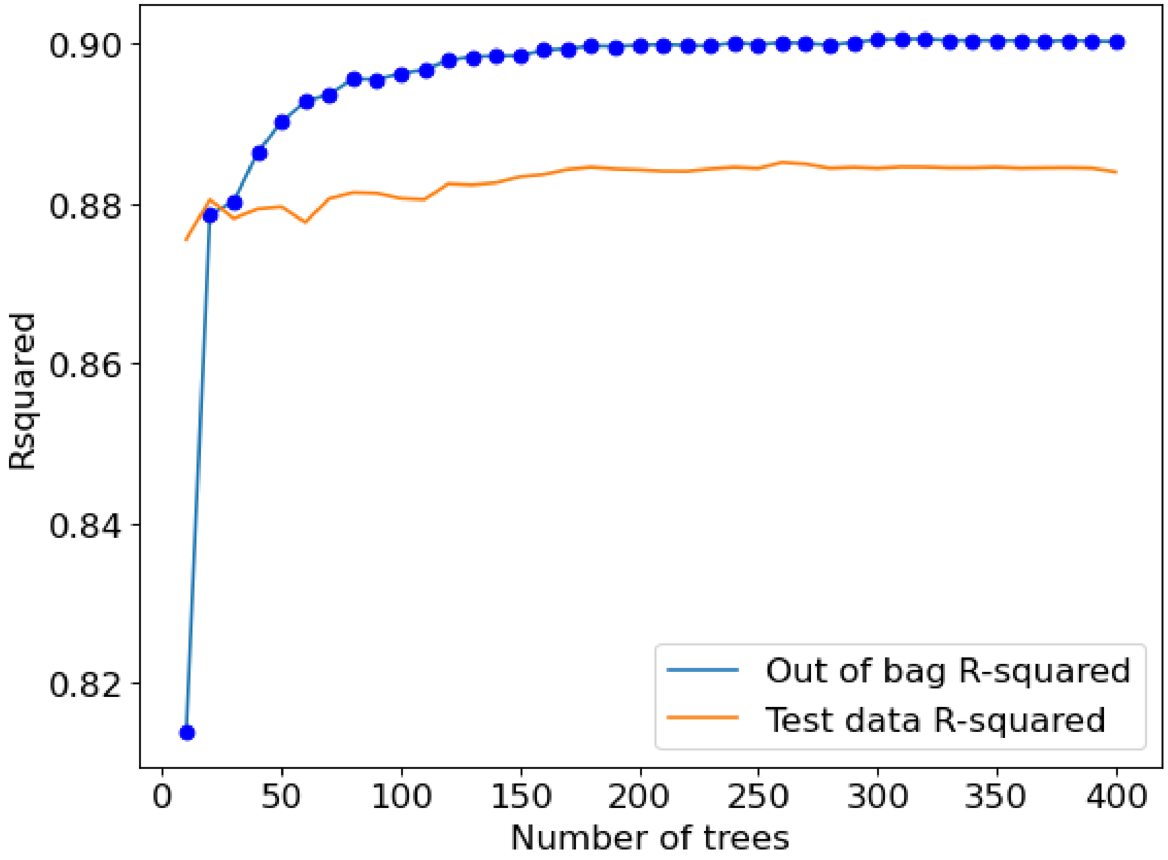
As we increase the number of trees, it will tend to reduce the variance of individual trees leading to a more accurate prediction.

```
#Finding model accuracy vs number of trees
warnings.filterwarnings("ignore")
oob_rsquared={};test_rsquared={};oob_rmse={};test_rmse = {}

for i in np.linspace(10,400,40,dtype=int):
    model = RandomForestRegressor(n_estimators=i, random_state=1,max_features="sqrt",
                        n_jobs=-1,oob_score=True).fit(X, y)
    oob_rsquared[i]=model.oob_score_  #Returns the out-of_bag R-squared of the model
    test_rsquared[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
    oob_rmse[i]=np.sqrt(mean_squared_error(model.oob_prediction_,y))
    test_rmse[i]=np.sqrt(mean_squared_error(model.predict(Xtest),ytest))

warnings.resetwarnings()

# The hidden warning is: "Some inputs do not have OOB scores. This probably means too few
# estimators were used to compute any reliable oob estimates." This warning will appear
# in case of small number of estimators. In such a case, some observations may be use
# by all the estimators, and their OOB score can't be computed
```

As we are ensemble only 10 trees in the first iteration, some of the observations are selected in every bootstrapped sample, and thus they don't have an out-of-bag error, which is producing the warning. For every observation to have an out-of-bag error, the number of trees must be sufficiently large.

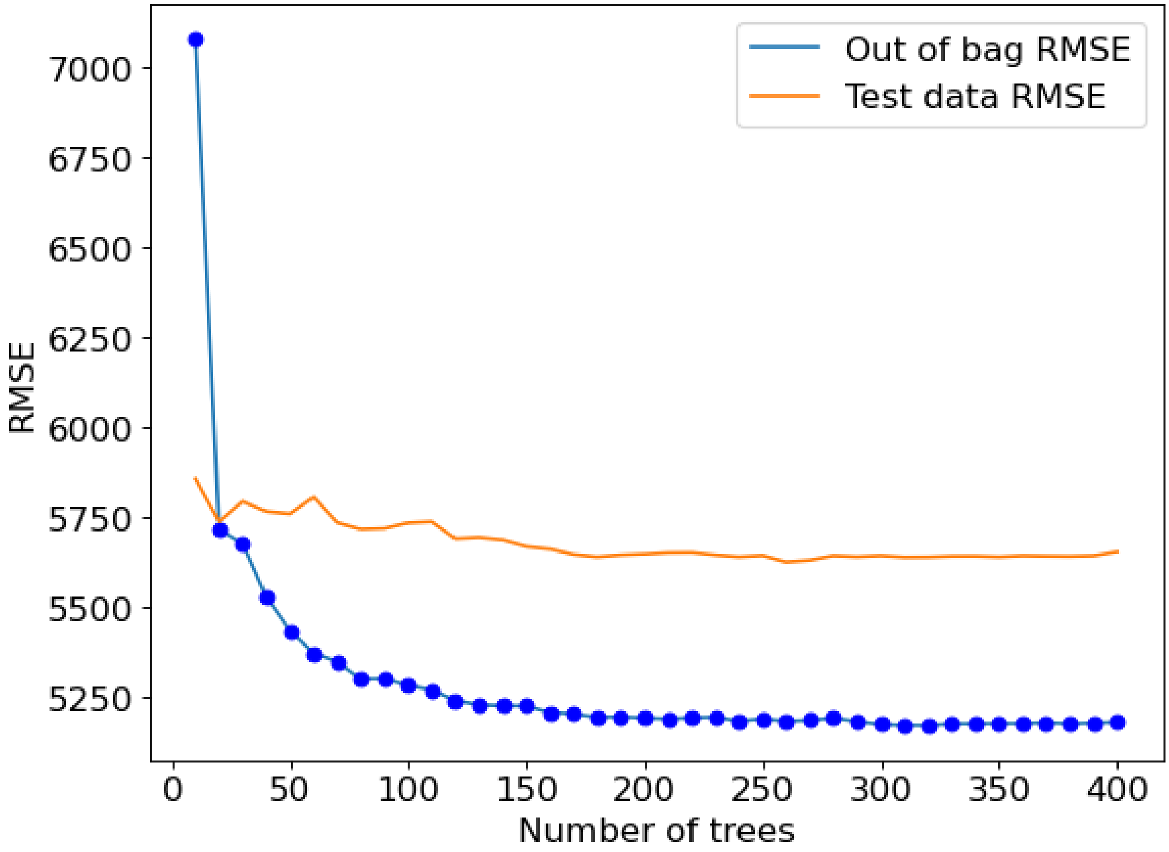
Let us visualize the out-of-bag (OOB) R-squared and R-squared on test data vs the number of trees.

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),label = 'Out of bag R-squared')
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),'o',color = 'blue')
plt.plot(test_rsquared.keys(),test_rsquared.values(), label = 'Test data R-squared')
plt.xlabel('Number of trees')
plt.ylabel('Rsquared')
plt.legend();
```



The out-of-bag R-squared initially increases, and then stabilizes after a certain number of trees (around 200 in this case). Note that increasing the number of trees further will not lead to overfitting. However, increasing the number of trees will increase the computations. Thus, the number of trees developed should be the number beyond which the R-squared stabilizes.

```
#Visualizing out-of-bag RMSE and test data RMSE
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rmse.keys(),oob_rmse.values(),label = 'Out of bag RMSE')
plt.plot(oob_rmse.keys(),oob_rmse.values(),'o',color = 'blue')
plt.plot(test_rmse.keys(),test_rmse.values(), label = 'Test data RMSE')
plt.xlabel('Number of trees')
plt.ylabel('RMSE')
plt.legend();
```



A similar trend can be seen by plotting out-of-bag RMSE and test RMSE. Note that RMSE is proportional to R-squared. You only need to visualize one of RMSE or R-squared to find the optimal number of trees.

```
#Bagging with 150 trees
model = RandomForestRegressor(n_estimators=200, random_state=1,max_features="sqrt",
                              oob_score=True,n_jobs=-1).fit(X, y)
```

```
#OOB R-squared
model.oob_score_
```

```
0.8998265006519903
```

```
#RMSE on test data
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
5647.195064555622
```

8.1.2 Tuning random forest

The Random forest object has options to set parameters such as depth, leaves, minimum number of observations in a leaf etc., for individual trees. These parameters are useful to prune a decision tree model consisting of a single tree, in order to avoid overfitting due to high variance of an unpruned tree.

Pruning individual trees in random forests is not likely to add much value, since averaging a sufficient number of unpruned trees reduces the variance of the trees, which enhances prediction accuracy. Pruning individual trees is unlikely to further reduce the prediction variance.

Here is a comment from page 596 of the The Elements of Statistical Learning that supports the above statement: *Segal (2004) demonstrates small gains in performance by controlling the depths of the individual trees grown in random forests. Our experience is that using full-grown trees seldom costs much, and results in one less tuning parameter.*

Below we attempt to optimize parameters that prune individual trees. However, as expected, it does not result in a substantial increase in prediction accuracy.

Also, note that we don't need to tune the number of trees in random forest with `GridSearchCV`. As we know the prediction accuracy will keep increasing with number of trees, we can tune the other hyperparameters with a constant value for the number of trees.

```
model.estimators_[0].get_n_leaves()
```

```
3086
```

```
model.estimators_[0].get_depth()
```

```
29
```

Coarse grid search

```
#Optimizing with OOB score takes half the time as compared to cross validation.
#The number of models developed with OOB score tuning is one-fifth of the number of models de
#5-fold cross validation
start_time = time.time()

n_samples = train.shape[0]
n_features = train.shape[1]

params = {'max_depth': [5, 10, 15, 20, 25, 30],
          'max_leaf_nodes':[600, 1200, 1800, 2400, 3000],
          'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))

oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                    n_estimators = 100, max_depth=pr[0],
                    max_leaf_nodes=pr[1], max_features=pr[2], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))
```

```
time taken =  1.230358862876892  minutes
Best params =  (15, 1800, 3)
Optimal OOB validation RMSE =  5243.408784594606
```

Finer grid search

Based on the coarse grid search, hyperparameters will be tuned in a finer grid around the optimal hyperparamter values obtained.

```
#Optimizing with OOB score takes half the time as compared to cross validation.
#The number of models developed with OOB score tuning is one-fifth of the number of models de
#5-fold cross validation
start_time = time.time()

n_samples = train.shape[0]
n_features = train.shape[1]

params = {'max_depth': [12, 15, 18],
          'max_leaf_nodes':[1600, 1800, 2000],
          'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))
```

```
oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                  n_estimators = 100, max_depth=pr[0], max_leaf_nodes=pr[1],
                        max_features=pr[2], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))
```

```
time taken =  0.4222299337387085  minutes
Best params =  (15, 1800, 3)
Best score =  5243.408784594606
```

```
#Model with optimal parameters
model = RandomForestRegressor(n_estimators = 100, random_state=1, max_leaf_nodes = 1800, max_
                        oob_score=True,n_jobs=-1, max_features=3).fit(X, y)
```

```
#RMSE on test data
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
5671.410705964455
```

Optimizing depth and leaves of individual trees didn't improve the prediction accuracy of the model. Important parameters to optimize in random forests will be the number of trees (`n_estimators`), and number of predictors considered at each split (`max_features`). However, sometimes individual pruning of trees may be useful. This may happen when the increase in bias in individual trees *(when pruned)* is lesser than the decrease in variance of the tree. However, if the pairwise correlation coefficient ρ of the trees increases by a certain extent on pruning, pruning may again be not useful.

```
#Tuning only n_estimators and max_features produces similar results
start_time = time.time()
params = {'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))
```

```
oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                      n_estimators = 100, max_features=pr[0], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))
```

```
time taken =  0.02856200933456421  minutes
Best params =  (3,)
Best score (R-squared) =  5252.291978670057
```

```
#Model with optimal parameters
model = RandomForestRegressor(n_estimators=100, random_state=1,
                        n_jobs=-1, max_features=3).fit(X, y)
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
5656.561522632323
```

Considering hyperparameters involving pruning, we observe a marginal decrease in the out-of-bag RMSE. Thus, other hyperparameters *(such as `max_features` and `max_samples`)* must be prioritized for tuning over hyperparameters involving pruning.

8.2 Random forest for classification

Random forest model to predict if a person has diabetes.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

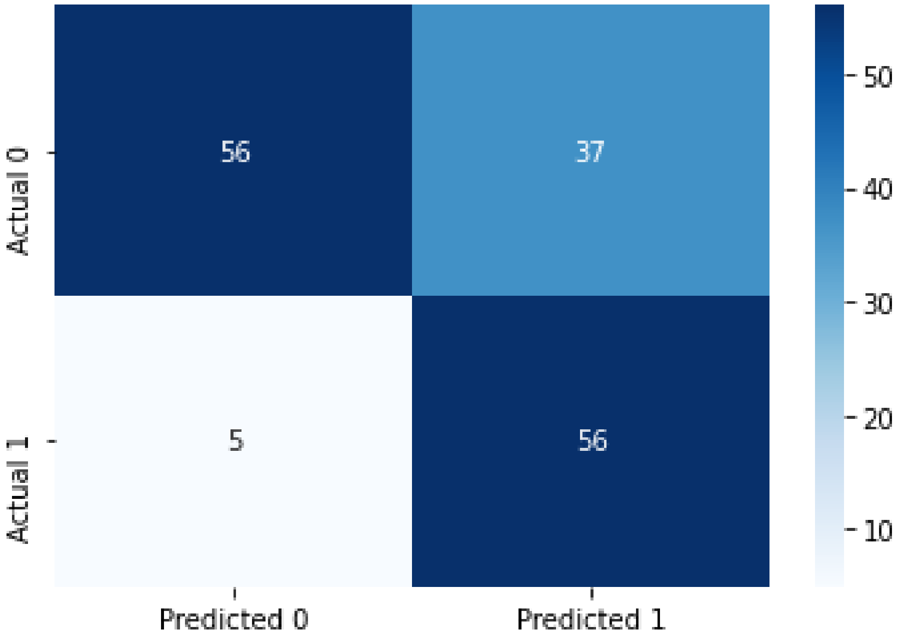
```
#Ensembling the results of 10 decision trees
model = RandomForestClassifier(n_estimators=200, random_state=1,max_features="sqrt",n_jobs=-
```

```
#Feature importance for Random forest
np.mean([tree.feature_importances_ for tree in model.estimators_],axis=0)
```

```
array([0.08380406, 0.25403736, 0.09000104, 0.07151063, 0.07733353,
       0.16976023, 0.12289303, 0.13066012])
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.23

y_pred_prob = model.predict_proba(Xtest)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is grea
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  72.72727272727273
ROC-AUC:  0.8744050766790058
Precision:  0.6021505376344086
Recall:  0.9180327868852459
```

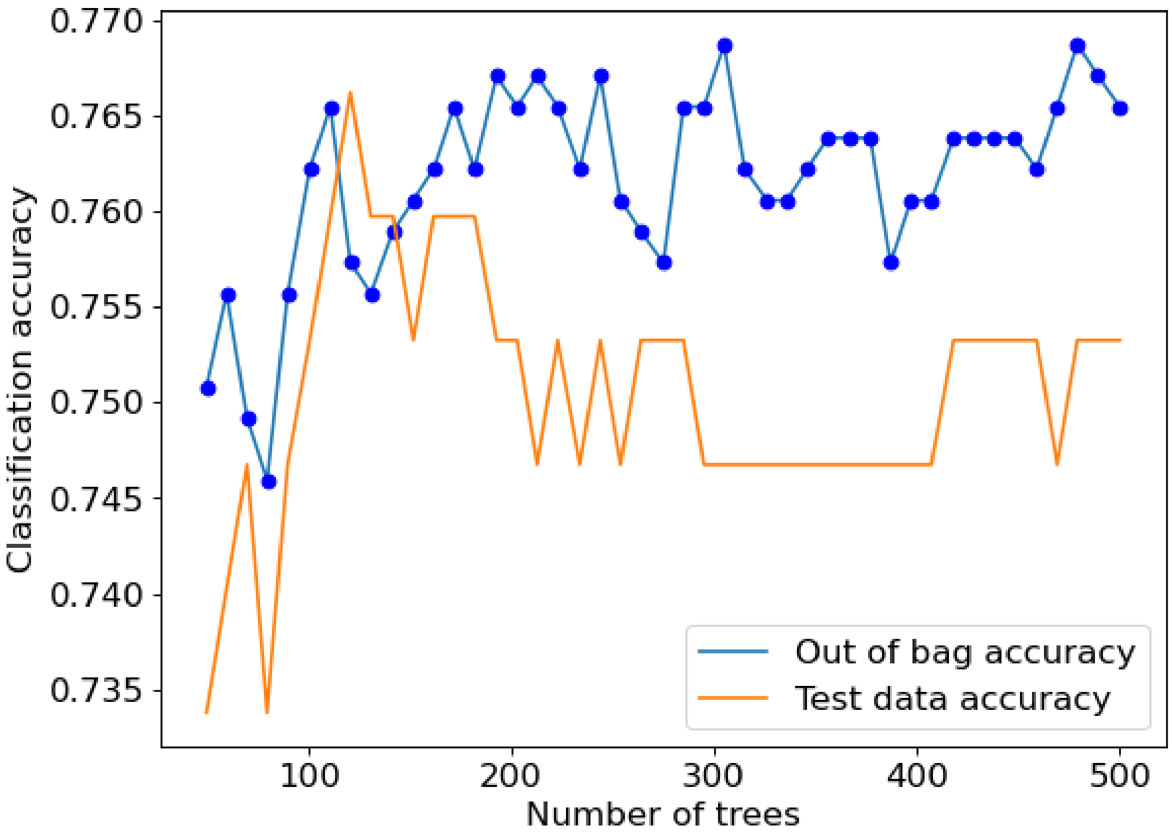


The model obtained above is similar to the one obtained by bagging. We'll discuss the comparison later.

8.2.1 Model accuracy vs number of trees

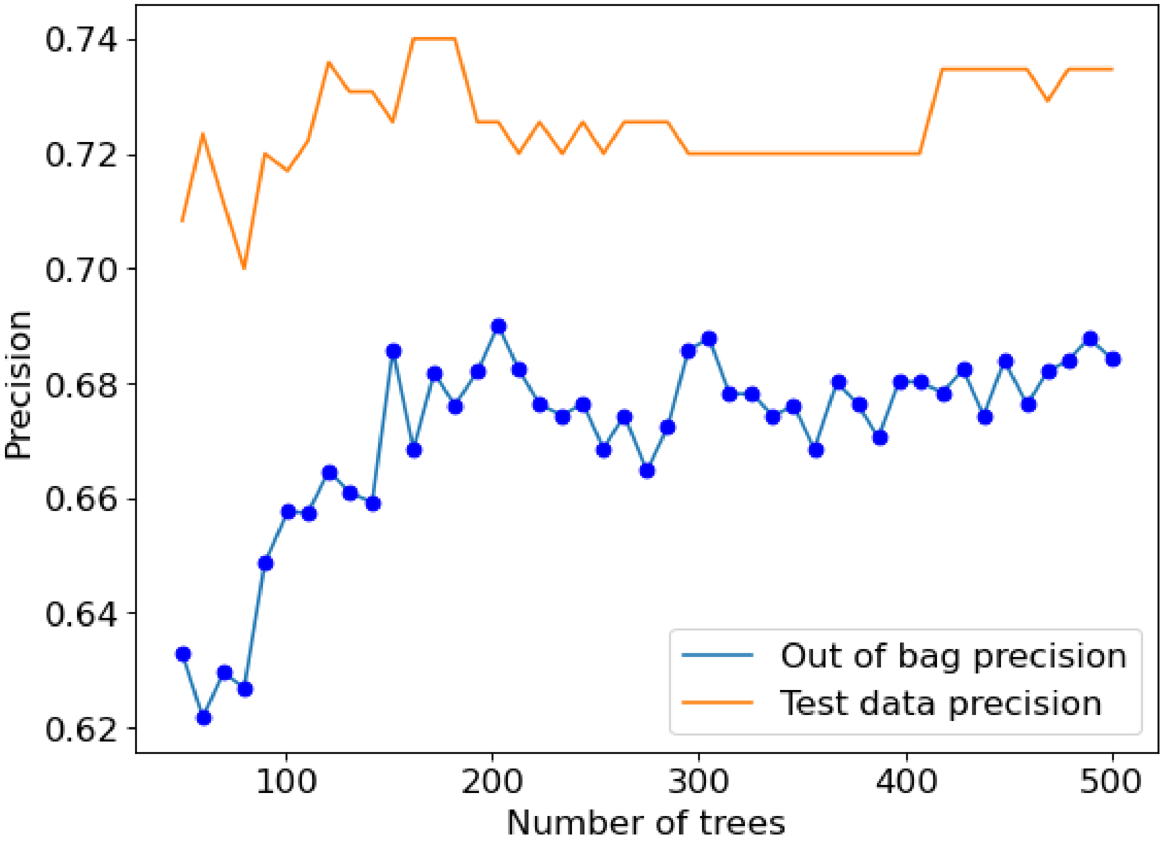
```
#Finding model accuracy vs number of trees
oob_accuracy={};test_accuracy={};oob_precision={}; test_precision = {}
for i in np.linspace(50,500,45,dtype=int):
    model = RandomForestClassifier(n_estimators=i, random_state=1,max_features="sqrt",n_jobs=
    oob_accuracy[i]=model.oob_score_  #Returns the out-of_bag R-squared of the model
    test_accuracy[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
    oob_pred = (model.oob_decision_function_[:,1]>=0.5).astype(int)
    oob_precision[i] = precision_score(y, oob_pred)
    test_pred = model.predict(Xtest)
    test_precision[i] = precision_score(ytest, test_pred)
```

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),label = 'Out of bag accuracy')
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),'o',color = 'blue')
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Test data accuracy')
plt.xlabel('Number of trees')
plt.ylabel('Classification accuracy')
plt.legend();
```



We can also plot other metrics of interest such as out-of-bag precision vs number of trees.

```
#Precision vs number of trees
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_precision.keys(),oob_precision.values(),label = 'Out of bag precision')
plt.plot(oob_precision.keys(),oob_precision.values(),'o',color = 'blue')
plt.plot(test_precision.keys(),test_precision.values(), label = 'Test data precision')

plt.xlabel('Number of trees')
plt.ylabel('Precision')
plt.legend();
```



8.2.2 Tuning random forest

Here we tune the number of predictors to be considered at each node for the split to maximize recall.

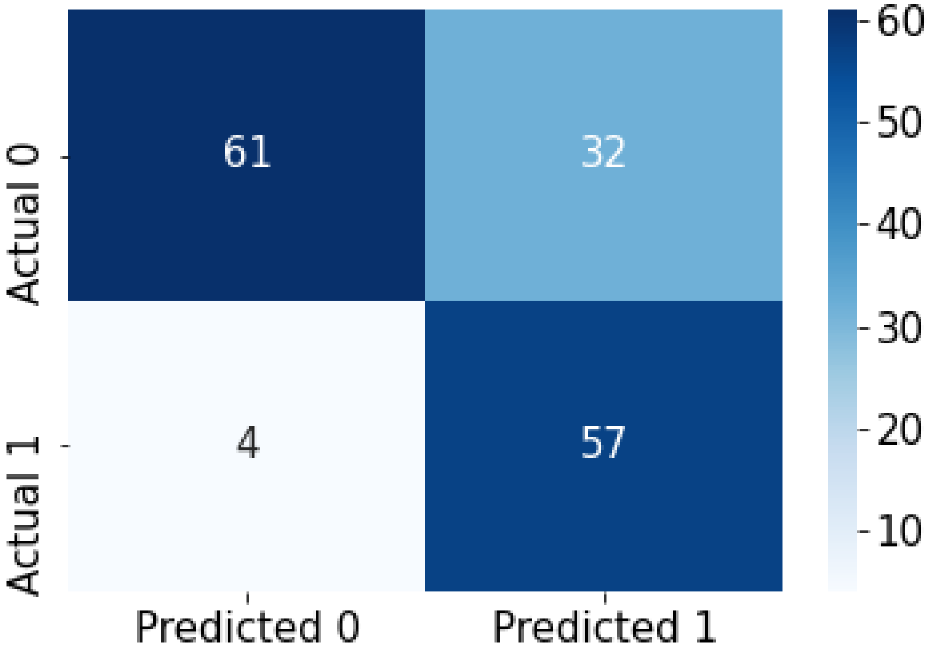
```
start_time = time.time()

params = {'n_estimators': [500],
          'max_features': range(1,9),
         }

param_list=list(it.product(*(params[Name] for Name in list(params.keys()))))
oob_recall = [0]*len(param_list)

i=0
for pr in param_list:
    model = RandomForestClassifier(random_state=1,oob_score=True,verbose=False,n_estimators =
                                  max_features=pr[1], n_jobs=-1).fit(X,y)

    oob_pred = (model.oob_decision_function_[:,1]>=0.5).astype(int)
    oob_recall[i] = recall_score(y, oob_pred)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max recall = ", np.max(oob_recall))
print("params= ", param_list[np.argmax(oob_recall)])
```

```
time taken =  0.08032723267873128  minutes
max recall =  0.5990338164251208
params=  (500, 8)
```

```
model = RandomForestClassifier(random_state=1,n_jobs=-1,max_features=8,n_estimators=500).fit
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.23

y_pred_prob = model.predict_proba(Xtest)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is grea
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  76.62337662337663
ROC-AUC:  0.8787237793054822
Precision:  0.6404494382022472
Recall:  0.9344262295081968
```



```
model.feature_importances_
```

```
array([0.069273  , 0.31211579, 0.08492953, 0.05225877, 0.06179047,
       0.17732674, 0.12342981, 0.1188759 ])
```

8.3 Random forest vs Bagging

We saw in the above examples that the performance of random forest was similar to that of bagged trees. This may happen in some cases including but not limited to:

1. All the predictors are more or less equally important, and the bagged trees are not highly correlated.

2. One of the predictors dominates the trees, resulting in highly correlated trees. However, each of the highly correlated trees have high prediction accuracy, leading to overall high prediction accuracy of the bagged trees despite the high correlation.

When can random forests perform poorly: When the number of variables is large, but the fraction of relevant variables small, random forests are likely to perform poorly with small m (fraction of predictors considered for each split). At each split the chance can be small that the relevant variables will be selected. - *Elements of Statistical Learning, page 596*.

However, in general, random forests are expected to decorrelate and improve the bagged trees.

Let us consider a classification example.

```
data = pd.read_csv('Heart.csv')
data.dropna(inplace = True)
data.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca
0	63	1	typical	145	233	1	2	150	0	2.3	3	0.0
1	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0
2	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0
3	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0
4	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0

In the above dataset, we wish to predict if a person has acquired heart disease (AHD = 'Yes'), based on their symptoms.

```
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy variables replacing the categorical variables
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)
X.head()
```

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
0	63	1	145	233	1	2	150	0	2.3	3	0.0	0
1	67	1	160	286	0	2	108	1	1.5	2	3.0	1
2	67	1	120	229	0	2	129	1	2.6	2	2.0	1
3	37	1	130	250	0	0	187	0	3.5	3	0.0	0

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
4	41	0	130	204	0	2	172	0	1.4	1	0.0	0

```
X.shape
```

```
(297, 18)
```

```
#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)
```

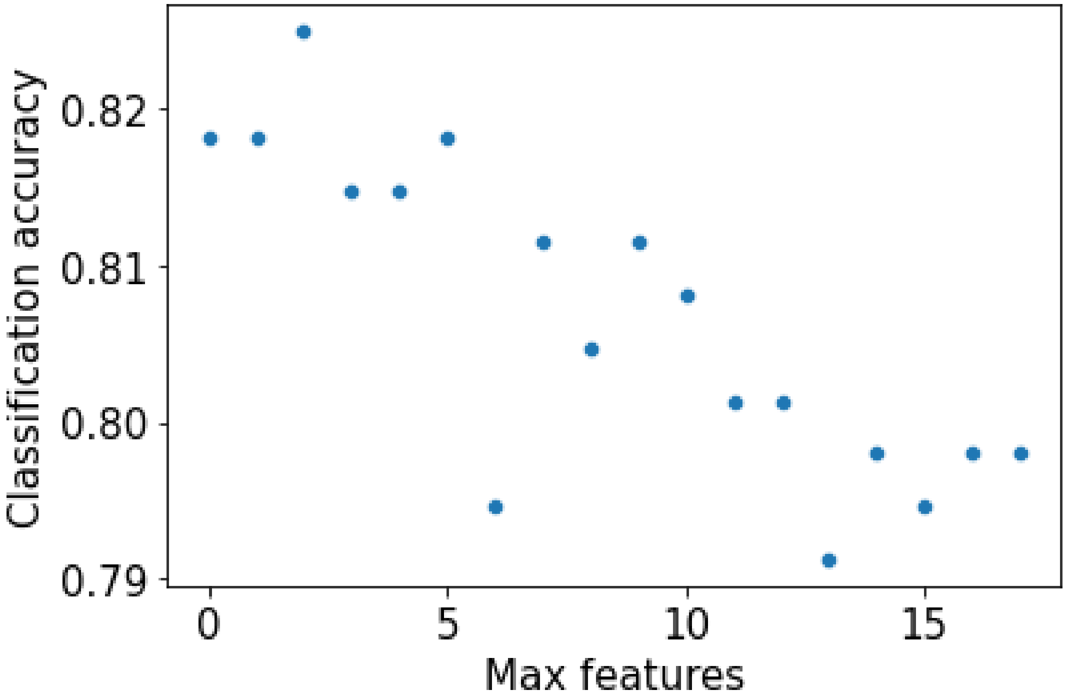
Tuning random forest

```
#Tuning the random forest parameters
start_time = time.time()

oob_score = {}

i=0
for pr in range(1,19):
    model = RandomForestClassifier(random_state=1,oob_score=True,verbose=False,n_estimators =
                                  max_features=pr, n_jobs=-1).fit(X,y)
    oob_score[i] = model.oob_score_
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max accuracy = ", np.max(list(oob_score.values())))
print("Best value of max_features= ", np.argmax(list(oob_score.values()))+1)
```

```
time taken =  0.21557459433873494  minutes
max accuracy =  0.8249158249158249
Best value of max_features=  3
```

```
sns.scatterplot(x = oob_score.keys(),y = oob_score.values())
plt.xlabel('Max features')
plt.ylabel('Classification accuracy')
```

```
Text(0, 0.5, 'Classification accuracy')
```



Note that as the value of `max_features` is increasing, the accuracy is decreasing. This is probably due to the trees getting correlated as we consider more predictors for each split.

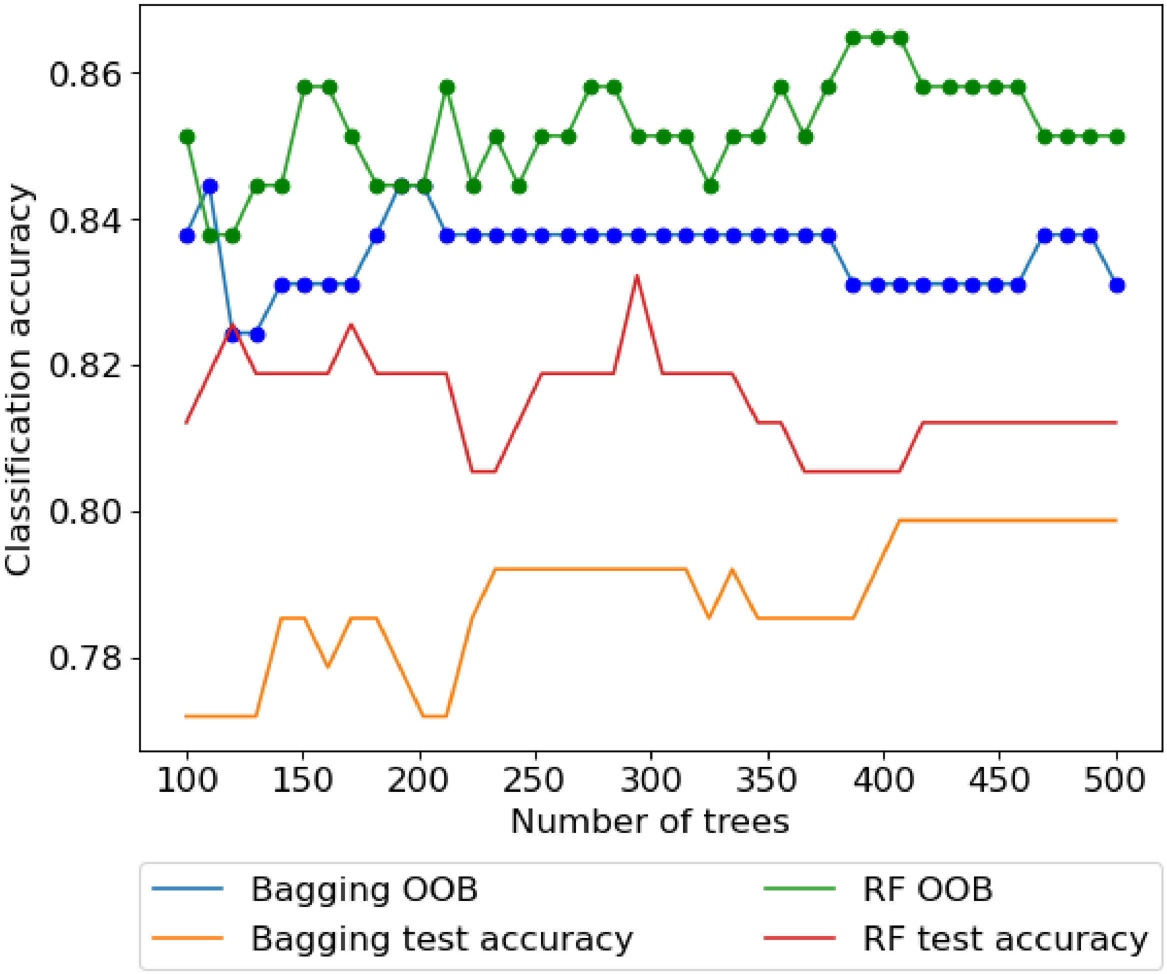
```
#Finding model accuracy vs number of trees
oob_accuracy={};test_accuracy={};
oob_accuracy2={};test_accuacy2={};

for i in np.linspace(100,500,40,dtype=int):
    #Bagging
    model = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=i, rando
                          n_jobs=-1,oob_score=True).fit(Xtrain, ytrain)
    oob_accuracy[i]=model.oob_score_  #Returns the out-of-bag classification accuracy of the
    test_accuracy[i]=model.score(Xtest,ytest) #Returns the classification accuracy of the mo

    #Random forest
    model2 = RandomForestClassifier(n_estimators=i, random_state=1,max_features=3,
                          n_jobs=-1,oob_score=True).fit(Xtrain, ytrain)
    oob_accuracy2[i]=model2.oob_score_  #Returns the out-of-bag classification accuracy of t
    test_accuacy2[i]=model2.score(Xtest,ytest) #Returns the classification accuracy of the m
```

```
#Feature importance for bagging
np.mean([tree.feature_importances_ for tree in model.estimators_],axis=0)
```

```
array([0.04381883, 0.05913479, 0.08585651, 0.07165678, 0.00302965,
       0.00903484, 0.05890448, 0.01223421, 0.072461  , 0.01337919,
       0.17495662, 0.18224651, 0.00527156, 0.00953965, 0.00396654,
       0.00163193, 0.09955286, 0.09332406])
```

Note that no predictor is too important to consider. That's why a small value of three for `max_features` is likely to decorrelate trees without compromising the quality of predictions.

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),label = 'Bagging OOB')
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),'o',color = 'blue')
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Bagging test accuracy')

plt.plot(oob_accuracy2.keys(),oob_accuracy2.values(),label = 'RF OOB')
plt.plot(oob_accuracy2.keys(),oob_accuracy2.values(),'o',color = 'green')
plt.plot(test_accuacy2.keys(),test_accuacy2.values(), label = 'RF test accuracy')

plt.xlabel('Number of trees')
plt.ylabel('Classification accuracy')
plt.legend(bbox_to_anchor=(0, -0.15, 1, 0), loc=2, ncol=2, mode="expand", borderaxespad=0)
```



In the above example we observe that random forest does improve over bagged trees in terms of classification accuracy. Unlike the previous two examples, the optimal value of `max_features` for random forests is much smaller than the total number of available predictors, thereby making the random forest model much different than the bagged tree model.

9 Adaptive Boosting

Read section 8.2.3 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

For the exact algorithms underlying the AdaBoost algorithm, check out the papers `AdaBoostRegressor()` and `AdaBoostClassifier()`.

9.1 Hyperparameters

There are 3 important parameters to tune in AdaBoost:

1. Number of trees
2. Depth of each tree
3. Learning rate

Let us visualize the accuracy of AdaBoost when we independently tweak each of the above parameters.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split, KFold, cross_val_predi
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold
from sklearn.ensemble import BaggingRegressor,BaggingClassifier,AdaBoostRegressor,AdaBoostCla
RandomForestRegressor
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
```

```
import itertools as it
import time as time

from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

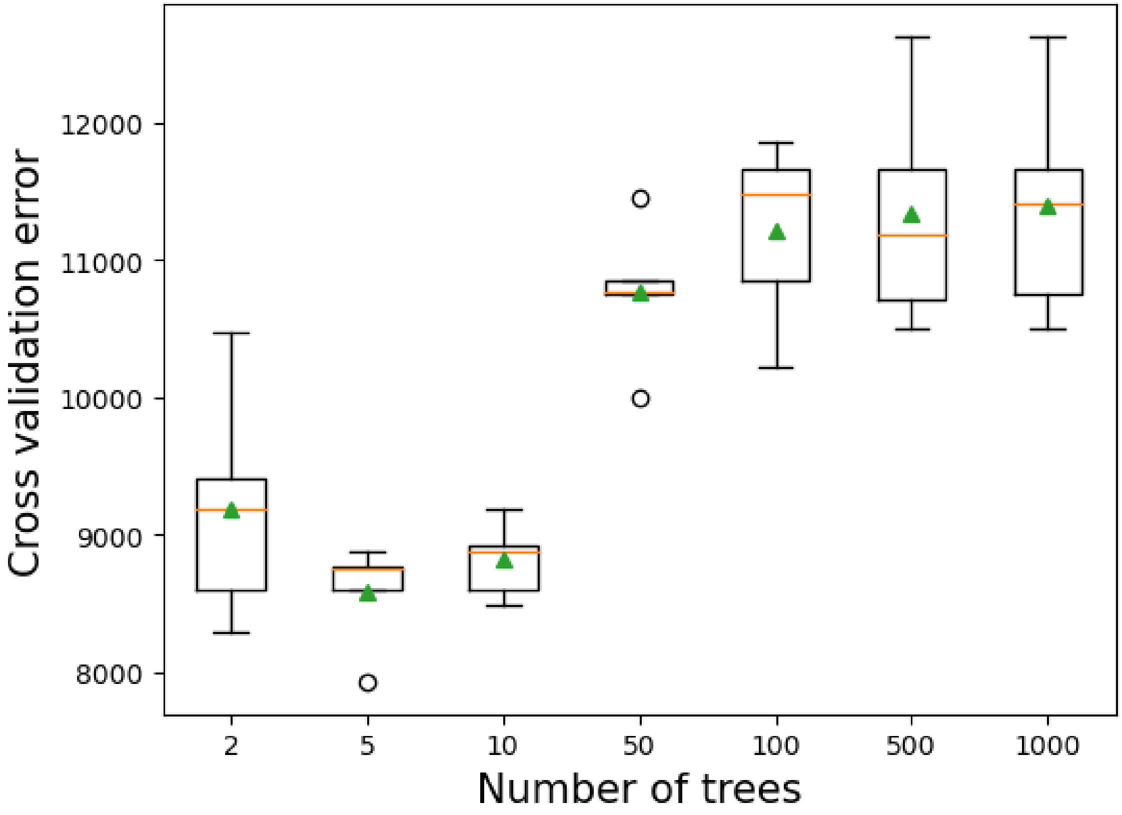
9.2 AdaBoost for regression

9.2.1 Number of trees vs cross validation error

As the number of trees increases, the prediction bias will decrease, and the prediction variance will increase. Thus, there will be an optimal number of trees that minimizes the prediction error.

```python
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [2, 5, 10, 50, 100, 500, 1000]
    for n in n_trees:
        models[str(n)] = AdaBoostRegressor(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_jo
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15);
```

```
>2 9190.253 (757.408)
>5 8583.629 (341.406)
>10 8814.328 (248.891)
>50 10763.138 (465.677)
>100 11217.783 (602.642)
>500 11336.088 (763.288)
>1000 11390.043 (752.446)
```



9.2.2 Depth of tree vs cross validation error

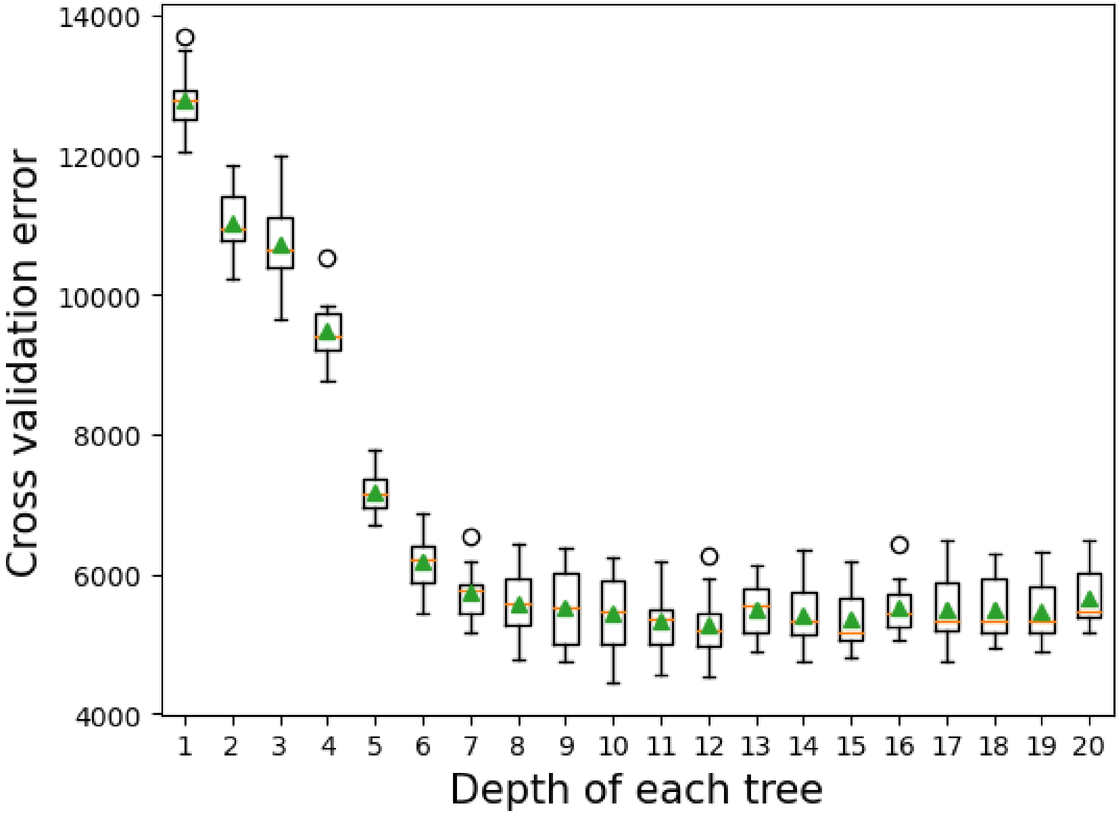
As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth for each weak learner that minimizes the prediction error.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define base model
        base = DecisionTreeRegressor(max_depth=i)
        # define ensemble model
        models[str(i)] = AdaBoostRegressor(base_estimator=base,n_estimators=50)
    return models
```

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_j
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15);
```

```
>1 12798.764 (490.538)
>2 11031.451 (465.520)
>3 10739.302 (636.517)
>4 9491.714 (466.764)
>5 7184.489 (324.484)
>6 6181.533 (411.394)
>7 5746.902 (407.451)
>8 5587.726 (473.619)
>9 5526.291 (541.512)
>10 5444.928 (554.170)
>11 5321.725 (455.899)
>12 5279.581 (492.785)
>13 5494.982 (393.469)
>14 5423.982 (488.564)
>15 5369.485 (441.799)
>16 5536.739 (409.166)
>17 5511.002 (517.384)
```

```
>18 5510.922 (478.285)
>19 5482.119 (465.565)
>20 5667.969 (468.964)
```



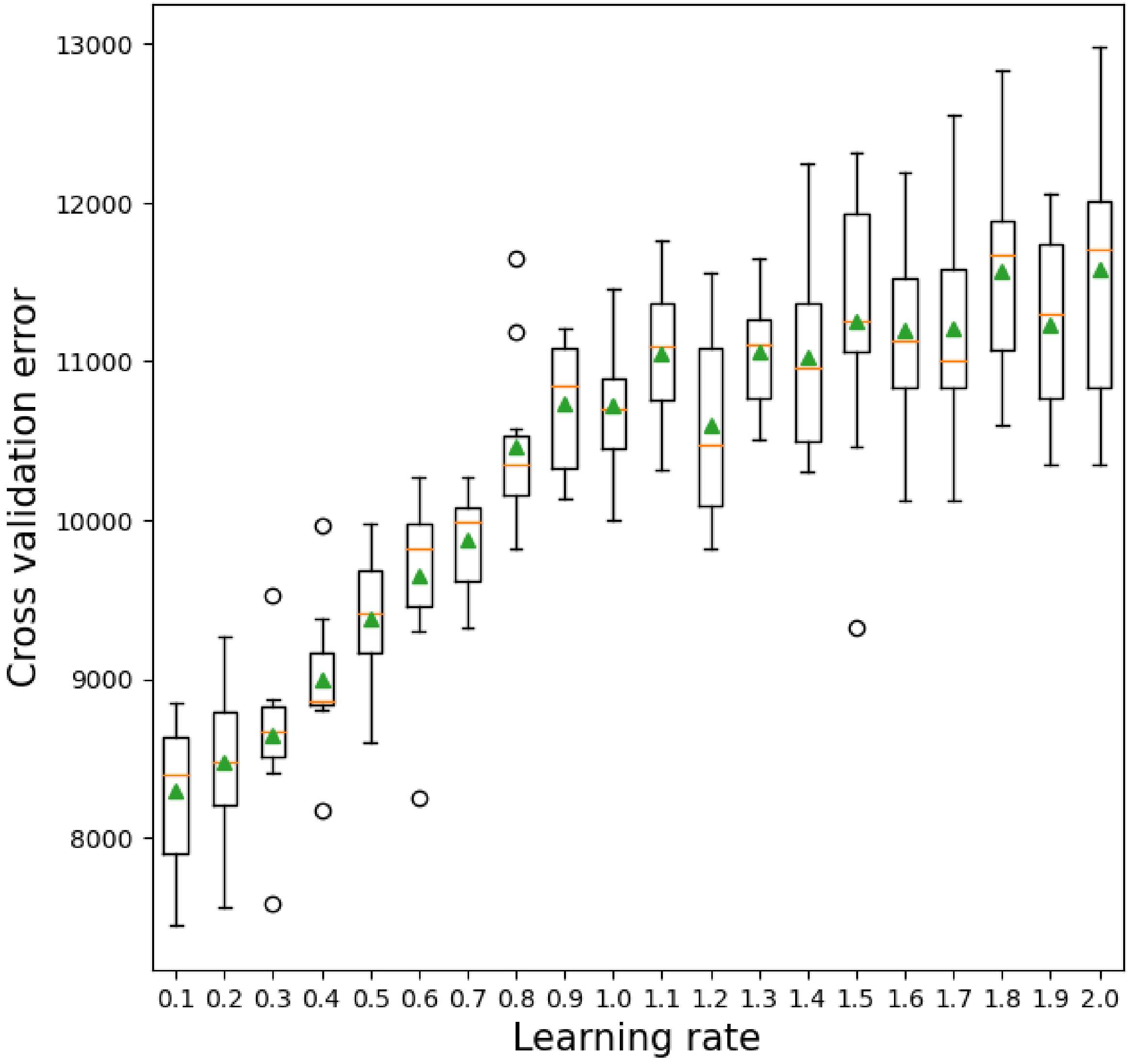
9.2.3 Learning rate vs cross validation error

The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to "learn" the response. If the learning rate is high, the response will be "learned" quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```python
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = AdaBoostRegressor(learning_rate=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_jo
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15);
```

```
>0.1 8291.9 (452.4)
>0.2 8475.7 (465.3)
>0.3 8648.5 (458.8)
>0.4 8995.5 (438.6)
>0.5 9376.1 (388.2)
>0.6 9655.3 (551.8)
>0.7 9877.3 (319.8)
>0.8 10466.8 (528.3)
>0.9 10728.9 (386.8)
>1.0 10720.2 (410.6)
>1.1 11043.9 (432.5)
>1.2 10602.5 (570.0)
>1.3 11058.8 (362.1)
```

```
>1.4 11022.7 (616.0)
>1.5 11252.5 (839.3)
>1.6 11195.3 (604.5)
>1.7 11206.3 (636.1)
>1.8 11569.1 (674.6)
>1.9 11232.3 (605.6)
>2.0 11581.0 (824.8)
```



9.2.4 Tuning AdaBoost for regression

As the optimal value of the parameters depend on each other, we need to optimize them simultaneously.

```
model = AdaBoostRegressor(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['estimator'] = [DecisionTreeRegressor(max_depth=3), DecisionTreeRegressor(max_depth=5),
                           DecisionTreeRegressor(max_depth=10),DecisionTreeRegressor(max_dept
# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (-grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

```
Best: 5346.490675 using {'estimator': DecisionTreeRegressor(max_depth=10), 'learning_rate': 
```

Note that for tuning `max_depth` of the base estimator - decision tree, we specified 4 different base estimators with different depths. However, there is a more concise way to do that. We can specify the `max_depth` of the estimator by adding a double underscore "`__`" between the `estimator` and the hyperparameter that we wish to tune *(`max_depth` here)*, and then specify its potential values in the `grid` itself as shown below. However, we'll then need to add `DecisionTreeRegressor()` as the estimator within the `AdaBoostRegressor()` function.

```
model = AdaBoostRegressor(random_state=1, estimator = DecisionTreeRegressor(random_state=1))
grid = dict()
grid['n_estimators'] = [10, 50, 100,200]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['estimator__max_depth'] = [3, 5, 10, 15]
# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
```

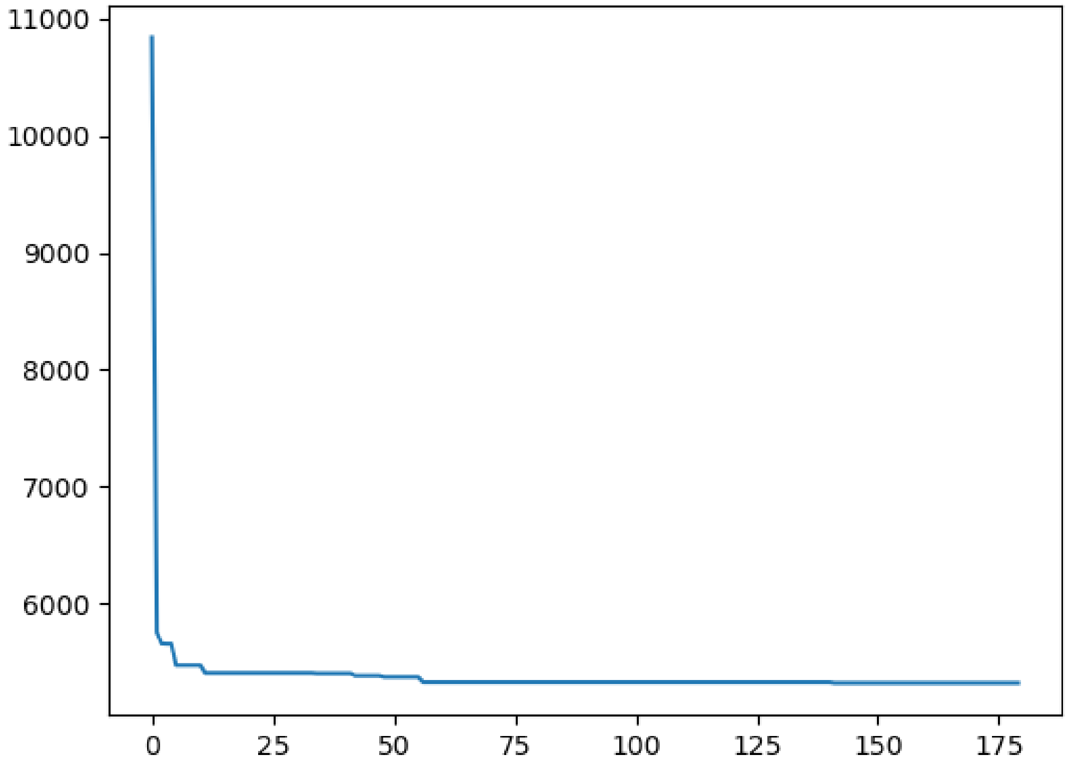
```
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (-grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

```
Best: 5346.490675 using {'estimator__max_depth': 10, 'learning_rate': 1.0, 'n_estimators': 5
```

The BayesSearchCV() approach also coverges to a slightly different set of optimal hyperparameter values. However, it gives a similar cross-validated RMSE. This is possible. There may be multiple hyperparameter values that are different from each other, but similar in performance. It may be a good idea to ensemble models based on these two distinct set of hyperparameter values that give an equally accurate model.

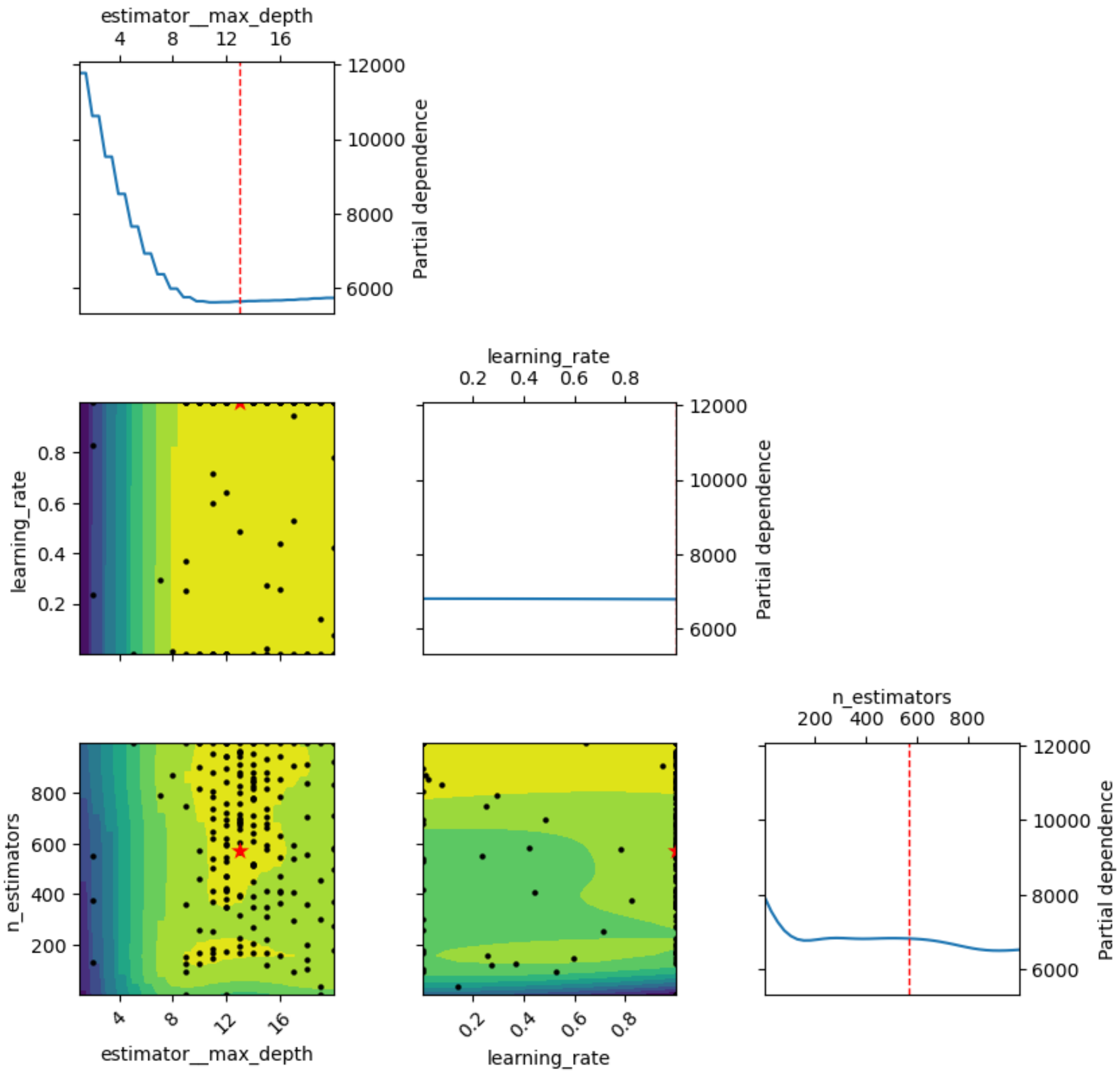
```
model = AdaBoostRegressor(estimator=DecisionTreeRegressor())
grid = dict()
grid['n_estimators'] = Integer(2, 1000)
grid['learning_rate'] = Real(0.0001, 1.0)
grid['estimator__max_depth'] = Integer(1, 20)

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
['estimator__max_depth', 'learning_rate', 'n_estimators'] = [13, 1.0, 570] 5325.017602505734
```

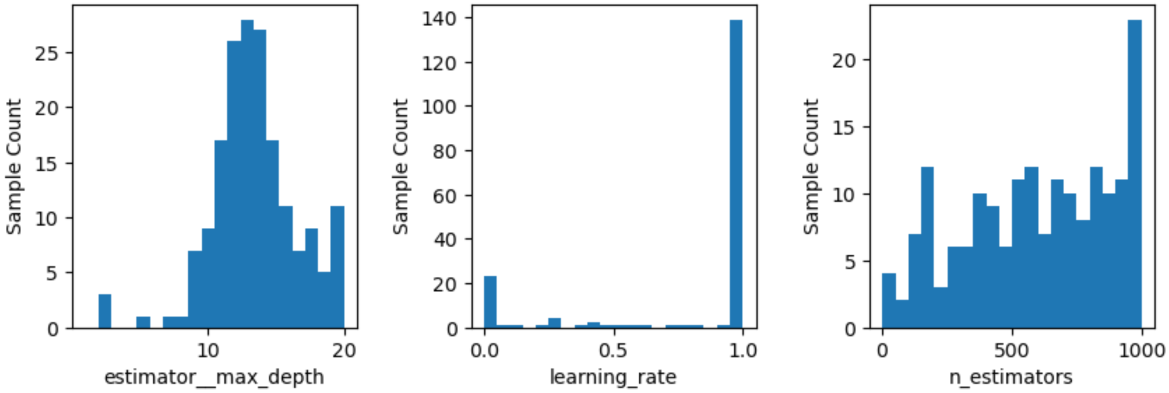


```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=AdaBoostRegressor(estimator=DecisionTreeRegressor()),
              n_iter=180, n_jobs=-1, random_state=10,
              scoring='neg_root_mean_squared_error',
              search_spaces={'estimator__max_depth': Integer(low=1, high=20, prior='uniform'
                             'learning_rate': Real(low=0.0001, high=1.0, prior='uniform', tra
                             'n_estimators': Integer(low=2, high=1000, prior='uniform', trans
```

```
plot_objective(gcv.optimizer_results_[0],
                    dimensions=['estimator__max_depth', 'learning_rate', 'n_estimators'], size
plt.show();
```



```
fig, ax = plt.subplots(1, 3, figsize = (10, 3))
plt.subplots_adjust(wspace=0.4)
plot_histogram(gcv.optimizer_results_[0], 0, ax = ax[0])
plot_histogram(gcv.optimizer_results_[0], 1, ax = ax[1])
plot_histogram(gcv.optimizer_results_[0], 2, ax = ax[2])
plt.show()
```



```
#Model based on the optimal hyperparameters
model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),n_estimators=50,lear
                          random_state=1).fit(X,y)
```

```
#RMSE of the optimized model on test data
pred1=model.predict(Xtest)
print("AdaBoost model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

```
AdaBoost model RMSE =  5693.165811600585
```

```
#Model based on the optimal hyperparameters
model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=13),n_estimators=570,lea
                          random_state=1).fit(X,y)
```

```
#RMSE of the optimized model on test data
pred2=model.predict(Xtest)
print("AdaBoost model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

```
AdaBoost model RMSE =  5434.852990644646
```

```
model = RandomForestRegressor(n_estimators=300, random_state=1,
                         n_jobs=-1, max_features=2).fit(X, y)
pred3 = model.predict(Xtest)
print("Random Forest model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest))
```

```
Random Forest model RMSE =  5642.45839697972
```

```
#Ensemble modeling
pred = 0.33*pred1+0.33*pred2 + 0.34*pred3
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

```
Ensemble model RMSE =  5402.832128650372
```

Combined, the random forest model and the Adaboost models do better than each of the individual models.

9.3 AdaBoost for classification

Below is the AdaBoost implementation on a classification problem. The takeaways are the same as that of the regression problem above.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

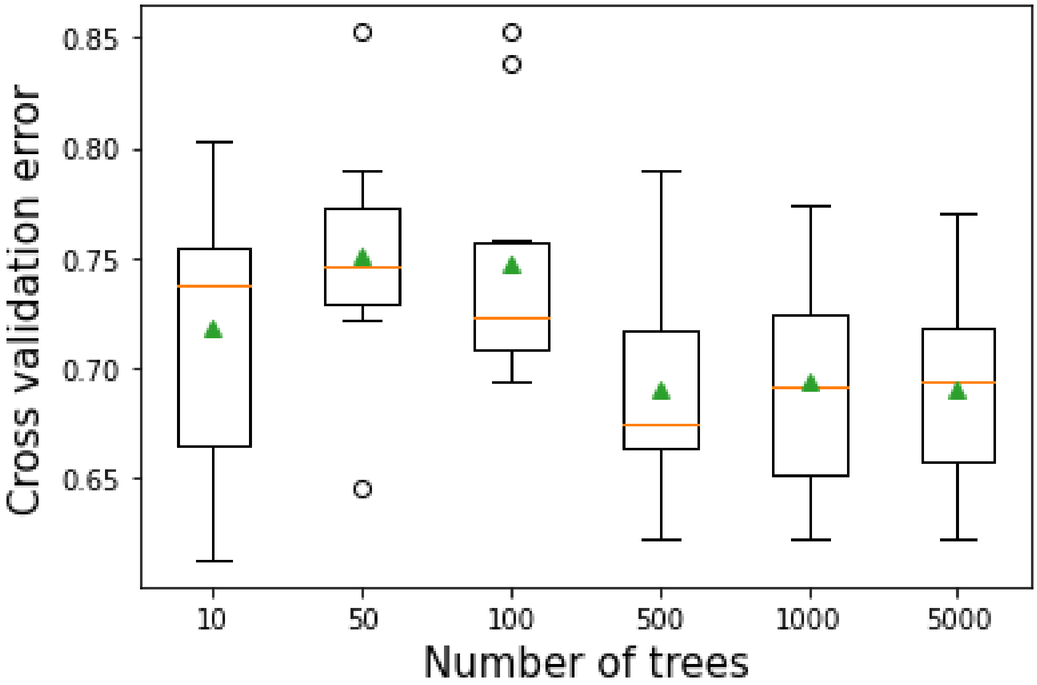
```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

9.3.1 Number of trees vs cross validation accuracy

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 1000, 5000]
    for n in n_trees:
        models[str(n)] = AdaBoostClassifier(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)
```

```
>10 0.718 (0.060)
>50 0.751 (0.051)
>100 0.748 (0.053)
>500 0.690 (0.045)
>1000 0.694 (0.048)
>5000 0.691 (0.044)
```

```
Text(0.5, 0, 'Number of trees')
```



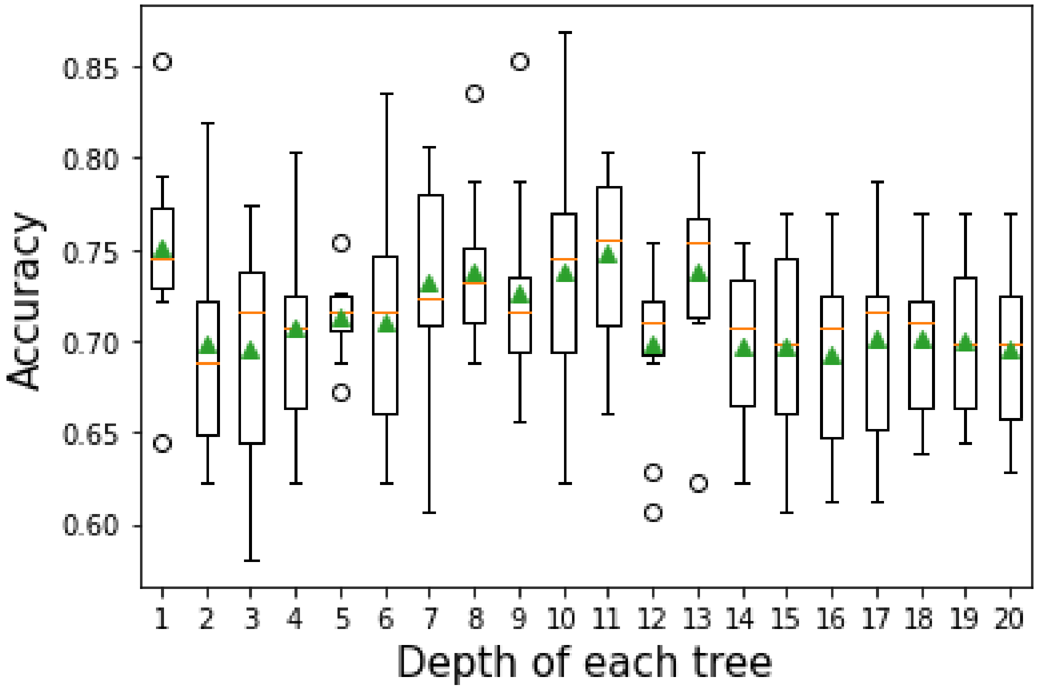
9.3.2 Depth of each tree vs cross validation accuracy

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define base model
        base = DecisionTreeClassifier(max_depth=i)
        # define ensemble model
        models[str(i)] = AdaBoostClassifier(estimator=base)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores
```

```
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)
```

```
>1 0.751 (0.051)
>2 0.699 (0.063)
>3 0.696 (0.062)
>4 0.707 (0.055)
>5 0.713 (0.021)
>6 0.710 (0.061)
>7 0.733 (0.057)
>8 0.738 (0.044)
>9 0.727 (0.053)
>10 0.738 (0.065)
>11 0.748 (0.048)
>12 0.699 (0.044)
>13 0.738 (0.047)
>14 0.697 (0.041)
>15 0.697 (0.052)
>16 0.692 (0.052)
>17 0.702 (0.056)
>18 0.702 (0.045)
>19 0.700 (0.040)
>20 0.696 (0.042)
```

```
Text(0.5, 0, 'Depth of each tree')
```



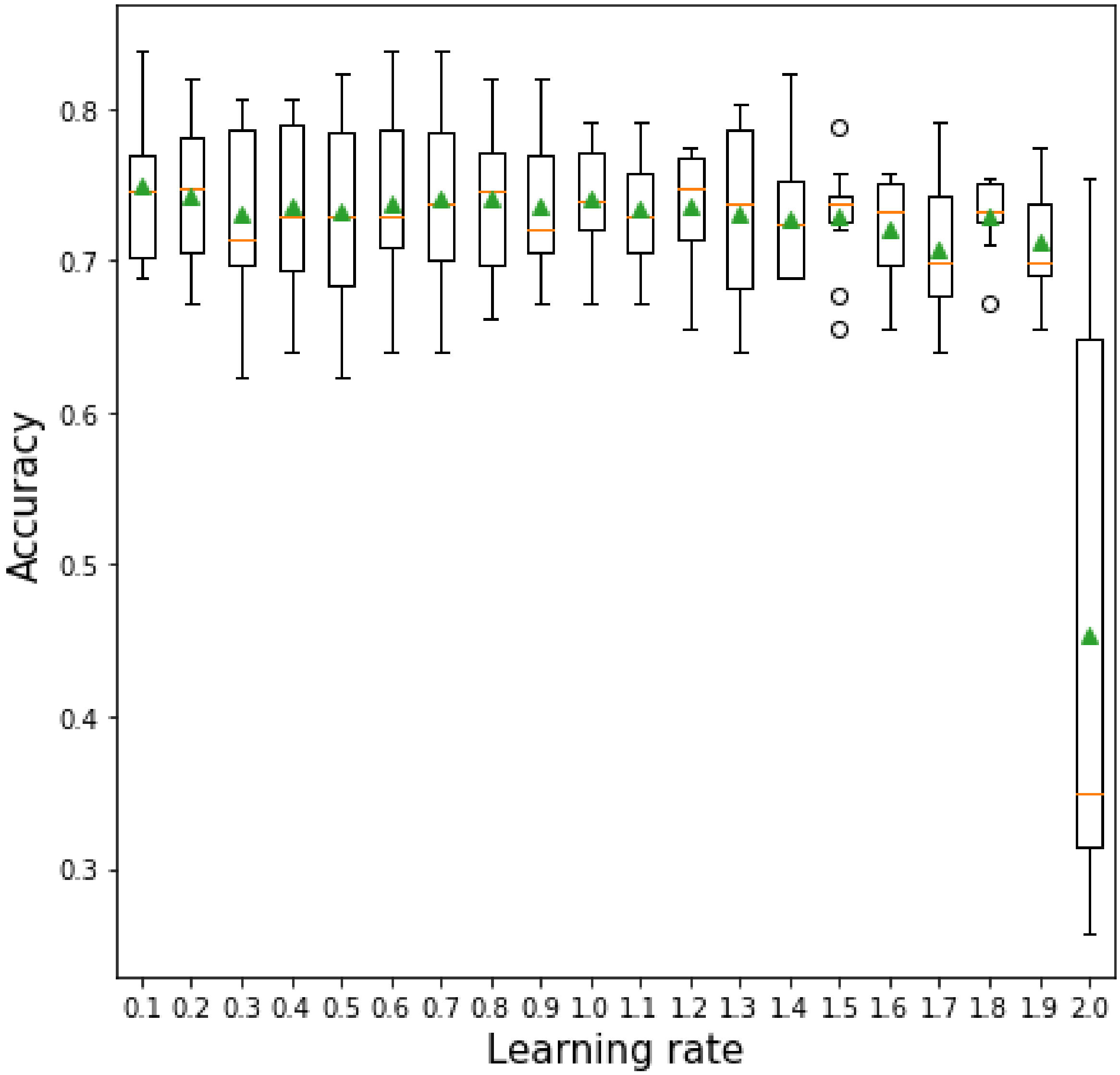
9.3.3 Learning rate vs cross validation accuracy

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = AdaBoostClassifier(learning_rate=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
```

```
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)
```

```
>0.1 0.749 (0.052)
>0.2 0.743 (0.050)
>0.3 0.731 (0.057)
>0.4 0.736 (0.053)
>0.5 0.733 (0.062)
>0.6 0.738 (0.058)
>0.7 0.741 (0.056)
>0.8 0.741 (0.049)
>0.9 0.736 (0.048)
>1.0 0.741 (0.035)
>1.1 0.734 (0.037)
>1.2 0.736 (0.038)
>1.3 0.731 (0.057)
>1.4 0.728 (0.041)
>1.5 0.730 (0.036)
>1.6 0.720 (0.038)
>1.7 0.707 (0.045)
>1.8 0.730 (0.024)
>1.9 0.712 (0.033)
>2.0 0.454 (0.191)
```

```
Text(0.5, 0, 'Learning rate')
```



9.3.4 Tuning AdaBoost Classifier hyperparameters

```
model = AdaBoostClassifier(random_state=1, estimator = DecisionTreeClassifier())
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['estimator__max_depth'] = [1, 2, 3, 4]
# define the evaluation procedure
```

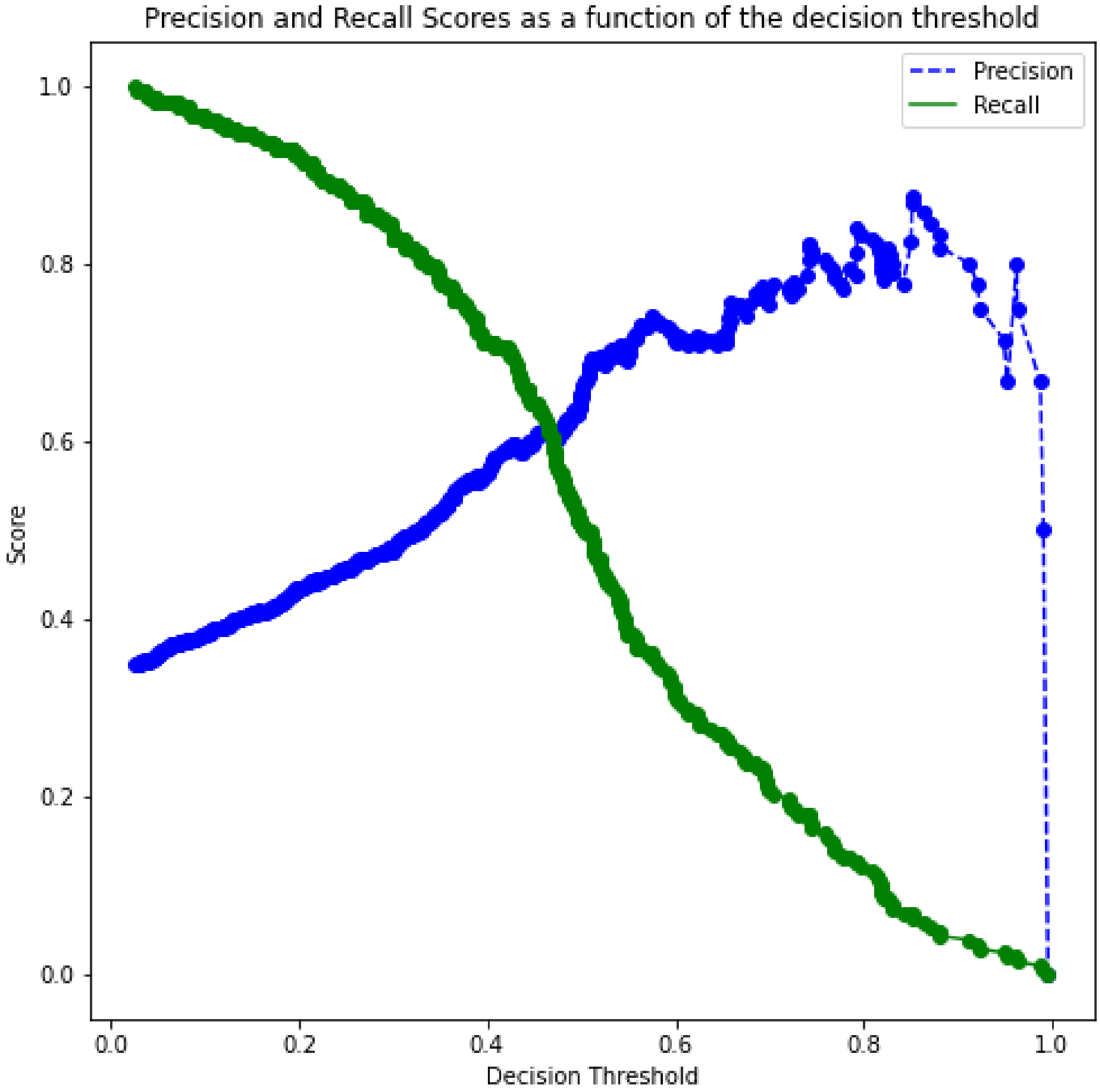
```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                          verbose = True)
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
#for mean, stdev, param in zip(means, stds, params):
#    print("%f (%f) with: %r" % (mean, stdev, param)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
Best: 0.763934 using {'estimator__max_depth': 3, 'learning_rate': 0.01, 'n_estimators': 200}
```

9.3.5 Tuning the decision threshold probability

We'll find a decision threshold probability that balances recall with precision.

```
#Model based on the optimal parameters
model = AdaBoostClassifier(random_state=1, estimator = DecisionTreeClassifier(max_depth=3),le
                          n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly usi
# predicted probabilities on train data, as the model may be overfitting on the train data, a
# may lead to misleading results
cross_val_ypred = cross_val_predict(AdaBoostClassifier(random_state=1,base_estimator = Decis
                          n_estimators=200), X, y, cv = 5, method = 'predict_proba')

p, r, thresholds = precision_recall_curve(y, cross_val_ypred[:,1])
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
    plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```



```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].resha
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
```

```
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and incre
# the last value will provide the maximum threshold probability for the recall to be more tha
# We wish to find the maximum threshold probability to obtain the maximum possible precision
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

```
array([0.33488762, 0.50920245, 0.80193237])
```

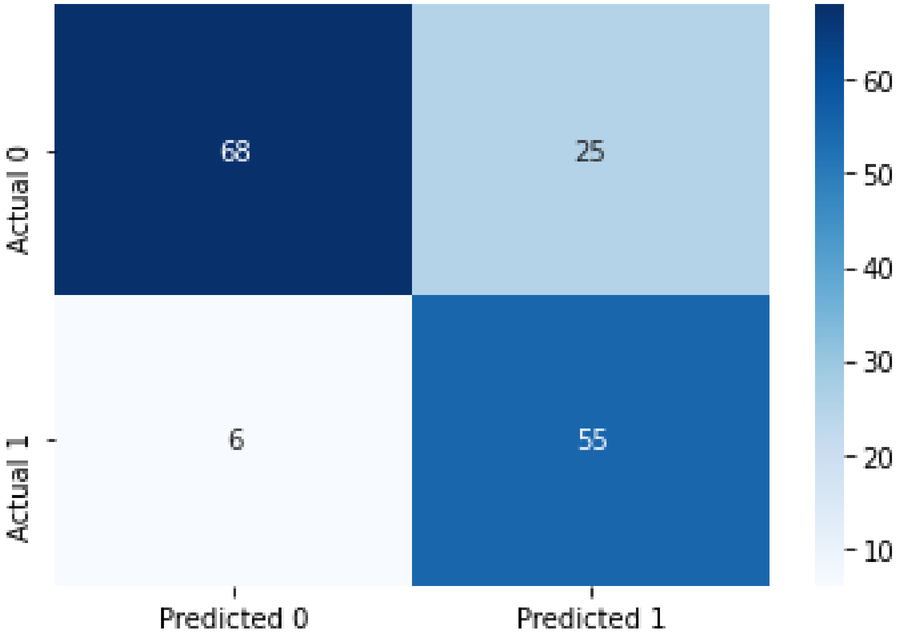
```
#Optimal decision threshold probability
thres = recall_more_than_80[recall_more_than_80.shape[0]-1][0]
thres
```

```
0.3348876199649718
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = thres

y_pred_prob = model.predict_proba(Xtest)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is grea
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  79.87012987012987
```

```
ROC-AUC:  0.8884188260179798
Precision:  0.6875
Recall:  0.9016393442622951
```



The above model is similar to the one obtained with bagging / random forest. However, adaptive boosting may lead to better classification performance as compared to bagging / random forest.

10 Gradient Boosting

Check the gradient boosting algorithm in section 10.10.2 of the book, Elements of Statistical Learning before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

10.1 Hyperparameters

There are 5 important parameters to tune in Gradient boosting:

1. Number of trees
2. Depth of each tree
3. Learning rate
4. Subsample fraction
5. Maximum features

Let us visualize the accuracy of Gradient boosting when we independently tweak each of the above parameters.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split, KFold, cross_val_predi
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold
from sklearn.ensemble import GradientBoostingRegressor,GradientBoostingClassifier, BaggingReg
from sklearn.linear_model import LinearRegression,LogisticRegression
```

```python
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time

from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

```python
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```python
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

10.2 Gradient boosting for regression

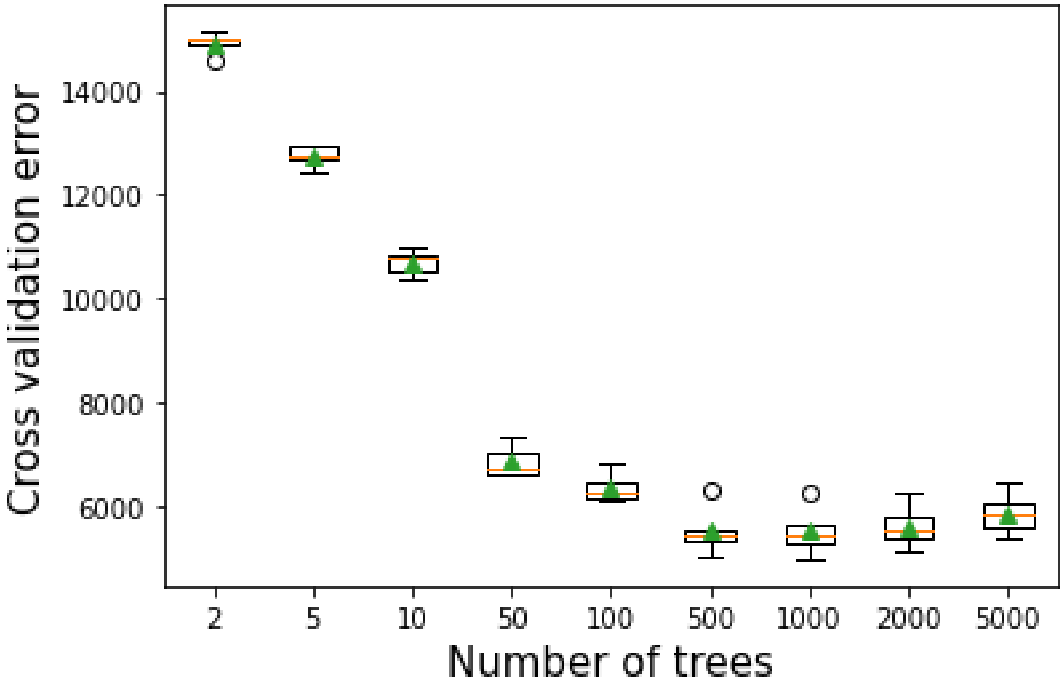
10.2.1 Number of trees vs cross validation error

As per the documentation, Gradient boosting is fairly robust *(as compared to AdaBoost)* to over-fitting (why?) so a large number usually results in better performance. Note that the number of trees still need to be tuned for optimal performance.

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [2, 5, 10, 50, 100, 500, 1000, 2000, 5000]
    for n in n_trees:
        models[str(n)] = GradientBoostingRegressor(n_estimators=n,random_state=1,loss='huber
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv,
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)
```

```
>2 14927.566 (179.475)
>5 12743.148 (189.408)
>10 10704.199 (226.234)
>50 6869.066 (278.885)
>100 6354.656 (270.097)
>500 5515.622 (424.516)
>1000 5515.251 (427.767)
>2000 5600.041 (389.687)
>5000 5854.168 (362.223)
```

```
Text(0.5, 0, 'Number of trees')
```



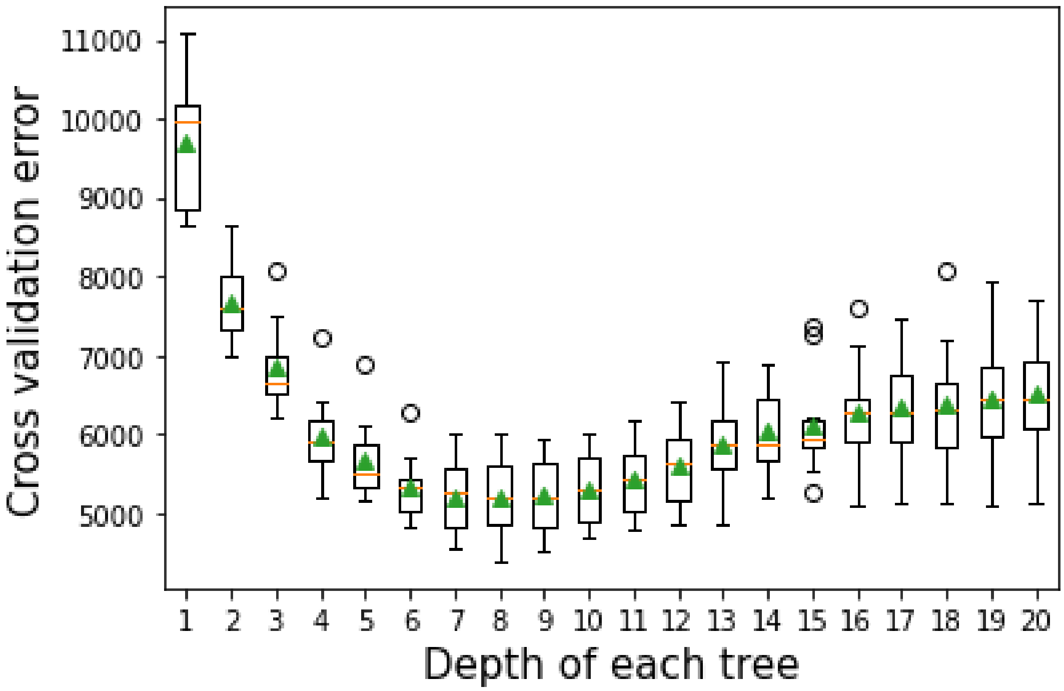
10.2.2 Depth of tree vs cross validation error

As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth of each weak learner that minimizes the prediction error.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = GradientBoostingRegressor(n_estimators=50,random_state=1,max_depth=
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)
```

```
>1 9693.731 (810.090)
>2 7682.569 (489.841)
>3 6844.225 (536.792)
>4 5972.203 (538.693)
>5 5664.563 (497.882)
>6 5329.130 (404.330)
>7 5210.934 (461.038)
>8 5197.204 (494.957)
>9 5227.975 (478.789)
>10 5299.782 (446.509)
>11 5433.822 (451.673)
>12 5617.946 (509.797)
>13 5876.424 (542.981)
>14 6030.507 (560.447)
>15 6125.914 (643.852)
>16 6294.784 (672.646)
>17 6342.327 (677.050)
>18 6372.418 (791.068)
>19 6456.471 (741.693)
>20 6503.622 (759.193)
```

```
Text(0.5, 0, 'Depth of each tree')
```



10.2.3 Learning rate vs cross validation error

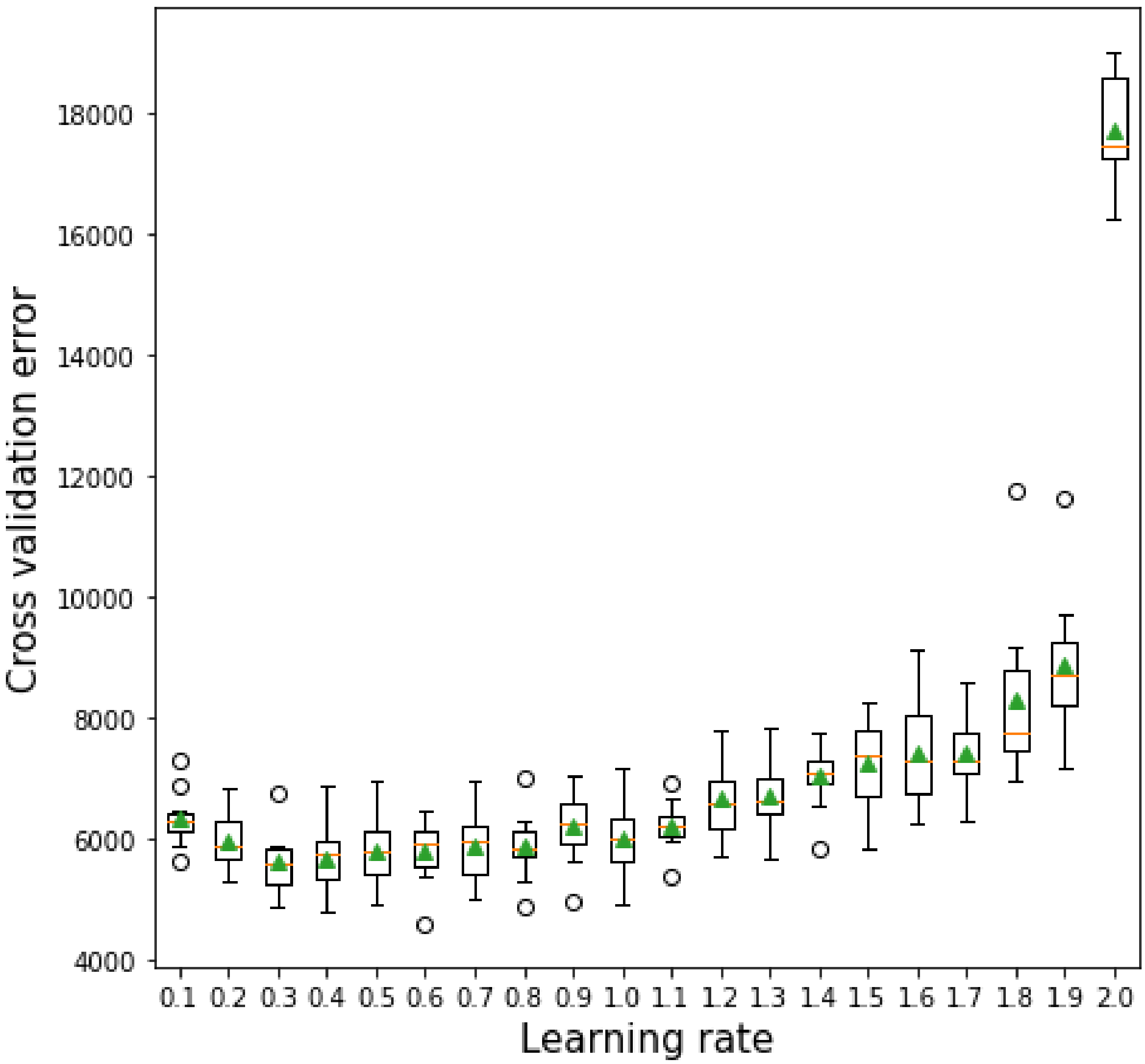
The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to "learn" the response. If the learning rate is high, the response will be "learned" quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = GradientBoostingRegressor(learning_rate=i,random_state=1,loss='huber')
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, r
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)
```

```
>0.1 6329.8 (450.7)
>0.2 5942.9 (454.8)
>0.3 5618.4 (490.8)
>0.4 5665.9 (577.3)
>0.5 5783.5 (561.7)
>0.6 5773.8 (500.3)
>0.7 5875.5 (565.7)
>0.8 5878.5 (540.5)
>0.9 6214.4 (594.3)
>1.0 5986.1 (601.5)
>1.1 6216.5 (395.3)
>1.2 6667.5 (657.2)
>1.3 6717.4 (594.4)
>1.4 7048.4 (531.7)
>1.5 7265.0 (742.0)
>1.6 7404.4 (868.2)
>1.7 7425.8 (606.3)
>1.8 8283.0 (1345.3)
```

```
>1.9 8872.2 (1137.9)
>2.0 17713.3 (865.3)
```

```
Text(0.5, 0, 'Learning rate')
```

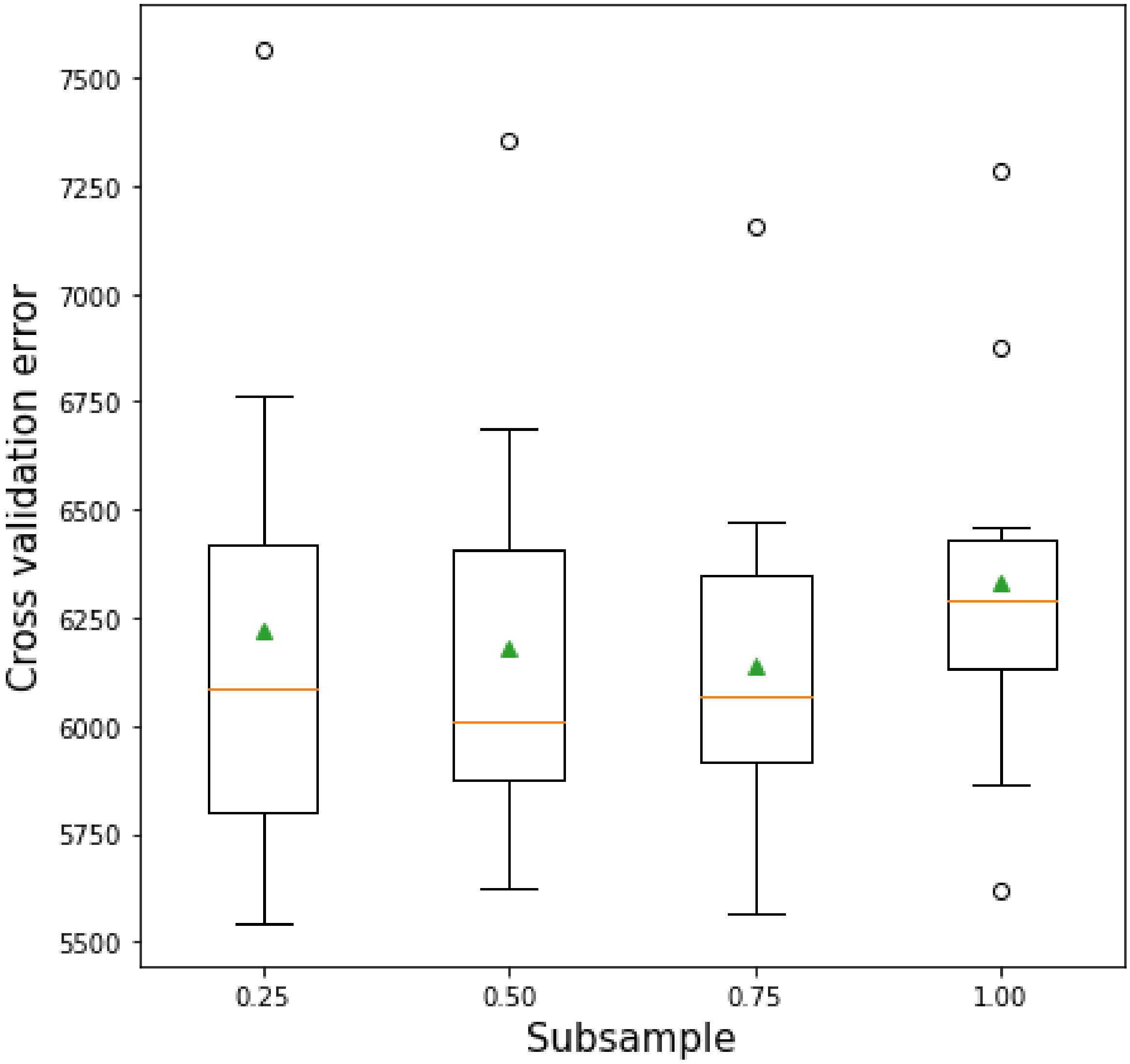


10.2.4 Subsampling vs cross validation error

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for s in np.arange(0.25, 1.1, 0.25):
        key = '%.2f' % s
        models[key] = GradientBoostingRegressor(random_state=1,subsample=s,loss='huber')
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv,
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Subsample',fontsize=15)
```

```
>0.25 6219.59 (569.97)
>0.50 6178.28 (501.87)
>0.75 6141.96 (432.66)
>1.00 6329.79 (450.72)
```

```
Text(0.5, 0, 'Subsample')
```



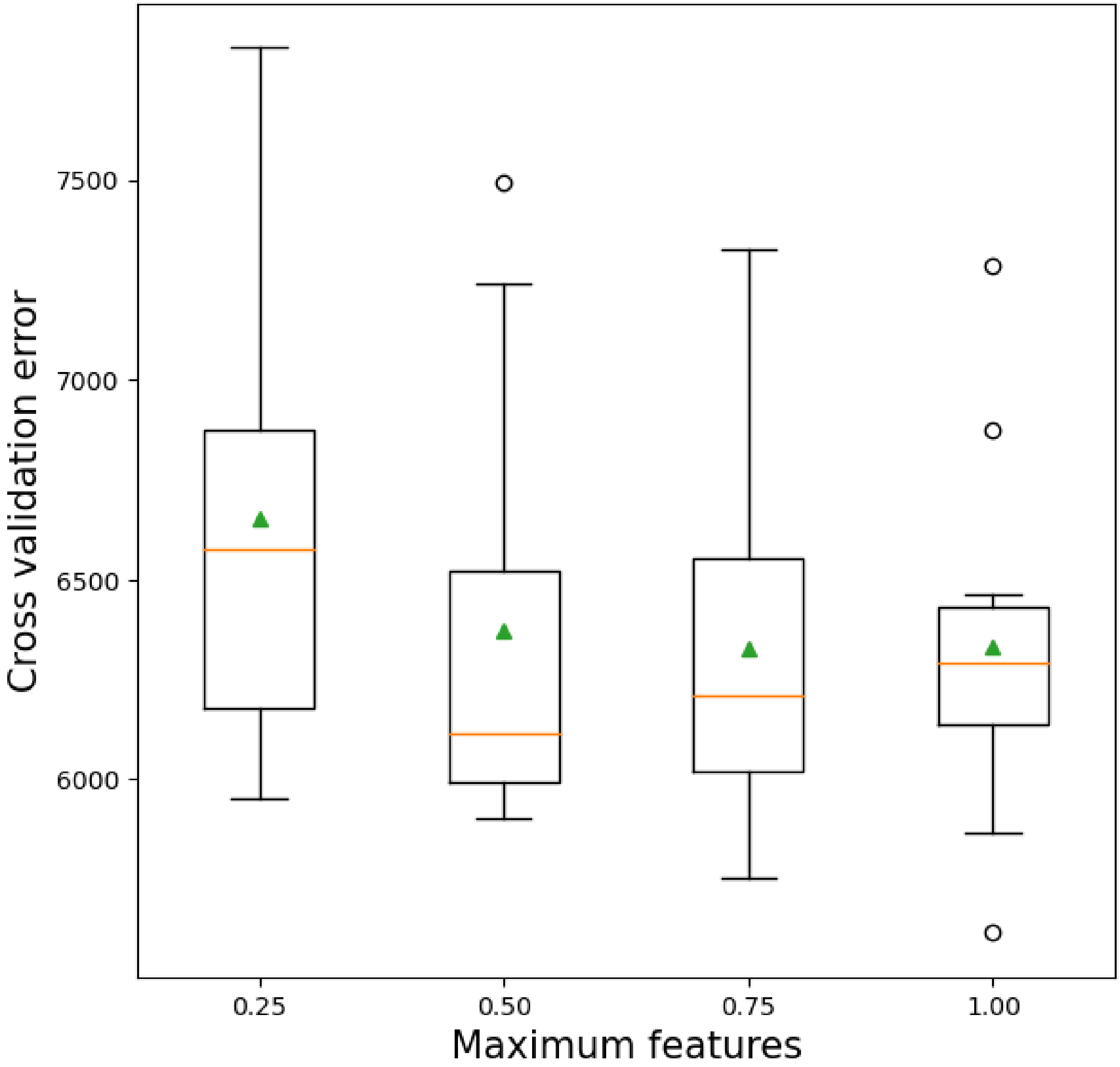
10.2.5 Maximum features vs cross-validation error

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for s in np.arange(0.25, 1.1, 0.25):
        key = '%.2f' % s
        models[key] = GradientBoostingRegressor(random_state=1,max_features=s,loss='huber')
    return models
```

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Maximum features',fontsize=15)
```

```
>0.25 6654.27 (567.72)
>0.50 6373.92 (538.53)
>0.75 6325.55 (470.41)
>1.00 6329.79 (450.72)
```

```
Text(0.5, 0, 'Maximum features')
```



10.2.6 Tuning Gradient boosting for regression

As the optimal value of the parameters depend on each other, we need to optimize them simultaneously.

```
start_time = time.time()
model = GradientBoostingRegressor(random_state=1,loss='huber')
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
```

```
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['max_depth'] = [3,5,8,10,12,15]

# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_r
                          verbose = True)
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (np.sqrt(-grid_result.best_score_), grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
#for mean, stdev, param in zip(means, stds, params):
#    print("%f (%f) with: %r" % (mean, stdev, param)
print("Time taken = ",(time.time()-start_time)/60," minutes")
```

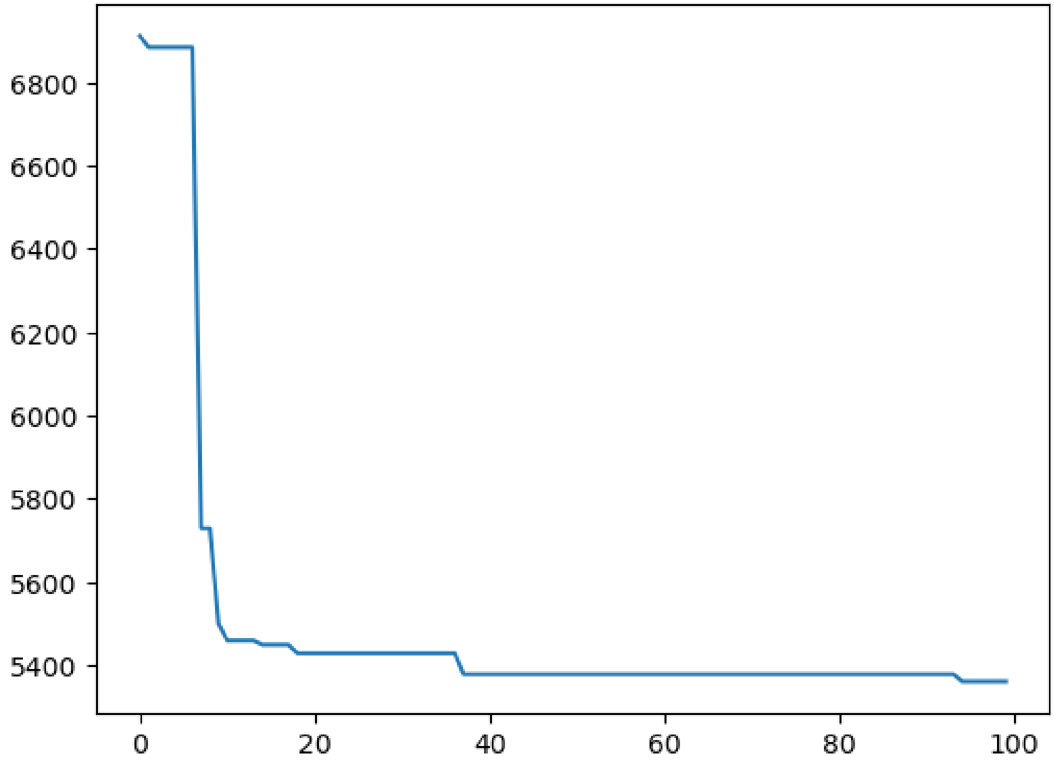
```
Best: 5190.765919 using {'learning_rate': 0.1, 'max_depth': 8, 'n_estimators': 100}
Time taken =  46.925597019990285  minutes
```

Note that the code takes 46 minutes to run. In case of a lot of hyperparameters, RandomizedSearchCV may be preferred to trade-off between optimality of the solution and computational cost.

```
model = GradientBoostingRegressor(random_state=1, loss='huber')
grid = dict()
grid['n_estimators'] = Integer(2, 1000)
grid['learning_rate'] = Real(0.0001, 1.0)
grid['max_leaf_nodes'] = Integer(4, 5000)
grid['subsample'] = Real(0.1, 1)
grid['max_features'] = Real(0.1, 1)

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 100, random_state = 1,
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()
start_time = time.time()
```

```
def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    print("Time so far = ", np.round((time.time()-start_time)/60), "minutes")
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
['learning_rate', 'max_features', 'max_leaf_nodes', 'n_estimators', 'subsample'] = [0.2310208
Time so far =  21.0 minutes
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=GradientBoostingRegressor(loss='huber', random_state=1),
              n_iter=100, n_jobs=-1, random_state=1,
              scoring='neg_root_mean_squared_error',
              search_spaces={'learning_rate': Real(low=0.0001, high=1.0, prior='uniform', tra
                             'max_features': Real(low=0.1, high=1, prior='uniform', transform
                             'max_leaf_nodes': Integer(low=4, high=5000, prior='uniform', tra
                             'n_estimators': Integer(low=2, high=1000, prior='uniform', trans
                             'subsample': Real(low=0.1, high=1, prior='uniform', transform='r
```

```
#Model based on the optimal parameters
model = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                         random_state=1,loss='huber').fit(X,y)
```

```
#RMSE of the optimized model on test data
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

```
Gradient boost RMSE =  5405.787029062213
```

```
#Model based on the optimal parameters
model_bayes = GradientBoostingRegressor(max_leaf_nodes=5000,n_estimators=817,learning_rate=0
                         random_state=1,subsample=1.0,loss='huber').fit(X,y)
```

```
#RMSE of the optimized model on test data
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model_bayes.predict(Xtest),ytest))
```

```
Gradient boost RMSE =  5734.200307094321
```

```
#Let us combine the Gradient boost model with other models
model2 = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimators=50
                         random_state=1).fit(X,y)
print("AdaBoost RMSE = ",np.sqrt(mean_squared_error(model2.predict(Xtest),ytest)))
model3 = RandomForestRegressor(n_estimators=300, random_state=1,
                        n_jobs=-1, max_features=2).fit(X, y)
print("Random Forest RMSE = ",np.sqrt(mean_squared_error(model3.predict(Xtest),ytest)))
```

```
AdaBoost RMSE =  5693.165811600585
Random Forest RMSE =  5642.45839697972
```

```
#Ensemble model
pred1=model.predict(Xtest)#Gradient boost
pred2=model2.predict(Xtest)#Adaboost
pred3=model3.predict(Xtest)#Random forest
pred = 0.34*pred1+0.33*pred2+0.33*pred3 #Higher weight to the better model
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

```
Ensemble model RMSE =  5364.478227748279
```

10.2.7 Ensemble modeling (for regression models)

```
#Ensemble model
pred1=model.predict(Xtest)#Gradient boost
pred2=model2.predict(Xtest)#Adaboost
pred3=model3.predict(Xtest)#Random forest
pred = 0.6*pred1+0.2*pred2+0.2*pred3 #Higher weight to the better model
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

```
Ensemble model RMSE =  5323.119083375402
```

Combined, the random forest model, gradient boost and the Adaboost model do better than each of the individual models.

Note that ideally we should do K-fold cross validation to figure out the optimal weights. We'll learn about ensembling techniques later in the course.

10.3 Gradient boosting for classification

Below is the Gradient boost implementation on a classification problem. The takeaways are the same as that of the regression problem above.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

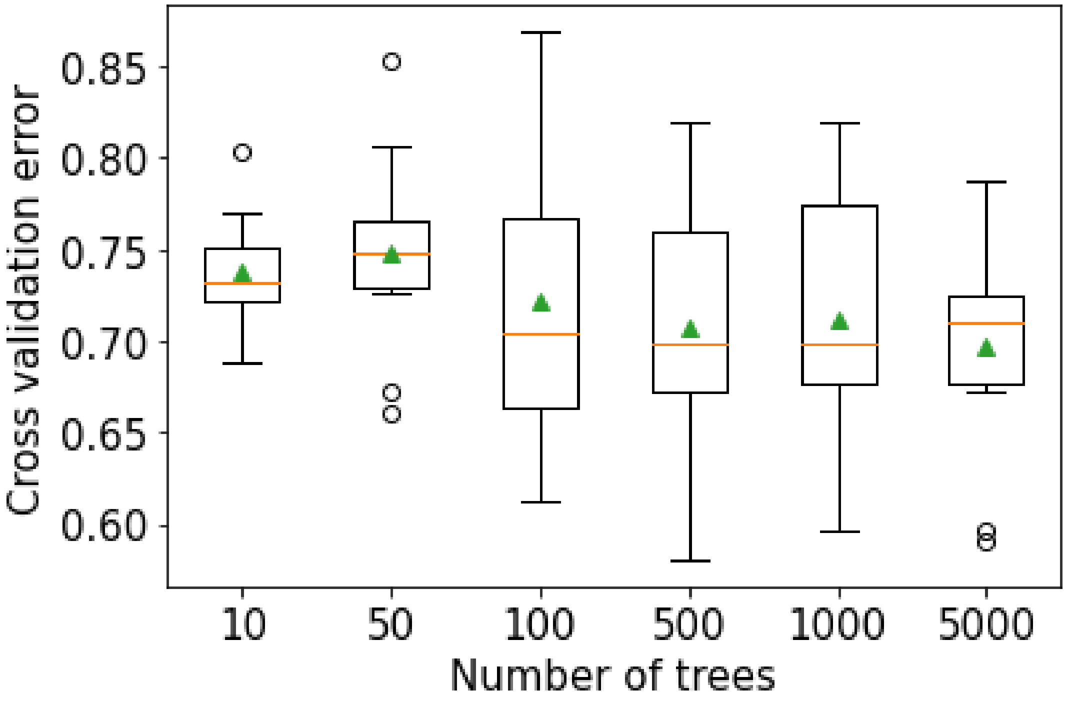
```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

10.3.1 Number of trees vs cross validation accuracy

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 1000, 5000]
    for n in n_trees:
        models[str(n)] = GradientBoostingClassifier(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)
```

```
>10 0.738 (0.031)
>50 0.748 (0.054)
>100 0.722 (0.075)
>500 0.707 (0.066)
>1000 0.712 (0.075)
>5000 0.697 (0.061)
```

```
Text(0.5, 0, 'Number of trees')
```



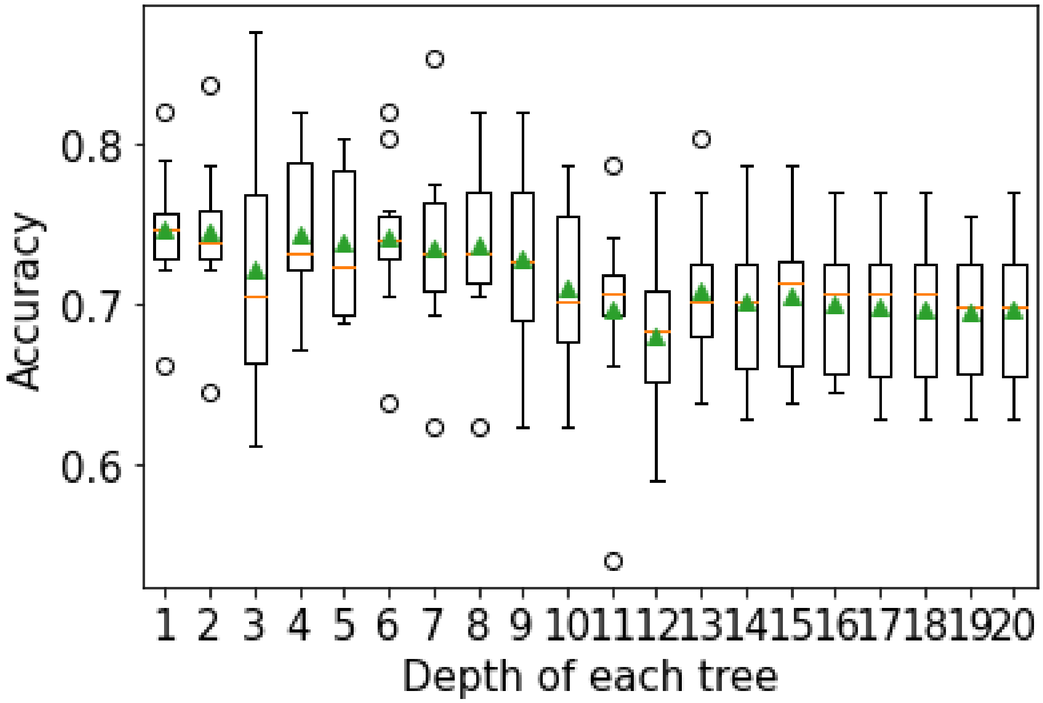
10.3.2 Depth of each tree vs cross validation accuracy

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = GradientBoostingClassifier(random_state=1,max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
```

```
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)
```

```
>1 0.746 (0.040)
>2 0.744 (0.046)
>3 0.722 (0.075)
>4 0.743 (0.049)
>5 0.738 (0.046)
>6 0.741 (0.047)
>7 0.735 (0.057)
>8 0.736 (0.051)
>9 0.728 (0.055)
>10 0.710 (0.050)
>11 0.697 (0.061)
>12 0.681 (0.056)
>13 0.709 (0.047)
>14 0.702 (0.048)
>15 0.705 (0.048)
>16 0.700 (0.042)
>17 0.699 (0.048)
>18 0.697 (0.050)
>19 0.696 (0.042)
>20 0.697 (0.048)
```

```
Text(0.5, 0, 'Depth of each tree')
```



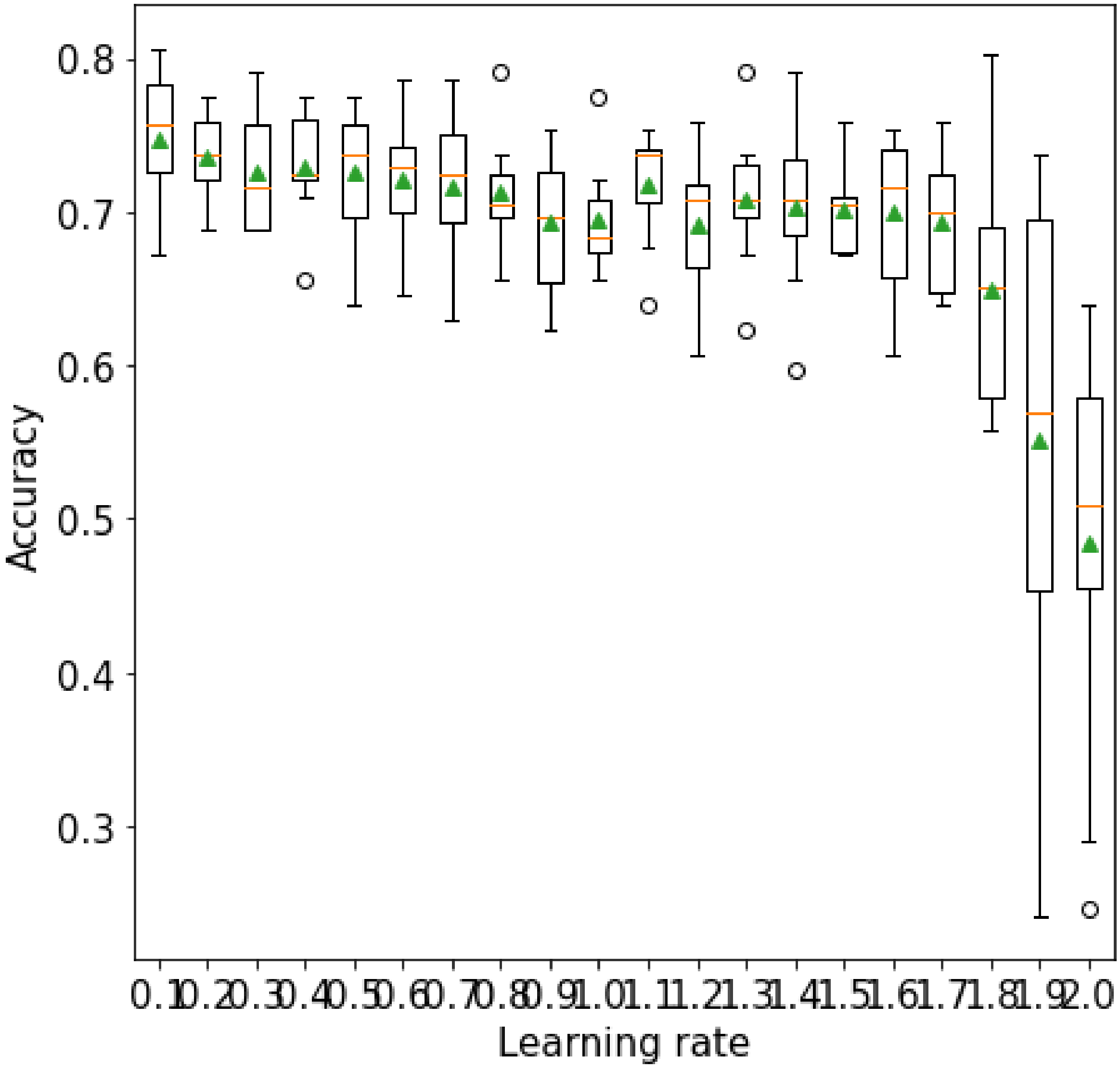
10.3.3 Learning rate vs cross validation accuracy

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = GradientBoostingClassifier(learning_rate=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
```

```python
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)
```

```
>0.1 0.747 (0.044)
>0.2 0.736 (0.028)
>0.3 0.726 (0.039)
>0.4 0.730 (0.034)
>0.5 0.726 (0.041)
>0.6 0.722 (0.043)
>0.7 0.717 (0.050)
>0.8 0.713 (0.033)
>0.9 0.694 (0.045)
>1.0 0.695 (0.032)
>1.1 0.718 (0.034)
>1.2 0.692 (0.045)
>1.3 0.708 (0.042)
>1.4 0.704 (0.050)
>1.5 0.702 (0.028)
>1.6 0.700 (0.050)
>1.7 0.694 (0.044)
>1.8 0.650 (0.075)
>1.9 0.551 (0.163)
>2.0 0.484 (0.123)
```

```
Text(0.5, 0, 'Learning rate')
```



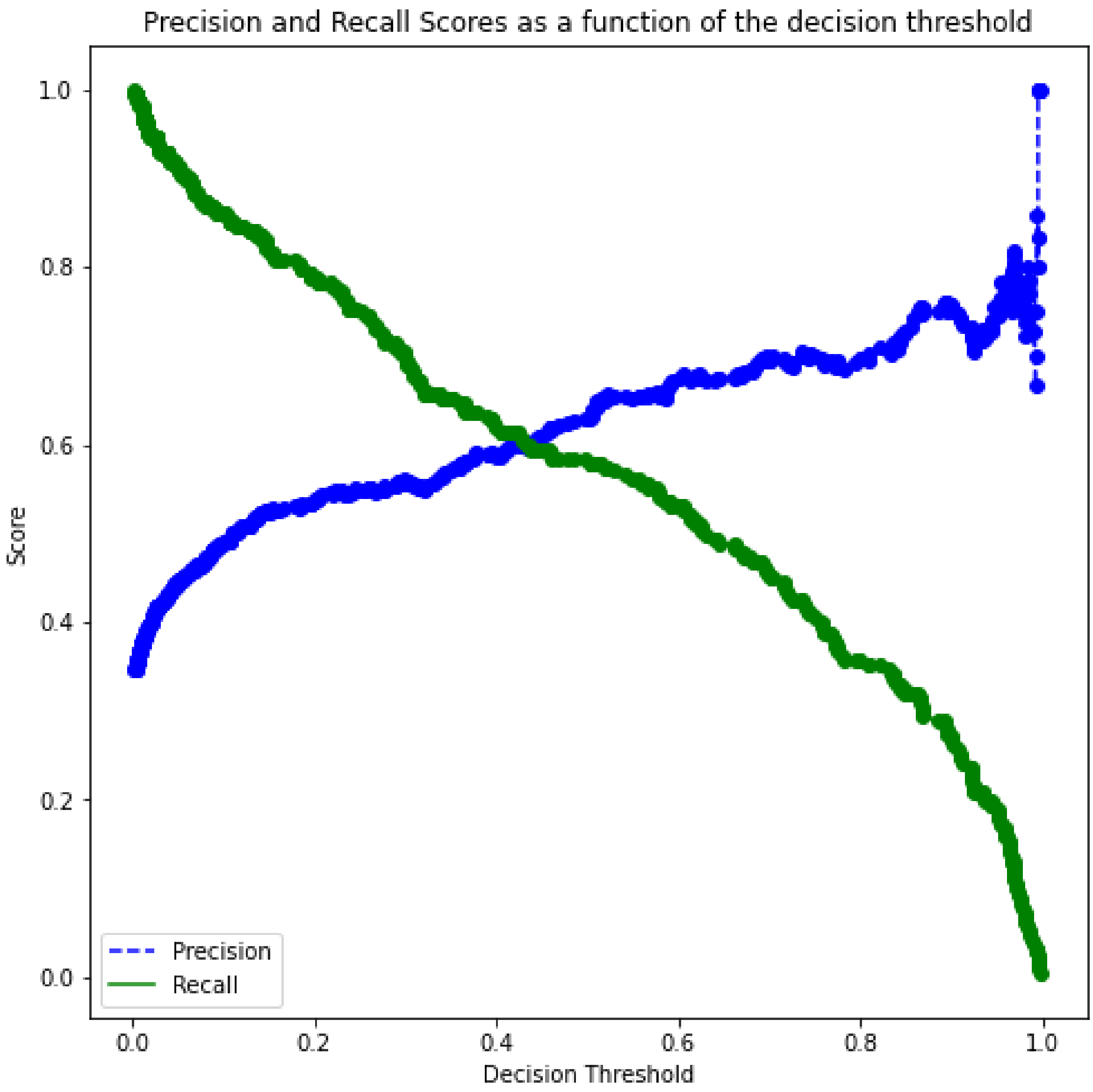
10.3.4 Tuning Gradient boosting Classifier

```
start_time = time.time()
model = GradientBoostingClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['max_depth'] = [1,2,3,4,5]
```

```
grid['subsample'] = [0.5,1.0]
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, verbose = Tru
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
print("Time taken = ", time.time() - start_time, "seconds")
```

```
Fitting 5 folds for each of 250 candidates, totalling 1250 fits
Best: 0.701045 using {'learning_rate': 1.0, 'max_depth': 3, 'n_estimators': 200, 'subsample'
Time taken =  32.46394085884094
```

```
#Model based on the optimal parameters
model = GradientBoostingClassifier(random_state=1,max_depth=3,learning_rate=0.1,subsample=0.
                          n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly us
# predicted probabilities on train data, as the model may be overfitting on the train data,
# may lead to misleading results
cross_val_ypred = cross_val_predict(GradientBoostingClassifier(random_state=1,max_depth=3,
                                                              learning_rate=0.1,subsample=0
                          n_estimators=200), X, y, cv = 5, method = 'predict_proba')

p, r, thresholds = precision_recall_curve(y, cross_val_ypred[:,1])
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
    plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```



```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].resha
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and incre
# the last value will provide the maximum threshold probability for the recall to be more tha
# We wish to find the maximum threshold probability to obtain the maximum possible precision
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

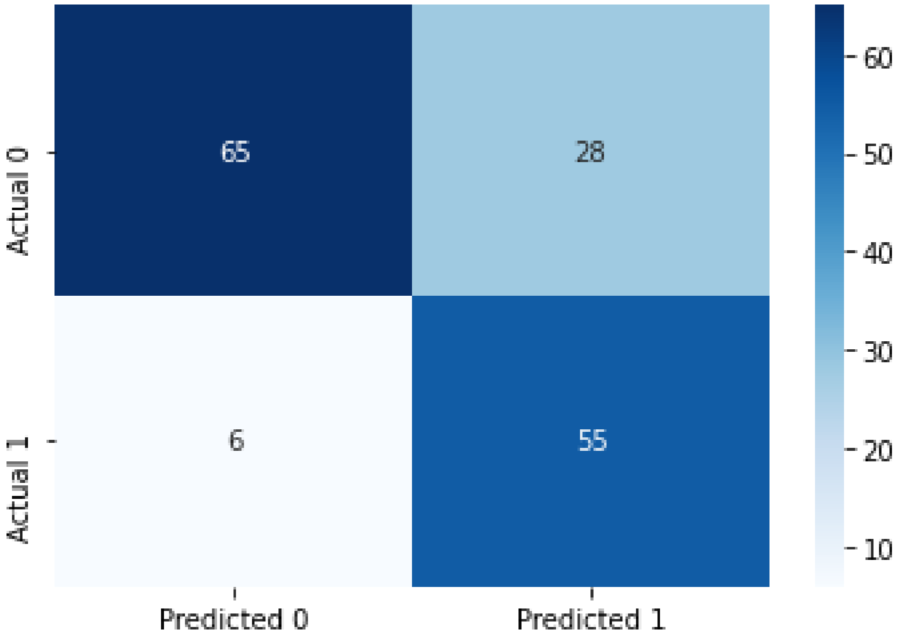
```
array([0.18497144, 0.53205128, 0.80193237])
```

```
#Optimal decision threshold probability
thres = recall_more_than_80[recall_more_than_80.shape[0]-1][0]
thres
```

```
0.18497143500912738
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = thres

y_pred_prob = model.predict_proba(Xtest)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is grea
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  77.92207792207793
ROC-AUC:  0.8704389212057112
Precision:  0.6626506024096386
Recall:  0.9016393442622951
```



The model seems to be similar to the Adaboost model. However, gradient boosting algorithms with robust loss functions can perform better than Adaboost in the presence of outliers *(in terms of response)* in the data.

10.4 Faster algorithms and tuning tips

Check out `HistGradientBoostingRegressor()` and `HistGradientBoostingClassifier()` for a faster gradient boosting algorithm for big datasets *(more than 10,000 observations)*.

Check out tips for faster hyperparameter tuning, such as tuning `max_leaf_nodes` instead of `max_depth` here.

11 XGBoost

XGBoost is a very recently developed algorithm (2016). Thus, it's not yet there in standard textbooks. Here are some resources for it.

Documentation

Slides

Reference paper

Video by author (Tianqi Chen)

Video by StatQuest

11.1 Hyperparameters

The following are some of the important hyperparameters to tune in XGBoost:

1. Number of trees (`n_estimators`)
2. Depth of each tree (`max_depth`)
3. Learning rate (`learning_rate`)
4. Sampling observations / predictors (`subsample` for observations, `colsample_bytree` for predictors)
5. Regularization parameters (`reg_lambda` & `gamma`)

However, there are other hyperparameters that can be tuned as well. Check out the list of all hyperparameters in the XGBoost documentation.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split, KFold, cross_val_predi
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_curve
```

```
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, RandomizedS
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, StackingCl
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, Elas
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

11.2 XGBoost for regression

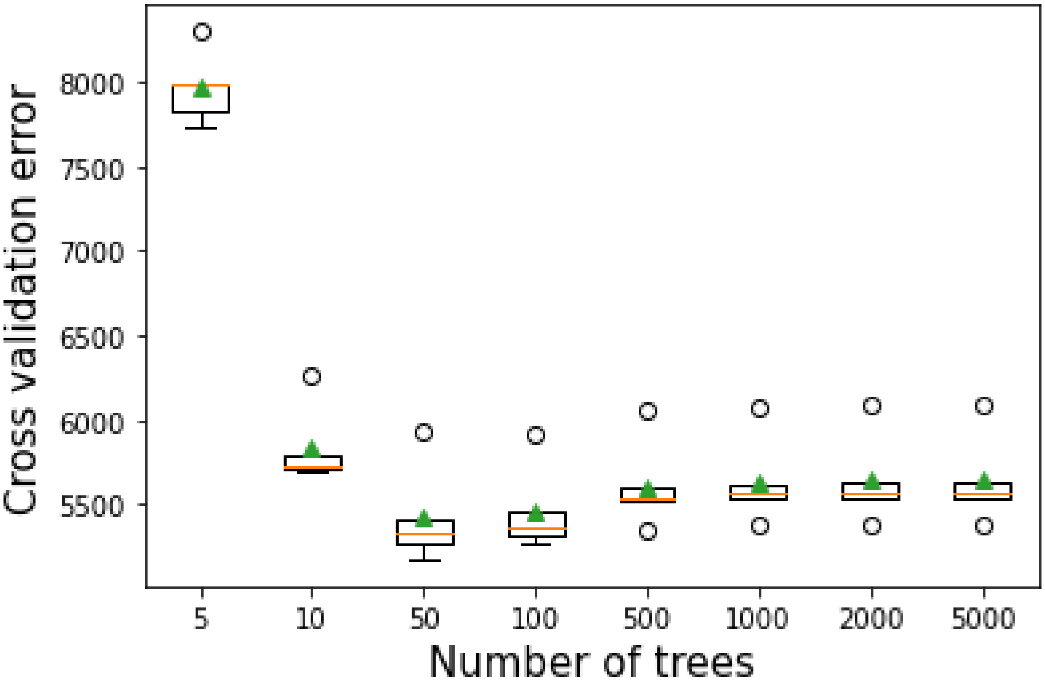
11.2.1 Number of trees vs cross validation error

As the number of trees increase, the prediction bias will decrease. Like gradient boosting is relatively robust *(as compared to AdaBoost)* to over-fitting (why?) so a large number usually results in better performance. Note that the number of trees still need to be tuned for optimal performance.

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [5, 10, 50, 100, 500, 1000, 2000, 5000]
    for n in n_trees:
        models[str(n)] = xgb.XGBRegressor(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv,
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)
```

```
>5 7961.485 (192.906)
>10 5837.134 (217.986)
>50 5424.788 (263.890)
>100 5465.396 (237.938)
>500 5608.350 (235.903)
>1000 5635.159 (236.664)
>2000 5642.669 (236.192)
>5000 5643.411 (236.074)
```

```
Text(0.5, 0, 'Number of trees')
```



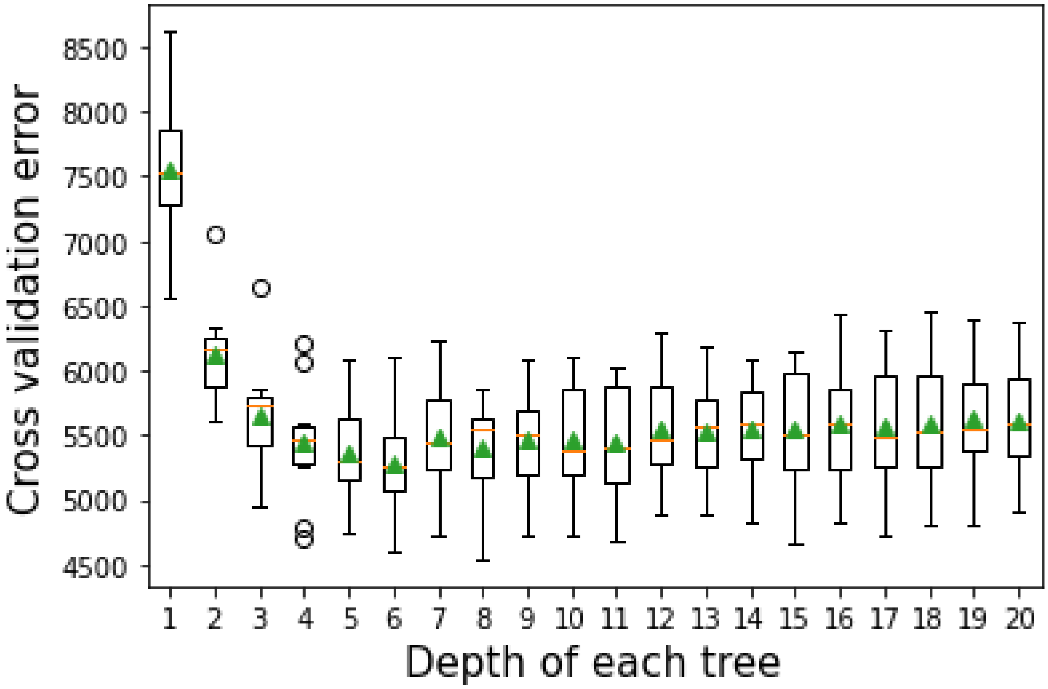
11.2.2 Depth of tree vs cross validation error

As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth of each weak learner that minimizes the prediction error.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = xgb.XGBRegressor(random_state=1,max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)
```

```
>1 7541.827 (545.951)
>2 6129.425 (393.357)
>3 5647.783 (454.318)
>4 5438.481 (453.726)
>5 5358.074 (379.431)
>6 5281.675 (383.848)
>7 5495.163 (459.356)
>8 5399.145 (380.437)
>9 5469.563 (384.004)
```

```
>10 5461.549 (416.630)
>11 5443.210 (432.863)
>12 5546.447 (412.097)
>13 5532.414 (369.131)
>14 5556.761 (362.746)
>15 5540.366 (452.612)
>16 5586.004 (451.199)
>17 5563.137 (464.344)
>18 5594.919 (480.221)
>19 5641.226 (451.713)
>20 5616.462 (417.405)

Text(0.5, 0, 'Depth of each tree')
```



11.2.3 Learning rate vs cross validation error

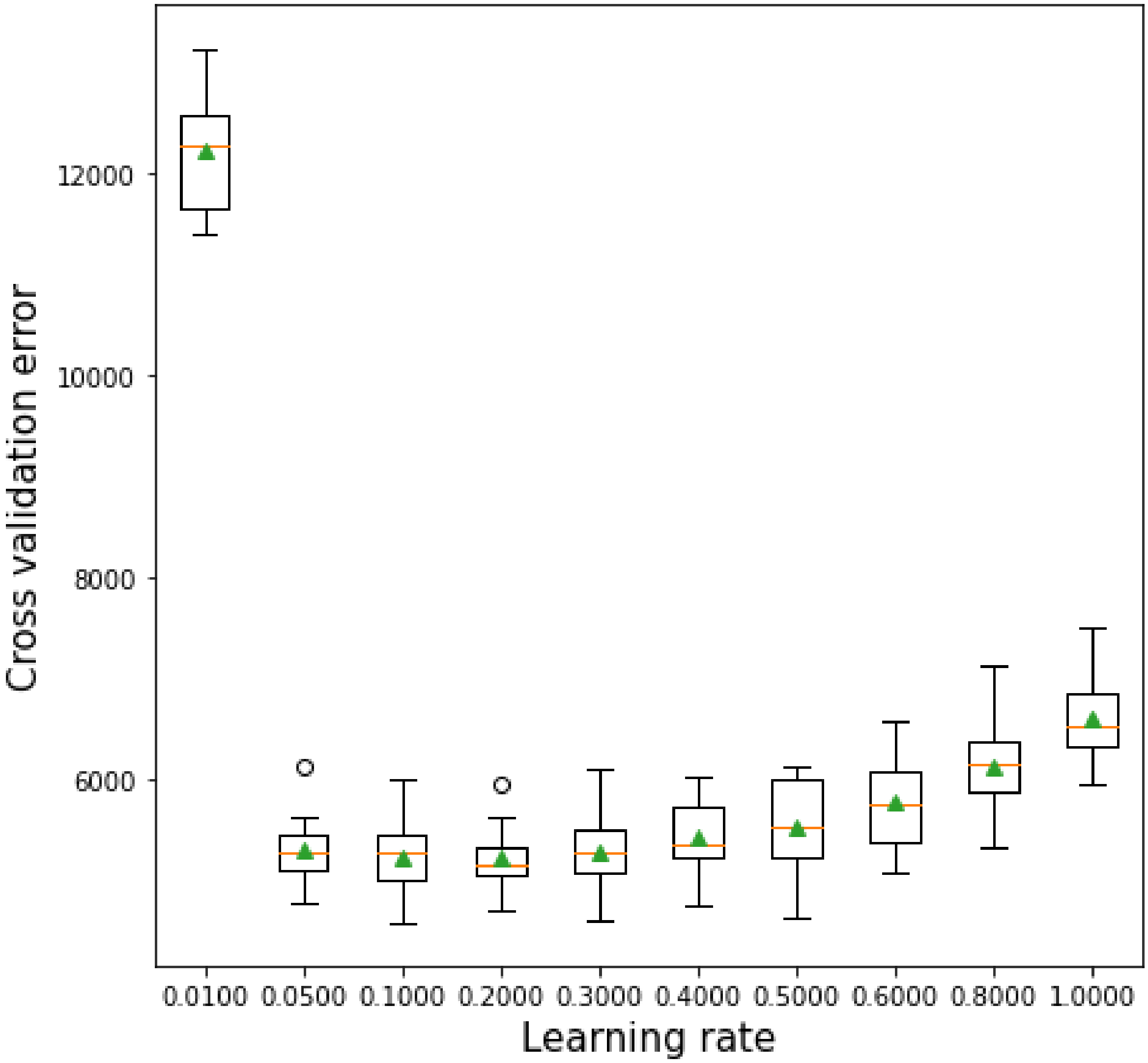
The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to “learn” the response. If the learning rate is high, the response will be “learned” quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in [0.01,0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.8,1.0]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(learning_rate=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv,
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)
```

```
>0.0100 12223.8 (636.7)
>0.0500 5298.5 (383.5)
>0.1000 5236.3 (397.5)
>0.2000 5221.5 (347.5)
>0.3000 5281.7 (383.8)
>0.4000 5434.1 (364.6)
>0.5000 5537.0 (471.9)
>0.6000 5767.4 (478.5)
```

```
>0.8000 6132.7 (472.5)
>1.0000 6593.6 (408.9)
```

```
Text(0.5, 0, 'Learning rate')
```



11.2.4 Regularization (`reg_lambda`) vs cross validation error

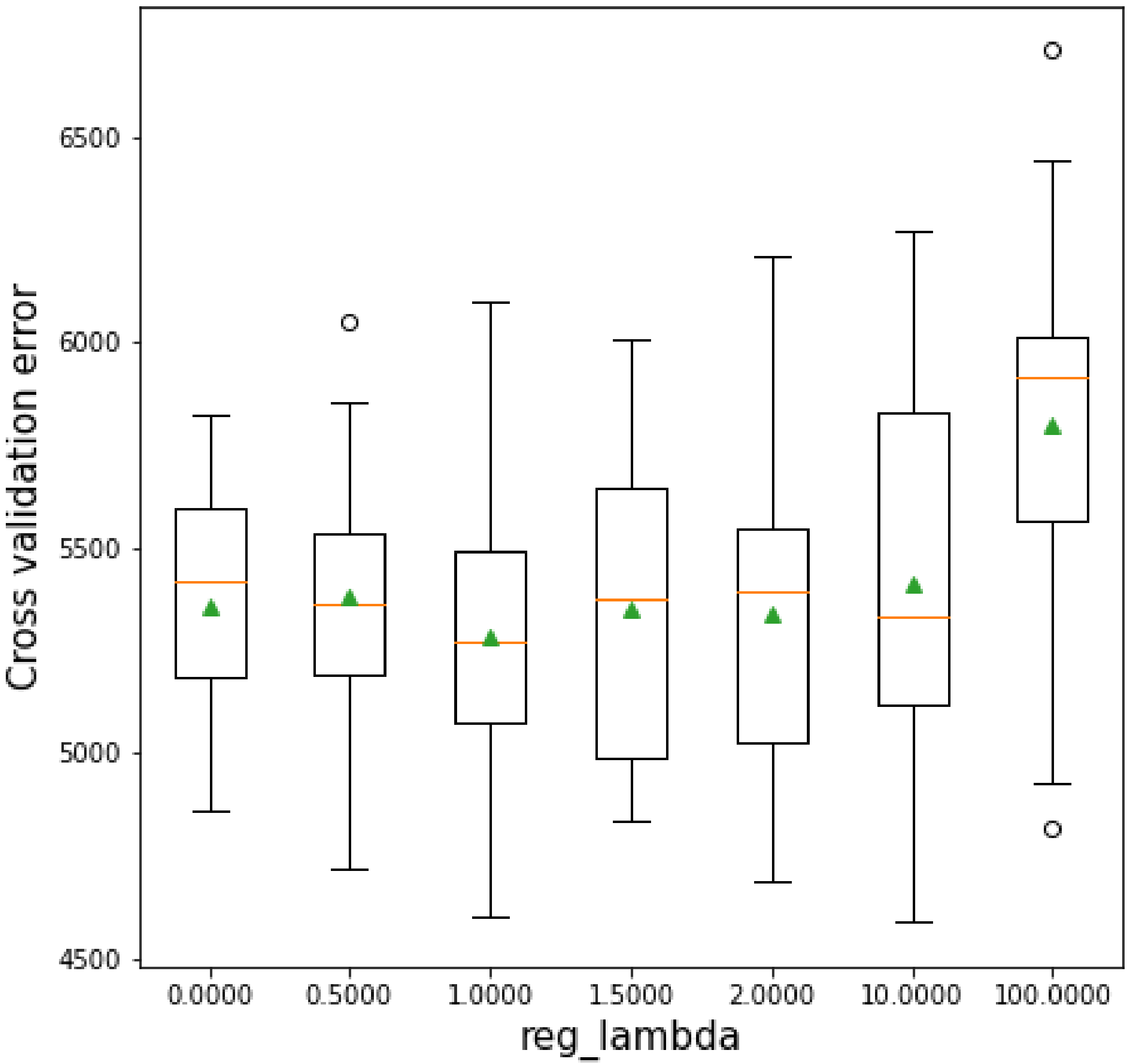
The parameter `reg_lambda` penalizes the *L2 norm* of the leaf scores. For example, in case of classification, it will penalize the summation of the square of log odds of the predicted probability. This penalization will tend to reduce the log odds, thereby reducing the tendency to overfit. "*Reducing the log odds*" in layman terms will mean not being overly sure about the prediction.

Without regularization, the algorithm will be closer to the gradient boosting algorithm. Regularization may provide some additional boost to prediction accuracy by reducing over-fitting. In the example below, regularization with *reg_lambda=1 turns out to be better than no regularization* (reg_lambda=0)*. Of course, too much regularization may increase bias so much such that it leads to a decrease in prediction accuracy.

```
def get_models():
    models = dict()
    # explore 'reg_lambda' from 0.1 to 2 in 0.1 increments
    for i in [0,0.5,1.0,1.5,2,10,100]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(reg_lambda=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, 
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('reg_lambda',fontsize=15)
```

```
>0.0000 5359.2 (317.0)
>0.5000 5382.7 (363.1)
>1.0000 5281.7 (383.8)
>1.5000 5348.0 (383.9)
>2.0000 5336.4 (426.6)
>10.0000 5410.9 (521.9)
>100.0000 5801.1 (563.7)
```

```
Text(0.5, 0, 'reg_lambda')
```



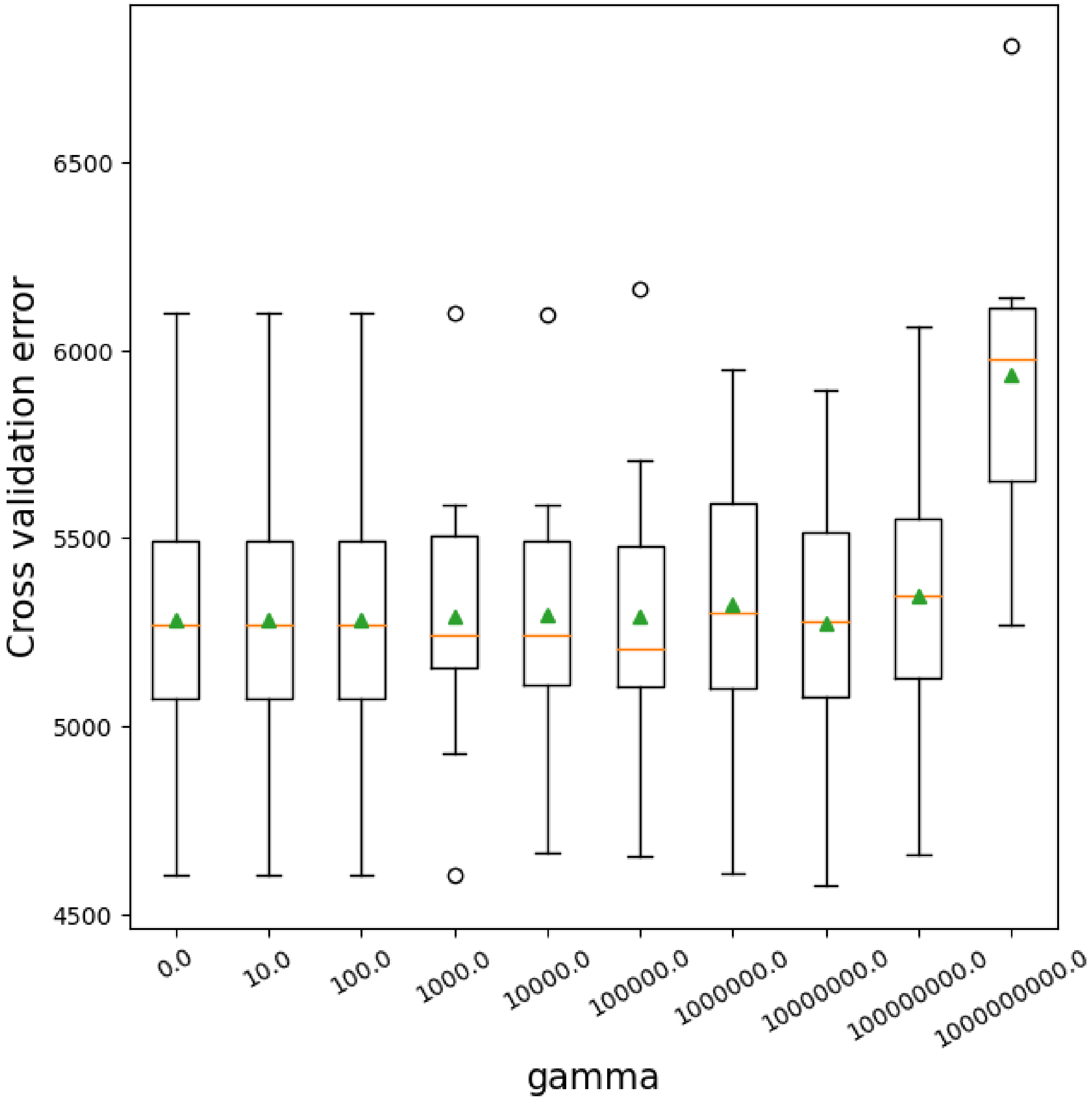
11.2.5 Regularization (`gamma`) vs cross validation error

The parameter `gamma` penalizes the tree based on the number of leaves. This is similar to the parameter `alpha` of cost complexity pruning. As `gamma` increases, more leaves will be pruned. Note that the previous parameter `reg_lambda` penalizes the leaf score, but does not prune the tree.

Without regularization, the algorithm will be closer to the gradient boosting algorithm. Regularization may provide some additional boost to prediction accuracy by reducing over-fitting. However, in the example below, no regularization (in terms of `gamma=0`) turns out to be better than a non-zero regularization. *(reg_lambda=0)*.

```
def get_models():
    models = dict()
    # explore gamma from 0.1 to 2 in 0.1 increments
    for i in [0,10,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(gamma=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('gamma',fontsize=15)
#ax.set_xticklabels(x.astype(int))
plt.xticks(ticks=plt.xticks()[0].astype(int), labels=np.round([0,10,1e2,1e3,1e4,1e5,1e6,1e7,
          rotation = 30);
```

```
>0.0000 5281.7 (383.8)
>10.0000 5281.7 (383.8)
>100.0000 5281.7 (383.8)
>1000.0000 5291.8 (381.8)
>10000.0000 5295.7 (370.2)
>100000.0000 5293.0 (402.5)
>1000000.0000 5322.2 (368.9)
>10000000.0000 5273.7 (409.8)
>100000000.0000 5345.4 (373.9)
>1000000000.0000 5932.3 (397.6)
```



11.2.6 Tuning XGboost regressor

Along with `max_depth`, `learning_rate`, and `n_estimators`, here we tune `reg_lambda` - the regularization parameter for penalizing the tree predictions.

```
#K-fold cross validation to find optimal parameters for XGBoost
start_time = time.time()
param_grid = {'max_depth': [4,6,8],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 1, 10],
                'n_estimators':[100, 500, 1000],
                'gamma': [0, 10, 100],
                'subsample': [0.5, 0.75, 1.0],
                'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=xgb.XGBRegressor(random_state=1),
                             param_distributions = param_grid, n_iter = 200,
                             verbose = 1,
                             n_jobs=-1,
                             cv = cv)
optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation R-squared = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Optimal parameter values = {'subsample': 0.75, 'reg_lambda': 1, 'n_estimators': 1000, 'max_de
Optimal cross validation R-squared =  0.9002580404500382
Time taken =  4  minutes
```

```
#RMSE based on the optimal parameter values
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

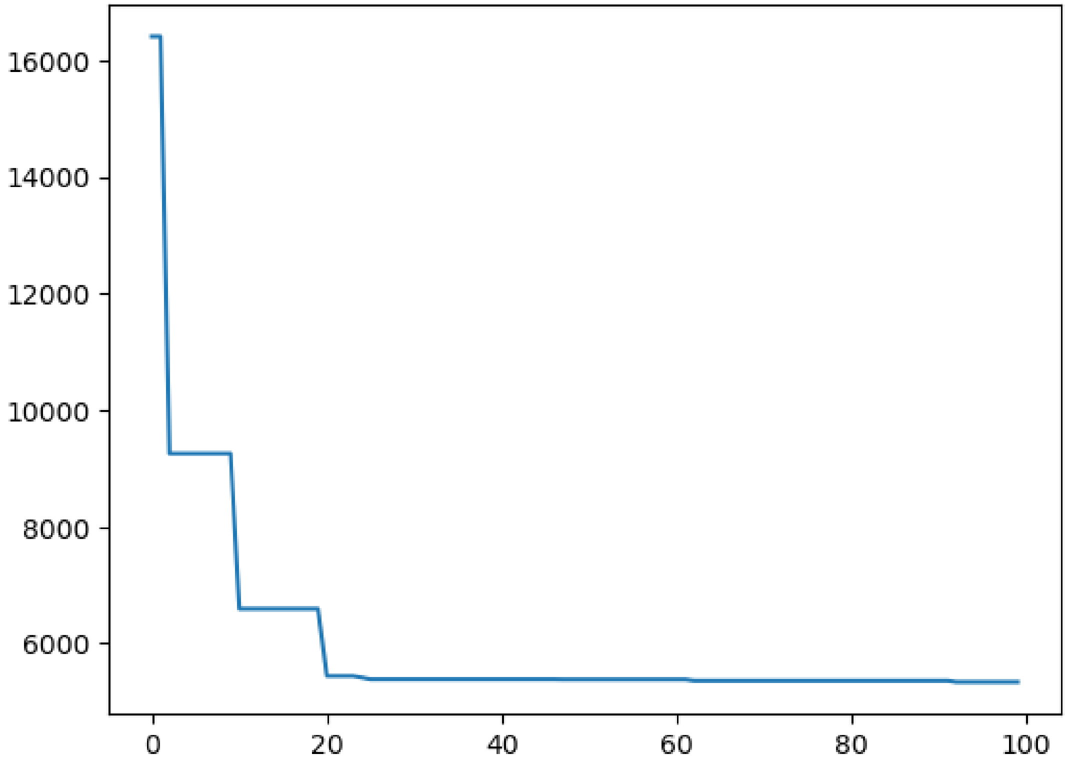
```
5497.553788113875
```

Let us use Bayes search to tune the model.

```
model = xgb.XGBRegressor(random_state = 1)

grid = {'max_leaves': Integer(4, 5000),
              'learning_rate': Real(0.0001, 1.0),
               'reg_lambda':Real(0, 1e4),
                'n_estimators':Integer(2, 2000),
                'gamma': Real(0, 1e11),
                'subsample': Real(0.1,1.0),
                'colsample_bytree': Real(0.1, 1.0)}
```

```
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 100, random_state = 1,
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
['colsample_bytree', 'gamma', 'learning_rate', 'max_leaves', 'n_estimators', 'reg_lambda', 's
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
              estimator=XGBRegressor(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False, eval_metric=None,
                                     feature_types=None, gamma=None,
                                     gpu_id=None, grow_policy=None,
                                     importance_type=None,
                                     inte...
                             'learning_rate': Real(low=0.0001, high=1.0, prior='uniform', tra
                             'max_leaves': Integer(low=4, high=5000, prior='uniform', transfo
                             'n_estimators': Integer(low=2, high=2000, prior='uniform', trans
                             'reg_lambda': Real(low=0, high=10000.0, prior='uniform', transfo
                             'subsample': Real(low=0.1, high=1.0, prior='uniform', transform=
```

```
model1 = xgb.XGBRegressor(random_state = 1, colsample_bytree = 0.85, gamma = 0, learning_rate
                          max_leaves = 802, n_estimators = 1023, reg_lambda = 1394, subsample
```

```
np.sqrt(mean_squared_error(model1.predict(Xtest),ytest))
```

```
5466.076861800755
```

We got a different set of optimal hyperparameters with Bayes search. Thus, ensembling the model based on the two sets of hyperparameters is likely to improve the accuracy over the individual models.

```
model2 = xgb.XGBRegressor(random_state = 1, colsample_bytree = 1.0, gamma = 100, learning_rat
                          max_depth = 8, n_estimators = 1000, reg_lambda = 1, subsample = 0.7
```

```
np.sqrt(mean_squared_error(0.5*model1.predict(Xtest)+0.5*model2.predict(Xtest),ytest))
```

```
5393.379834226845
```

11.2.7 Early stopping with XGBoost

If we have a test dataset *(or we can further split the train data into a smaller train and test data)*, we can use it with the `early_stopping_rounds` argument of XGBoost, where it will stop growing trees once the model accuracy fails to increase for a certain number of consecutive iterations, given as `early_stopping_rounds`.

```
X_train_sub, X_test_sub, y_train_sub, y_test_sub = \
train_test_split(X, y, test_size = 0.2, random_state = 45)
```

```
model = xgb.XGBRegressor(random_state = 1, max_depth = 8, learning_rate = 0.01,
                        n_estimators = 20000,reg_lambda = 1, gamma = 100, subsample = 0.75, c
model.fit(X_train_sub, y_train_sub, eval_set = ([(X_test_sub, y_test_sub)]), early_stopping_r
```

The results of the code are truncated to save space. A snapshot of the beginning and end of the results is below. The algorithm keeps adding trees to the model until the RMSE ceases to decrease for 250 consecutive iterations.

```
<IPython.core.display.Image object>
```

```
print("XGBoost RMSE = ",np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

```
XGBoost RMSE =  5508.787454011525
```

Let us further reduce the `learning rate` to `0.001` and see if the accuracy increases further on the test data. We'll use the `early_stopping_rounds` argument to stop growing trees once the accuracy fails to increase for 250 consecutive iterations.

```
model = xgb.XGBRegressor(random_state = 1, max_depth = 8, learning_rate = 0.001,
                        n_estimators = 20000,reg_lambda = 1, gamma = 100, subsample = 0.75, c
model.fit(X_train_sub, y_train_sub, eval_set = ([(X_test_sub, y_test_sub)]), early_stopping_r
```

```
<IPython.core.display.Image object>
```

```
print("XGBoost RMSE = ",np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

```
XGBoost RMSE =  5483.518711988693
```

Note that the accuracy on this test data has further increased with a lower learning rate.

Let us combine the XGBoost model with other tuned models from earlier chapters.

```
#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimator
                         random_state=1).fit(X,y)
print("AdaBoost RMSE = ", np.sqrt(mean_squared_error(model_ada.predict(Xtest),ytest)))

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                        n_jobs=-1, max_features=2).fit(X, y)
print("Random Forest RMSE = ",np.sqrt(mean_squared_error(model_rf.predict(Xtest),ytest)))

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                         random_state=1,loss='huber').fit(X,y)
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model_gb.predict(Xtest),ytest)))
```

```
AdaBoost RMSE =  5693.165811600585
Random Forest RMSE =  5642.45839697972
Gradient boost RMSE =  5405.787029062213
```

```
#Ensemble model
pred_xgb = model.predict(Xtest)    #XGBoost
pred_ada = model_ada.predict(Xtest)#AdaBoost
pred_rf = model_rf.predict(Xtest)  #Random Forest
pred_gb = model_gb.predict(Xtest)  #Gradient boost
pred = 0.25*pred_xgb + 0.25*pred_ada + 0.25*pred_rf + 0.25*pred_gb #Option 1 - All models ar
#pred = 0.15*pred1+0.15*pred2+0.15*pred3+0.55*pred4 #Option 2 - Higher weight to the better r
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

```
Ensemble model RMSE =  5352.145010078119
```

Combined, the random forest model, gradient boost, XGBoost and the Adaboost model do better than each of the individual models.

11.3 XGBoost for classification

```
data = pd.read_csv('./Datasets/Heart.csv')
data.dropna(inplace = True)
data.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca
0	63	1	typical	145	233	1	2	150	0	2.3	3	0.0
1	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0
2	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0
3	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0
4	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0

```
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy varibles replacing the categorical variables
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)
X.head()
```

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
0	63	1	145	233	1	2	150	0	2.3	3	0.0	0
1	67	1	160	286	0	2	108	1	1.5	2	3.0	1
2	67	1	120	229	0	2	129	1	2.6	2	2.0	1
3	37	1	130	250	0	0	187	0	3.5	3	0.0	0
4	41	0	130	204	0	2	172	0	1.4	1	0.0	0

```
#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)
```

XGBoost has an additional parameter for classification: **`scale_pos_weight`**

Gradients are used as the basis for fitting subsequent trees added to boost or correct errors made by the existing state of the ensemble of decision trees.

The `scale_pos_weight` value is used to scale the gradient for the positive class.

This has the effect of scaling errors made by the model during training on the positive class and encourages the model to over-correct them. In turn, this can help the model achieve better performance when making predictions on the positive class. Pushed too far, it may result in the model overfitting the positive class at the cost of worse performance on the negative class or both classes.

As such, the `scale_pos_weight` hyperparameter can be used to train a class-weighted or cost-sensitive version of XGBoost for imbalanced classification.

A sensible default value to set for the `scale_pos_weight` hyperparameter is the inverse of the class distribution. For example, for a dataset with a 1 to 100 ratio for examples in the minority to majority classes, the `scale_pos_weight` can be set to 100. This will give classification errors made by the model on the minority class (positive class) 100 times more impact, and in turn, 100 times more correction than errors made on the majority class.

Reference

```
start_time = time.time()
param_grid = {'n_estimators':[25,100,500],
                'max_depth': [6,7,8],
              'learning_rate': [0.01,0.1,0.2],
               'gamma': [0.1,0.25,0.5],
               'reg_lambda':[0,0.01,0.001],
                'scale_pos_weight':[1.25,1.5,1.75]#Control the balance of positive and negati
             }

cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = GridSearchCV(estimator=xgb.XGBClassifier(objective = 'binary:logistic',rando
                                                          use_label_encoder=False),
                             param_grid = param_grid,
                             scoring = 'accuracy',
                             verbose = 1,
                             n_jobs=-1,
                             cv = cv)
optimal_params.fit(Xtrain,ytrain)
print(optimal_params.best_params_,optimal_params.best_score_)
print("Time taken = ", (time.time()-start_time)/60, " minutes")
```

```
Fitting 5 folds for each of 729 candidates, totalling 3645 fits
[22:00:02] WARNING: D:\bld\xgboost-split_1645118015404\work\src\learner.cc:1115: Starting in
{'gamma': 0.25, 'learning_rate': 0.2, 'max_depth': 6, 'n_estimators': 25, 'reg_lambda': 0.01
```

```
cv_results=pd.DataFrame(optimal_params.cv_results_)
cv_results.sort_values(by = 'mean_test_score',ascending=False)[0:5]
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_gamma	param_learnin
409	0.111135	0.017064	0.005629	0.000737	0.25	0.2
226	0.215781	0.007873	0.005534	0.001615	0.1	0.2
290	1.391273	0.107808	0.007723	0.006286	0.25	0.01
266	1.247463	0.053597	0.006830	0.002728	0.25	0.01

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_gamma	param_learnin
269	1.394361	0.087307	0.005530	0.001718	0.25	0.01

```
#Function to compute confusion matrix and prediction accuracy on test/train data
def confusion_matrix_data(data,actual_values,model,cutoff=0.5):
#Predict the values using the Logit model
    pred_values = model.predict_proba(data)[:,1]
# Specify the bins
    bins=np.array([0,cutoff,1])
#Confusion matrix
    cm = np.histogram2d(actual_values, pred_values, bins=bins)[0]
    cm_df = pd.DataFrame(cm)
    cm_df.columns = ['Predicted 0','Predicted 1']
    cm_df = cm_df.rename(index={0: 'Actual 0',1:'Actual 1'})
# Calculate the accuracy
    accuracy = 100*(cm[0,0]+cm[1,1])/cm.sum()
    fnr = 100*(cm[1,0])/(cm[1,0]+cm[1,1])
    precision = 100*(cm[1,1])/(cm[0,1]+cm[1,1])
    fpr = 100*(cm[0,1])/(cm[0,0]+cm[0,1])
    tpr = 100*(cm[1,1])/(cm[1,0]+cm[1,1])
    print("Accuracy = ", accuracy)
    print("Precision = ", precision)
    print("FNR = ", fnr)
    print("FPR = ", fpr)
    print("TPR or Recall = ", tpr)
    print("Confusion matrix = \n", cm_df)
    return (" ")
```

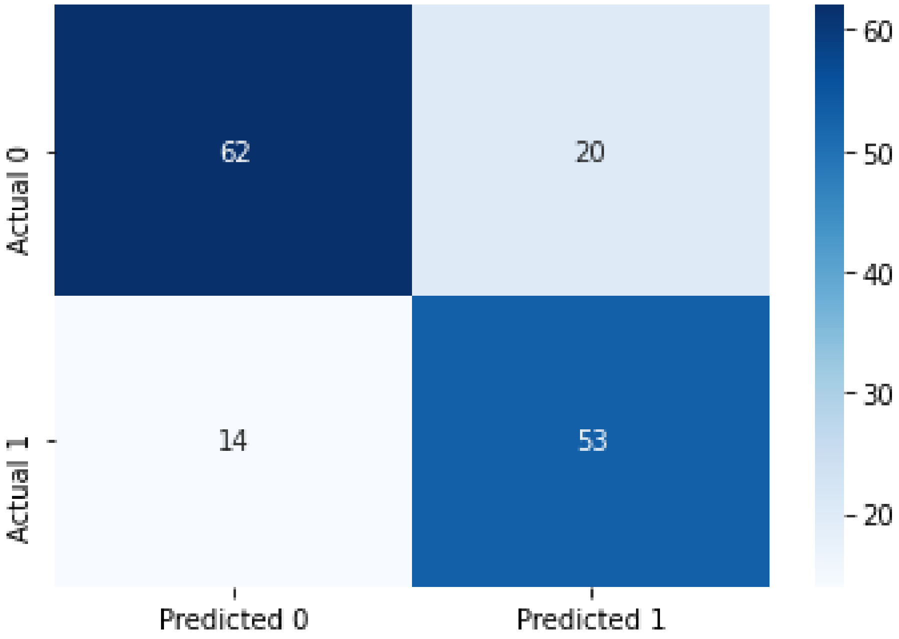
```
model4 = xgb.XGBClassifier(objective = 'binary:logistic',random_state=1,gamma=0.25,learning_
                              n_estimators = 500,reg_lambda = 0.01,scale_pos_weight=1.75)
model4.fit(Xtrain,ytrain)
model4.score(Xtest,ytest)
```

```
0.7718120805369127
```

```
#Computing the accuracy
y_pred = model4.predict(Xtest)
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
```

```
y_pred_prob = model4.predict_proba(Xtest)[:,1]
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred), columns=['Predicted 0', 'Predicted 1'],
            index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  77.18120805369128
ROC-AUC:  0.8815070986530761
Precision:  0.726027397260274
Recall:  0.7910447761194029
```



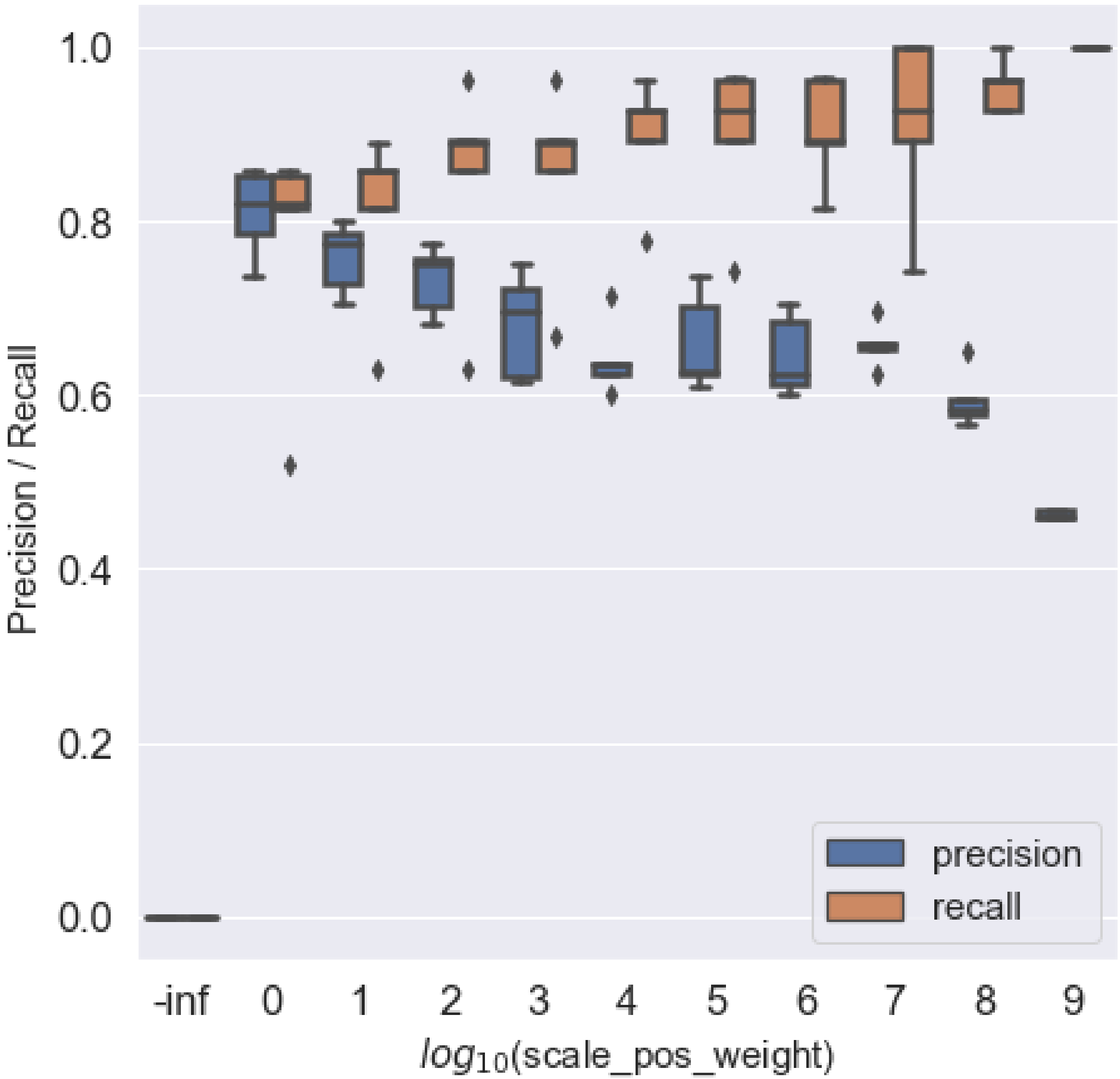
If we increase the value of `scale_pos_weight`, the model will focus on classifying positives more correctly. This will increase the recall (true positive rate) since the focus is on identifying all positives. However, this will lead to identifying positives aggressively, and observations ‘similar’ to observations of the positive class will also be predicted as positive resulting in an increase in false positives and a decrease in precision. See the trend below as we increase the value of `scale_pos_weight`.

11.3.1 Precision & recall vs `scale_pos_weight`

```
def get_models():
    models = dict()
    # explore 'scale_pos_weight' from 0.1 to 2 in 0.1 increments
    for i in [0,1,10,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9]:
        key = '%.0f' % i
        models[key] = xgb.XGBClassifier(objective = 'binary:logistic',scale_pos_weight=i,ran
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores_recall = cross_val_score(model, X, y, scoring='recall', cv=cv, n_jobs=-1)
    scores_precision = cross_val_score(model, X, y, scoring='precision', cv=cv, n_jobs=-1)
    return list([scores_recall,scores_precision])

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results_recall, results_precision, names = list(), list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    scores_recall = scores[0]
    scores_precision = scores[1]
    # store the results
    results_recall.append(scores_recall)
    results_precision.append(scores_precision)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores_recall), np.std(scores_recall)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
sns.set(font_scale = 1.5)
pdata = pd.DataFrame(results_precision)
```

```
pdata.columns = list(['p1','p2','p3','p4','p5'])
pdata['metric'] = 'precision'
rdata = pd.DataFrame(results_recall)
rdata.columns = list(['p1','p2','p3','p4','p5'])
rdata['metric'] = 'recall'
pr_data = pd.concat([pdata,rdata])
pr_data.reset_index(drop=False,inplace= True)
#sns.boxplot(x="day", y="total_bill", hue="time",pr_data=tips, linewidth=2.5)
pr_data_melt=pr_data.melt(id_vars = ['index','metric'])
pr_data_melt['index']=pr_data_melt['index']-1
pr_data_melt['index'] = pr_data_melt['index'].astype('str')
pr_data_melt.replace(to_replace='-1',value =  '-inf',inplace=True)
sns.boxplot(x='index', y="value", hue="metric", data=pr_data_melt, linewidth=2.5)
plt.xlabel('$log_{10}$(scale_pos_weight)',fontsize=15)
plt.ylabel('Precision / Recall ',fontsize=15)
plt.legend(loc="lower right", frameon=True, fontsize=15)
```

```
>0 0.00 (0.00)
>1 0.77 (0.13)
>10 0.81 (0.09)
>100 0.85 (0.11)
>1000 0.85 (0.10)
>10000 0.90 (0.06)
>100000 0.90 (0.08)
>1000000 0.90 (0.06)
>10000000 0.91 (0.10)
>100000000 0.96 (0.03)
>1000000000 1.00 (0.00)
```


1.0
0.8
0.6
0.4
0.2
0.0
Precision / Recall
-inf
0
1
2
3
4
5
6
7
8
9
log_{10}(scale_pos_weight)
precision
recall

12 LightGBM and CatBoost

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split, KFold, cross_val_predi
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, RandomizedS
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, StackingCl
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, Elas
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

We'll continue to use the same datasets that we have been using throughout the course.

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
```

```
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

12.1 LightGBM

LightGBM is a gradient boosting decision tree algorithm developed by Microsoft in 2017. LightGBM outperforms XGBoost in terms of compuational speed, and provides comparable accuracy in general. The following two key features in LightGBM that make it faster than XGBoost:

1. Gradient-based One-Side Sampling (GOSS): Recall, in gradient boosting, we fit trees on the gradient of the loss function *(refer the gradient boosting algorithm in section 10.10.2 of the book, Elements of Statistical Learning):*

$$r_m = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

Observations that correspond to relatively larger gradients contribute more to minimizing the loss function as compared to observations with smaller gradients. The algorithm down-samples the observations with small gradients, while selecting all the observations with large gradients. As observations with large gradients contribute the most to the reduction in loss function when considering a split, the accuracy of loss reduction estimate is maintained even with a reduced sample size. This leads to similar performance in terms of prediction accuracy while reducing computation speed due to reduction in sample size to fit trees.

2. Exclusive feature bundling (EFB): This is useful when there are a lot of predictors, but the predictor space is sparse, i.e., most of the values are zero for several predictors, and the predictors rarely take non-zero values simultaneously. This can typically happen in case of a lot of dummy variables in the data. In such a case, the predictors are bundled to create a single predictor.

In the example below you can see that feature1 and feature2 are mutually exclusive. In order to achieve non overlapping buckets we add bundle size of feature1 to feature2. This makes sure that non zero data points of bundled features (feature1 and feature2) reside in different buckets. In feature_bundle buckets 1 to 4 contains non zero instances of feature1 and buckets 5,6 contain non zero instances of feature2 *(Reference)*.

feature1	feature2	feature_bundle
0	2	6
0	1	5
0	2	6
1	0	1
2	0	2
3	0	3
4	0	4

Read the LightGBM paper for more details.

12.1.1 LightGBM for regression

Let us tune a lightGBM model for regression for our problem of predicting car price. We'll use the function `LGBMRegressor`. For classification problems, `LGBMClassifier` can be used. Note that we are using the GOSS algorithm to downsample observations with smaller gradients.

```
#K-fold cross validation to find optimal parameters for LightGBM regressor
start_time = time.time()
param_grid = {'num_leaves': [20, 31, 40],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 10, 100],
                'n_estimators':[100, 500, 1000],
                'reg_alpha': [0, 10, 100],
                'subsample': [0.5, 0.75, 1.0],
                'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=LGBMRegressor(boosting_type = 'goss'),
                             param_distributions = param_grid, n_iter = 200,
                             verbose = 1, scoring='neg_root_mean_squared_error',
                             n_jobs=-1,random_state=1,
                             cv = cv)
optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Optimal parameter values = {'subsample': 1.0, 'reg_lambda': 10, 'reg_alpha': 0, 'num_leaves'
Optimal cross validation R-squared =  -5670.309021679375
Time taken =  1  minutes
```

```
#RMSE based on the optimal parameter values of a LighGBM Regressor model
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

```
5614.374498193448
```

Note that downsampling of small-gradient observations leads to faster execution time, but potentially by compromising some accuracy. We can expect to improve the accuracy by increasing the `top_rate` or the `other_rate` hyperparameters, but at an increased computational cost. In the cross-validation below, we have increased the `top_rate` to 0.5 from the default value of 0.2.

```
#K-fold cross validation to find optimal parameters for LightGBM regressor
start_time = time.time()
param_grid = {'num_leaves': [20, 31, 40],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 10, 100],
                'n_estimators':[100, 500, 1000],
                'reg_alpha': [0, 10, 100],
                'subsample': [0.5, 0.75, 1.0],
                'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=LGBMRegressor(boosting_type = 'goss', top_rate
                             param_distributions = param_grid, n_iter = 200,
                             verbose = 1, scoring='neg_root_mean_squared_error',
                             n_jobs=-1,random_state=1,
                             cv = cv)
optimal_params.fit(X,y)
```

```
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Optimal parameter values = {'subsample': 0.5, 'reg_lambda': 0, 'reg_alpha': 100, 'num_leaves
Optimal cross validation R-squared =  -5436.062435616846
Time taken =  1  minutes
```

```
#RMSE based on the optimal parameter values of a LighGBM Regressor model
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

```
5355.964600884197
```

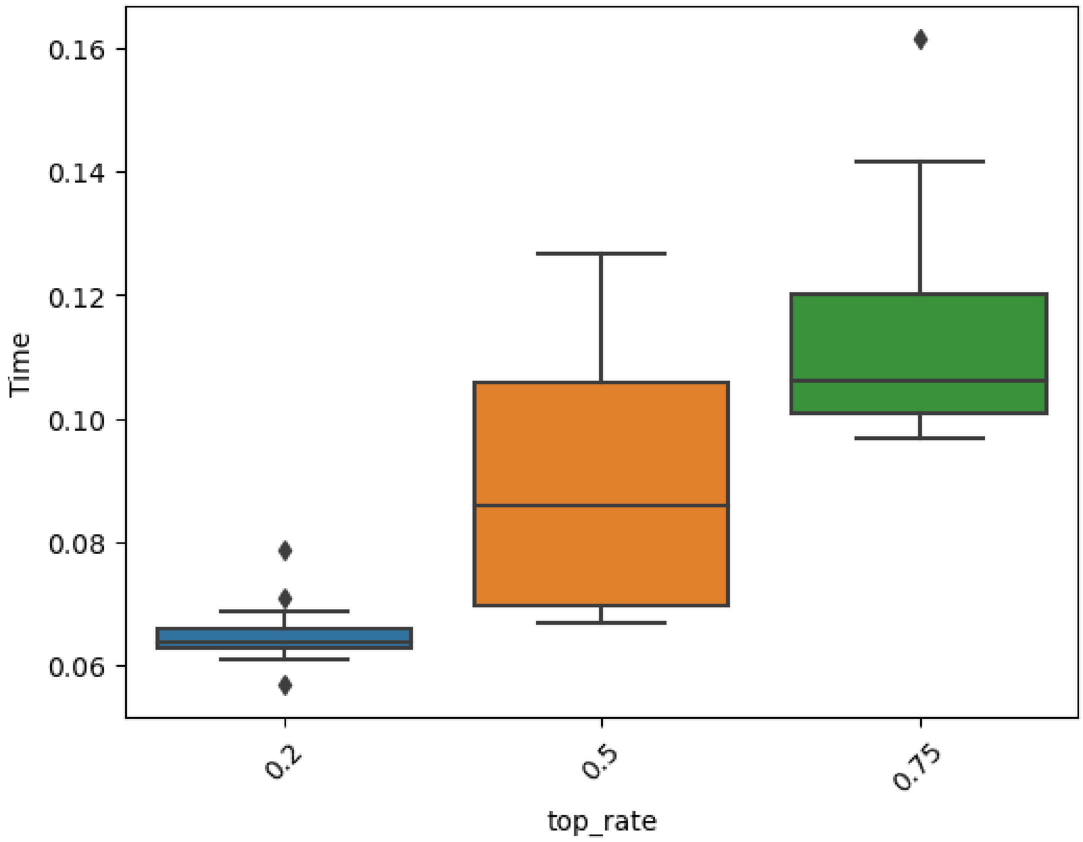
Note that the cross-validated RMSE has reduced. However, this is at an increased computational expense. In the simulations below, we compare the time taken to train models with increasing values of the `top_rate` hyperparameter.

```
time_list = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.2, n_jobs=-1).fit(X, y)
    time_list.append(time.time()-start_time)
```

```
time_list2 = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.5, n_jobs=-1).fit(X, y)
    time_list2.append(time.time()-start_time)
```

```
time_list3 = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.8, n_jobs=-1).fit(X, y)
    time_list3.append(time.time()-start_time)
```

```
ax = sns.boxplot([time_list, time_list2, time_list3]);
ax.set_xticklabels([0.2, 0.5, 0.75]);
plt.ylabel('Time');
plt.xlabel('top_rate');
plt.xticks(rotation = 45);
```



12.1.2 LightGBM vs XGBoost

LightGBM model took 2 minutes for a random search with 1000 fits as compared to 7 minutes for an XGBoost model with 1000 fits on the same data (as shown below). In terms of prediction accuracy, we observe that the accuracy of LightGBM on test *(unseen)* data is comparable to that of XGBoost.

```
#K-fold cross validation to find optimal parameters for XGBoost
start_time = time.time()
param_grid = {'max_depth': [4,6,8],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 1, 10],
                'n_estimators':[100, 500, 1000],
                'gamma': [0, 10, 100],
                'subsample': [0.5, 0.75, 1.0],
                'colsample_bytree': [0.5, 0.75, 1.0]}
```

```
cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=xgb.XGBRegressor(),
                             param_distributions = param_grid, n_iter = 200,
                             verbose = 1, scoring = 'neg_root_mean_squared_error',
                             n_jobs=-1,random_state = 1,
                             cv = cv)
optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation R-squared = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Optimal parameter values = {'subsample': 0.75, 'reg_lambda': 1, 'n_estimators': 1000, 'max_de
Optimal cross validation R-squared =  -5178.8689594137295
Time taken =  7  minutes
```

```
#RMSE based on the optimal parameter values
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

```
5420.661056398766
```

12.2 CatBoost

CatBoost is a gradient boosting algorithm developed by Yandex *(Russian Google)* in 2017. Like LightGBM, CatBoost is also faster than XGBoost in training. However, unlike LightGBM, the authors have claimed that it outperforms both LightGBM and XGBoost in terms of prediction accuracy as well.

The key feature of CatBoost that address the issue with the gradient boosting procedure is the idea of ordered boosting. Classic boosting algorithms are prone to overfitting on small/noisy datasets due to a problem known as prediction shift. Recall, in gradient boosting, we fit trees on the gradient of the loss function *(refer the gradient boosting algorithm in section 10.10.2 of the book, Elements of Statistical Learning):*

$$r_m = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

When calculating the gradient estimate of an observation, these algorithms use the same observations that the model was built with, thus having no chances of experiencing unseen data. CatBoost, on the other hand, uses the concept of ordered boosting, a permutation-driven approach to train model on a subset of data while calculating residuals on another subset, thus preventing "target leakage" and overfitting. The residuals of an observation are computed based on a model developed on the previous observations, where the observations are randomly shuffled at each iteration, i.e., for each tree.

Thus, the gradient of the loss function is based on test *(unseen)* data, instead of the data on which the model has been trained, which improves the generalizability of the model, and avoids overfitting on train data.

The authors have also shown that CatBoost performs better than XGBoost and LightGBM without tuning, i.e., with default hyperparameter settings.

Read the CatBoost paper for more details.

Here is a good blog listing the key features of CatBoost.

12.2.1 CatBoost for regression

We'll use the function CatBoostRegressor for regression. For classification problems CatBoostClassifier can be used.

Let us check the performance of `CatBoostRegressor()` without tuning, i.e., with default hyperparameter settings.

```
model_cat = CatBoostRegressor(verbose=0).fit(X, y)
```

```
cv = KFold(n_splits=5,shuffle=True,random_state=1)
np.mean(-cross_val_score(CatBoostRegressor(verbose=0), X, y, cv = cv, n_jobs = -1,
                 scoring='neg_root_mean_squared_error'))
```

```
5035.972129299527
```

```
np.sqrt(mean_squared_error(model_cat.predict(Xtest),ytest))
```

```
5288.82153844634
```

Even with default hyperparameter settings, CatBoost has outperformed both XGBoost and LightGBM in terms of cross-validated RMSE, and RMSE on test data for our example of predicting car prices.

12.2.2 Target encoding with CatBoost

Target encoding for categorical variables can be used with CatBoost, that may benefit in terms of both speed and accuracy. However, the benefit is not gauranteed. Let us use target encoding for the categorical predictors `brand`, `model`, `transmission` and `fuelType`.

```
X = train[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']
Xtest = test[['mileage','mpg','year','engineSize', 'brand', 'model',  'transmission', 'fuelTy
y = train['price']
ytest = test['price']
```

The parameter `cat_features` will be used to specify the indices of the categorical predictors for target encoding.

```
model = CatBoostRegressor(verbose = False, cat_features = range(4, 8)).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

```
3263.1348853593345
```

Let us compare the resuts with one-hot encoding of the categorical predictors.

```
X = train[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']
Xtest = test[['mileage','mpg','year','engineSize', 'brand', 'model',  'transmission', 'fuelTy
X = pd.get_dummies(X)
Xtest = pd.get_dummies(Xtest)
```

In one-hot encoding, we need to make sure that both the datasets have the same predictors. Let us find the predictors in train data that are not in test data. Note that this is not necessary in target encoding.

```
np.setdiff1d(X.columns, Xtest.columns)
```

```
array(['model_ M6'], dtype=object)
```

```
X.drop(columns = 'model_ M6', inplace = True)
y = train['price']
ytest = test['price']
```

```
model = CatBoostRegressor(verbose = False).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

```
3219.857899121199
```

In this case, target encoding has a slightly higher RMSE as compared to one-hot encoding. However, it may do better than one-hot-encoding in a different problem.

Let us use both target encoding and one-hot encoding together to see if it helps do better than each of the them individually.

```
X = pd.concat([train[['brand', 'model', 'transmission', 'fuelType']], X], axis = 1)
Xtest = pd.concat([test[['brand', 'model', 'transmission', 'fuelType']], Xtest], axis = 1)
```

```
model = CatBoostRegressor(verbose = False, cat_features = range(4)).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

```
3172.449374536484
```

In this case, using target-encoding and one-hot-encoding together does better on test data. Using both the encodings together will help reduce bias while increasing variance. The benefit of using both the encodings together depends on the bias-variance tradeoff.

12.2.3 CatBoost vs XGBoost

Let us see the performance of XGBoost with default hyperparameter settings.

```
model_xgb = xgb.XGBRFRegressor().fit(X, y)
np.mean(-cross_val_score(xgb.XGBRFRegressor(), X, y, cv = cv, n_jobs = -1,
                scoring='neg_root_mean_squared_error'))
```

```
6273.043859096154
```

```
np.sqrt(mean_squared_error(model_xgb.predict(Xtest),ytest))
```

```
6821.745153860935
```

XGBoost performance deteriorates showing that hyperparameter tuning is more important in XGBoost.

Let us see the performance of LightGBM with default hyperparameter settings.

```
model_lgbm = LGBMRegressor().fit(X, y)
np.mean(-cross_val_score(LGBMRegressor(), X, y, cv = cv, n_jobs = -1,
                scoring='neg_root_mean_squared_error'))
```

```
5562.149251902867
```

```
np.sqrt(mean_squared_error(model_lgbm.predict(Xtest),ytest))
```

```
5494.0777923513515
```

LightGBM's default hyperparameter settings also seem to be more robust as compared to those of XGBoost.

12.2.4 Tuning `CatBoostRegressor`

The CatBoost hyperparameters can be tuned just like the XGBoost hyperparameters. However, there is some difference in the hyperparameters of both the packages. For example, `reg_alpha` *(the L1 penalization on weights of leaves)* and `colsample_bytree` *(subsample ratio of columns when constructing each tree)* hyperparameters are not there in CatBoost.

```
#K-fold cross validation to find optimal parameters for CatBoost regressor
start_time = time.time()
param_grid = {'max_depth': [4,6,8, 10],
              'num_leaves': [20, 31, 40, 60],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 10, 100],
                'n_estimators':[500, 1000, 1500],
                'subsample': [0.5, 0.75, 1.0],
             'colsample_bylevel': [0.25, 0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=CatBoostRegressor(random_state=1, verbose=Fals
                            grow_policy='Lossguide'),
                             param_distributions = param_grid, n_iter = 200,
                             verbose = 1,random_state = 1, scoring='neg_root_mean_squared_er
                             n_jobs=-1,
                             cv = cv)
optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Optimal parameter values = {'subsample': 0.5, 'reg_lambda': 0, 'num_leaves': 40, 'n_estimato
Optimal cross validation RMSE =  -4993.129407810791
Time taken =  23  minutes
```

```
#RMSE based on the optimal parameter values
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

```
5249.434282204398
```

It takes 2 minutes to tune CatBoost, which is higher than LightGBM and lesser than XGBoost. CatBoost falls in between LightGBM and XGBoost in terms of speed. However, it is likely to be more accurate than XGBoost and LighGBM, and likely to require lesser tuning as compared to XGBoost.

```
model = CatBoostRegressor(grow_policy='Lossguide')

grid = {'num_leaves': Integer(4, 64),
              'learning_rate': Real(0.0001, 1.0),
               'reg_lambda':Real(0, 1e4),
                'n_estimators':Integer(2, 2000),
                'subsample': Real(0.1,1.0),
                'colsample_bylevel': Real(0.1, 1.0)}

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 200, random_state = 1,
                         scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
<IPython.core.display.Image object>
```

```
# Optimal values obtained
#['colsample_bylevel', 'learning_rate', 'n_estimators', 'num_leaves', 'reg_lambda', 'subsampl
#[0.3745508446405472, 0.1000958551500621, 2000, 11, 0.0, 0.3877212027881348] 5132.5378396768
```

12.2.5 Tuning Tips

Check the documentation for some tuning tips.

1. It is not recommended to use values greater than 64 for `num_leaves`, since it can significantly slow down the training process.
2. In most cases, the optimal depth ranges from 4 to 10. Values in the range from 6 to 10 are recommended. The maximum possible value of `max_depth` is 16.
3. Do not use one-hot encoding during preprocessing. This affects both the training speed and the resulting quality.
4. Symmetric trees have a very good prediction speed (roughly 10 times faster than non-symmetric trees) and give better quality in many cases.

13 Ensemble modeling

Ensembling models can help reduce error by leveraging the diversity and collective wisdom of multiple models. When ensembling, several individual models are trained independently and their predictions are combined to make the final prediction.

We have already seen examples of ensemble models in chapters 5 - 13. The ensembled models may reduce error by reducing the bias *(boosting)* and / or reducing the variance *(bagging / random forests / boosting)*.

However, in this chapter we'll ensemble different types of models, instead of the same type of model. We may ensemble a linear regression model, a random forest, a gradient boosting model, and as many different types of models as we wish.

Below are a couple of reasons why ensembling models can be effective in reducing error:

1. **Bias reduction:** Different models may have different biases and the ensemble can help mitigate the individual biases, leading to a more generalized and accurate prediction. For example, consider that one model has a positive bias, and another model has a negative bias for the same instance. By averaging or combining the predictions of the two models, the biases may cancel out.
2. **Variance reduction:** As seen in the case of random forests and bagged trees, by averaging or combining the predictions of multiple models, the ensemble can reduce the overall variance and improve the accuracy of the final prediction. Note that for variance reduction, the models should have a low correlation *(recall the variance reduction formula of random forests)*.

Mathematically also, we can show the effectiveness of an ensemble model. Let's consider the case of regression, and let the predictors be denoted as X, and the response as Y. Let $f_1, ..., f_m$ be individual models. The expected MSE of an ensemble can be written as:

$$E(MSE_{Ensemble}) = E\Bigg[\bigg(\frac{1}{m}\sum_{i=1}^{m} f_i(X) - Y\bigg)^2\Bigg] = \frac{1}{m^2}\sum_{i=1}^{m} E\Big[\big(f_i(X) - Y\big)^2\Big] + \frac{1}{m^2}\sum_{i \neq j} E\Big[\big(f_i(X) - Y\big)\big(f_j(X) - Y\big)\Big]$$

$$\implies E(MSE_{Ensemble}) = \frac{1}{m}\bigg(\frac{1}{m}\sum_{i=1}^{m} E\Big[\big(f_i(X) - Y\big)^2\Big]\bigg) + \frac{1}{m^2}\sum_{i \neq j} E\Big[\big(f_i(X) - Y\big)\big(f_j(X) - Y\big)\Big]$$

$$\implies E(MSE_{Ensemble}) = \frac{1}{m}\left(\frac{1}{m}\sum_{i=1}^{m} E(MSE_{f_i})\right) + \frac{1}{m^2}\sum_{i \neq j} E\Big[(f_i(X) - Y)(f_j(X) - Y)\Big]$$

If $f_1, ..., f_m$ are unbiased, then,

$$E(MSE_{Ensemble}) = \frac{1}{m}\left(\frac{1}{m}\sum_{i=1}^{m} E(MSE_{f_i})\right) + \frac{1}{m^2}\sum_{i \neq j} Cov(f_i(X), f_j(X))$$

Assuming the **models are uncorrelated** *(i.e., they have a zero correlation)*, the second term *(covariance of $f_i(.)$ and $f_j(.)$)* reduces to zero, and the expected MSE of the ensemble reduces to:

$$E(MSE_{Ensemble}) = \frac{1}{m}\left(\frac{1}{m}\sum_{i=1}^{m} E(MSE_{f_i})\right) \tag{13.1}$$

Thus, the expected MSE of an ensemble model with uncorrelated models is much smaller than the average MSE of all the models. Unless there is a model that is much better than the rest of the models, the MSE of the ensemble model is likely to be lower than the MSE of the individual models. However, there is no guarantee that the MSE of the ensemble model will be lower than the MSE of the individual models. Consider an extreme case where only one of the models have a zero MSE. The MSE of this model will be lower than the expected MSE of the ensemble model.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split, GridSearchCV, Paramete
StratifiedKFold, RandomizedSearchCV
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_curve
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, \
StackingClassifier, GradientBoostingRegressor,GradientBoostingClassifier, BaggingRegressor, 
BaggingClassifier,RandomForestRegressor,RandomForestClassifier,AdaBoostRegressor,AdaBoostCla
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, Elas
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

13.1 Ensembling regression models

13.1.1 Voting Regressor

Here, we will combine the predictions of different models. The function `VotingRegressor()` averages the predictions of all the models.

Below are the individual models tuned in the previous chapters.

```
#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),n_estimators=50,
                    learning_rate=1.0,  random_state=1).fit(X, y)
print("RMSE for AdaBoost = ", np.sqrt(mean_squared_error(model_ada.predict(Xtest), ytest)))

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                        n_jobs=-1, max_features=2).fit(X, y)
print("RMSE for Random forest = ", np.sqrt(mean_squared_error(model_rf.predict(Xtest), ytest

# Tuned XGBoost model from Section 9.2.6
model_xgb = xgb.XGBRegressor(random_state=1,max_depth=8,n_estimators=1000, subsample = 0.75,
                                         learning_rate = 0.01,reg_lambda=1, gamma = 100).fit
print("RMSE for XGBoost = ", np.sqrt(mean_squared_error(model_xgb.predict(Xtest), ytest)))

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                         random_state=1,loss='huber').fit(X, y)
print("RMSE for Gradient Boosting = ", np.sqrt(mean_squared_error(model_gb.predict(Xtest), y

# Tuned Light GBM model from Section 13.1.1
model_lgbm = LGBMRegressor(subsample = 0.5, reg_lambda = 0, reg_alpha = 100, boosting_type =
            num_leaves = 31, n_estimators = 500, learning_rate = 0.05, colsample_bytree = 1.
                          top_rate = 0.5).fit(X, y)
print("RMSE for LightGBM = ", np.sqrt(mean_squared_error(model_lgbm.predict(Xtest), ytest)))

# Tuned CatBoost model from Section 13.2.3
model_cat = CatBoostRegressor(subsample=0.5, num_leaves=40, n_estimators=500, max_depth=10,
                              verbose = False, learning_rate = 0.05, colsample_bylevel=0.75,
                              grow_policy='Lossguide', random_state = 1).fit(X, y)
print("RMSE for CatBoost = ", np.sqrt(mean_squared_error(model_cat.predict(Xtest), ytest)))
```

```
RMSE for AdaBoost =  5693.165811600585
RMSE for Random forest =  5642.45839697972
RMSE for XGBoost =  5497.553788113875
RMSE for Gradient Boosting =  5405.787029062213
RMSE for LightGBM =  5355.964600884197
RMSE for CatBoost =  5271.104736146779
```

Note that we **don't need to fit** the models **individually** before fitting them simultaneously in the voting ensemble. If we fit them individual, it will unnecessarily **waste time**.

Let us ensemble the models using the voting ensemble with equal weights.

```
#Voting ensemble: Averaging the predictions of all models

#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),
                    n_estimators=50,learning_rate=1.0,  random_state=1)

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                        n_jobs=-1, max_features=2)

# Tuned XGBoost model from Section 9.2.6
model_xgb = xgb.XGBRegressor(random_state=1,max_depth=8,n_estimators=1000, subsample = 0.75,
                colsample_bytree = 1.0, learning_rate = 0.01,reg_lambda=1, gamma = 100)

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                         random_state=1,loss='huber')

# Tuned CatBoost model from Section 13.2.3
model_cat = CatBoostRegressor(subsample=0.5, num_leaves=40, n_estimators=500, max_depth=10,
                             learning_rate = 0.05, colsample_bylevel=0.75, grow_policy='Lossg
                             random_state=1, verbose = False)

# Tuned Light GBM model from Section 13.1.1
model_lgbm = LGBMRegressor(subsample = 0.5, reg_lambda = 0, reg_alpha = 100, boosting_type =
                           num_leaves = 31, n_estimators = 500, learning_rate = 0.05,
                           colsample_bytree = 1.0, top_rate = 0.5)

start_time = time.time()
en = VotingRegressor(estimators = [('xgb',model_xgb),('ada',model_ada),('rf',model_rf),
                    ('gb',model_gb), ('cat', model_cat), ('lgbm', model_lgbm)], n_jobs = -1)
en.fit(X,y)
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Ensemble model RMSE =  5259.899392611916
Time taken =  0.21 minutes
```

As expected, RMSE of the ensembled model is less than that of each of the individual models.

Note that the RMSE can be **further improved** by **removing** the **weaker models** from the ensemble. Let us remove the three weakest models - XGBoost, Random forest, and AdaBoost.

```
#Voting ensemble: Averaging the predictions of all models

start_time = time.time()
en = VotingRegressor(estimators = [('gb',model_gb), ('cat', model_cat), ('lgbm', model_lgbm)
en.fit(X,y)
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Ensemble model RMSE =  5191.814866810768
Time taken =  0.18 minutes
```

13.1.2 Stacking Regressor

Stacking is a more sophisticated method of ensembling models. The method is as follows:

1. The training data is split into K folds. Each of the K folds serves as a test data in one of the K iterations, and the rest of the folds serve as train data.
2. Each model is used to make predictions on each of the K folds, after being trained on the remaining *K-1* folds. In this manner, each model predicts the response on each train data point - when that train data point was not used to train the model.
3. Predictions at each training data points are generated by each model in step 2 (the above step). These predictions are now used as predictors to train a meta-model (referred by the argument `final_estimator`), with the original response as the response. The meta-model (or `final_estimator`) learns to combine predictions of different models to make a better prediction.

13.1.2.1 Metamodel: Linear regression

```
#Stacking using LinearRegression as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_rf),
                                     ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgb
                     final_estimator=LinearRegression(),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
```

```
print("Linear regression metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),yt
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Linear regression metamodel RMSE =  5220.456280327686
Time taken =  2.03 minutes
```

```
#Co-efficients of the meta-model
en.final_estimator_.coef_
```

```
array([ 0.05502964,  0.14566665,  0.01093624,  0.30478283,  0.57403909,
       -0.07057344])
```

```
sum(en.final_estimator_.coef_)
```

```
1.0198810182715363
```

Note the above coefficients of the meta-model. The model gives the **highest weight** to the **gradient boosting** model *(with **huber** loss)*, and the **catboost** model, and the **lowest weight** to the relatively weak **random forest** model.

Also, note that the **coefficients need not sum to one.**

Let us try improving the RMSE further by removing the weaker models from the ensemble. Let us remove the three weakest models based on the size of their coefficients in the linear regression metamodel.

```
#Stacking using LinearRegression as the metamodel
en = StackingRegressor(estimators = [('gb', model_gb), ('cat', model_cat), ('ada', model_ada
                     final_estimator=LinearRegression(),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Linear regression metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),yt
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Linear regression metamodel RMSE =  5205.225710180056
Time taken =  1.36 minutes
```

The metamodel accuracy **improves further**, when **strong models** are ensembled.

```
#Co-efficients of the meta-model
en.final_estimator_.coef_
```

```
array([0.31824119, 0.54231032, 0.15998634])
```

```
sum(en.final_estimator_.coef_)
```

```
1.020537847948332
```

13.1.2.2 Metamodel: Lasso

```
#Stacking using Lasso as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_rf),
                        ('gb', model_gb),('cat', model_cat), ('lgbm', model_lgbm) ],
                     final_estimator = LassoCV(),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Lasso metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Lasso metamodel RMSE =  5206.021083501416
Time taken =  2.05 minutes
```

```
#Coefficients of the lasso metamodel
en.final_estimator_.coef_
```

```
array([ 0.03524446,  0.15077605,  0.        ,  0.30392268,  0.52946243,
       -0.        ])
```

Note that lasso **reduces the weight** of the **weak random forest** model, and **light gbm** model to **0**. Even though light GBM is a strong model, it may be **correlated or collinear** with XGBoost, or other models, and hence is not needed.

Note that as lasso performs **model selection** on its own, removing models with zero coefficients or weights does not make a difference, as shown below.

```
#Stacking using Lasso as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),
                        ('gb', model_gb),('cat', model_cat) ],
                     final_estimator = LassoCV(),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Lasso metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Lasso metamodel RMSE =  5205.93233977352
Time taken =  1.79 minutes
```

```
#Coefficients of the lasso metamodel
en.final_estimator_.coef_
```

```
array([0.03415944, 0.15053122, 0.30464838, 0.53006297])
```

13.1.2.3 Metamodel: Random forest

A highly flexible model such as a random forest may not be a good choice for ensembling correlated models. However, let us tune the random forest meta model, and check its accuracy.

```
# Tuning hyperparameter of the random forest meta-model
start_time = time.time()
oob_score_i = []
for i in range(1, 7):
    en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_
                        ('gb', model_gb),('cat', model_cat), ('lgbm', model_lgbm)],
                     final_estimator = RandomForestRegressor(max_features = i, oob_score = T
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1)).fit(X,y)
    oob_score_i.append(en.final_estimator_.oob_score_)
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Time taken =  12.08 minutes
```

```
print("Optimal value of max_features =", np.array(oob_score_i).argmax() + 1)
```

```
Optimal value of max_features = 1
```

```
# Training the tuned random forest metamodel
start_time = time.time()
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),
                    ('rf', model_rf), ('gb', model_gb),('cat', model_cat),
                    ('lgbm', model_lgbm)],
                final_estimator = RandomForestRegressor(max_features = 1,
                n_estimators=500), cv = KFold(n_splits = 5, shuffle = True,
                random_state=1)).fit(X,y)
print("Random Forest metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Random Forest metamodel RMSE =  5441.9155087961
Time taken =  1.71 minutes
```

Note that highly flexible models may not be needed when the predictors are highly correlated with the response. However, in some cases, they may be useful, as in the classification example in the next section.

13.1.2.4 Metamodel: CatBoost

```
#Stacking using MARS as the meta-model
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_rf),
                        ('gb', model_gb),('cat', model_cat), ('lgbm', model_lgbm)],
                     final_estimator = CatBoostRegressor(verbose = False),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Random Forest metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Random Forest metamodel RMSE =  5828.803609683251
Time taken =  1.66 minutes
```

13.2 Ensembling classification models

We'll ensemble models for predicting accuracy of identifying people having a heart disease.

```
data = pd.read_csv('./Datasets/Heart.csv')
data.dropna(inplace = True)
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy variables replacing the categorical variable
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)

#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)
```

Let us tune the individual models first.

AdaBoost

```
# Tuning Adaboost for maximizing accuracy
model = AdaBoostClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['base_estimator'] = [DecisionTreeClassifier(max_depth=1), DecisionTreeClassifier(max_dep
                          DecisionTreeClassifier(max_depth=3),DecisionTreeClassifier(max_dep
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accu
# execute the grid search
grid_result = grid_search.fit(Xtrain, ytrain)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.871494 using {'base_estimator': DecisionTreeClassifier(max_depth=1), 'learning_rate'
```

Gradient Boosting

```
# Tuning gradient boosting for maximizing accuracy
model = GradientBoostingClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['max_depth'] = [1,2,3,4,5]
grid['subsample'] = [0.5,1.0]
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accu
# execute the grid search
grid_result = grid_search.fit(Xtrain, ytrain)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.871954 using {'learning_rate': 1.0, 'max_depth': 4, 'n_estimators': 100, 'subsample'
```

XGBoost

```
# Tuning XGBoost for maximizing accuracy
start_time = time.time()
param_grid = {'n_estimators':[25, 100,250,500],
                'max_depth': [4, 6 ,8],
              'learning_rate': [0.01,0.1,0.2],
               'gamma': [0, 1, 10, 100],
               'reg_lambda':[0, 10, 100],
               'subsample': [0.5, 0.75, 1.0]
                'scale_pos_weight':[1.25,1.5,1.75]#Control the balance of positive and negat
             }

cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = GridSearchCV(estimator=xgb.XGBClassifier(random_state=1),
                             param_grid = param_grid,
                             scoring = 'accuracy',
                             verbose = 1,
                             n_jobs=-1,
                             cv = cv)
optimal_params.fit(Xtrain,ytrain)
```

```
print(optimal_params.best_params_,optimal_params.best_score_)
print("Time taken = ", (time.time()-start_time)/60, " minutes")
```

```
Fitting 5 folds for each of 972 candidates, totalling 4860 fits
{'gamma': 0, 'learning_rate': 0.2, 'max_depth': 4, 'n_estimators': 25, 'reg_lambda': 0, 'scal
Time taken =  0.9524135629336039  minutes
```

```
#Tuned Adaboost model
model_ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1), n_estimato
                               random_state=1,learning_rate=0.01).fit(Xtrain, ytrain)
test_accuracy_ada = model_ada.score(Xtest,ytest) #Returns the classification accuracy of the

#Tuned Random forest model from Section 6.3
model_rf = RandomForestClassifier(n_estimators=500, random_state=1,max_features=3,
                        n_jobs=-1,oob_score=False).fit(Xtrain, ytrain)
test_accuracy_rf = model_rf.score(Xtest,ytest) #Returns the classification accuracy of the mo

#Tuned gradient boosting model
model_gb = GradientBoostingClassifier(n_estimators=100, random_state=1,max_depth=4,learning_r
                                     subsample = 1.0).fit(Xtrain, ytrain)
test_accuracy_gb = model_gb.score(Xtest,ytest) #Returns the classification accuracy of the mo

#Tuned XGBoost model
model_xgb = xgb.XGBClassifier(random_state=1,gamma=0,learning_rate = 0.2,max_depth=4,
                              n_estimators = 25,reg_lambda = 0,scale_pos_weight=1.25).fit(Xtr
test_accuracy_xgb = model_xgb.score(Xtest,ytest) #Returns the classification accuracy of the

print("Adaboost accuracy = ",test_accuracy_ada)
print("Random forest accuracy = ",test_accuracy_rf)
print("Gradient boost accuracy = ",test_accuracy_gb)
print("XGBoost model accuracy = ",test_accuracy_xgb)
```

```
Adaboost accuracy =  0.7986577181208053
Random forest accuracy =  0.8120805369127517
Gradient boost accuracy =  0.7986577181208053
XGBoost model accuracy =  0.7785234899328859
```

13.2.1 Voting classifier - hard voting

In this type of ensembling, the predicted class is the one predicted by the majority of the classifiers.

```
ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_g
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

```
0.825503355704698
```

Note that the prediction accuracy of the ensemble is higher than the prediction accuracy of each of the individual models on unseen data.

13.2.2 Voting classifier - soft voting

In this type of ensembling, the predicted class is the one based on the average predicted probabilities of all the classifiers. The threshold probability is 0.5.

```
ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_g
                                 voting='soft')
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

```
0.7919463087248322
```

Note that soft voting will be good only for well calibrated classifiers, i.e., all the classifiers must have probabilities at the same scale.

13.2.3 Stacking classifier

Conceptually, the idea is similar to that of Stacking regressor.

```
#Using Logistic regression as the meta model (final_estimator)
ensemble_model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model
                                   final_estimator=LogisticRegression(random_state=1,max_ite
                                   cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

```
0.7986577181208053
```

```
#Coefficients of the logistic regression metamodel
ensemble_model.final_estimator_.coef_
```

```
array([[0.81748051, 1.28663164, 1.64593342, 1.50947087]])
```

```
#Using random forests as the meta model (final_estimator). Note that random forest will requi
ensemble_model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model
                                   final_estimator=RandomForestClassifier(n_estimators=500, m
                                                                          random_state=1,oob_
                                   cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

```
0.8322147651006712
```

Note that a complex `final_estimator` such as random forest will require tuning. In the above case, the `max_features` argument of random forests has been tuned to obtain the maximum OOB score. The tuning is shown below.

```
#Tuning the random forest parameters
start_time = time.time()
oob_score = {}

i=0
for pr in range(1,5):
    model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)
                                   final_estimator=RandomForestClassifier(n_estimators=500, m
                                    random_state=1,oob_score=True),n_jobs=-1,
                                   cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=
    oob_score[pr] = model.final_estimator_.oob_score_

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max accuracy = ", np.max(list(oob_score.values())))
print("Best value of max_features= ", np.argmax(list(oob_score.values()))+1)
```

```
time taken =  0.33713538646698  minutes
max accuracy =  0.8445945945945946
Best value of max_features=  1
```

```
#The final predictor (metamodel) - random forest obtains the maximum oob_score for max_featu
oob_score
```

```
{1: 0.8445945945945946,
 2: 0.831081081081081,
 3: 0.8378378378378378,
 4: 0.831081081081081}
```

13.2.4 Tuning all models simultaneously

Individual model hyperparameters can be tuned simultaneously while ensembling them with a `VotingClassifier()`. However, this approach can be too expensive for even moderately-sized datasets.

```
# Create the param grid with the names of the models as prefixes

model_ada = AdaBoostClassifier(base_estimator = DecisionTreeClassifier())
model_rf = RandomForestClassifier()
model_gb = GradientBoostingClassifier()
model_xgb = xgb.XGBClassifier()

ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_g

hp_grid = dict()

# XGBoost
hp_grid['xgb__n_estimators'] = [25, 100,250,50]
hp_grid['xgb__max_depth'] = [4, 6 ,8]
hp_grid['xgb__learning_rate'] = [0.01, 0.1, 1.0]
hp_grid['xgb__gamma'] = [0, 1, 10, 100]
hp_grid['xgb__reg_lambda'] = [0, 1, 10, 100]
hp_grid['xgb__subsample'] = [0, 1, 10, 100]
hp_grid['xgb__scale_pos_weight'] = [1.0, 1.25, 1.5]
hp_grid['xgb__colsample_bytree'] = [0.5, 0.75, 1.0]

# AdaBoost
hp_grid['ada__n_estimators'] = [10, 50, 100,200,500]
hp_grid['ada__base_estimator__max_depth'] = [1, 3, 5]
hp_grid['ada__learning_rate'] = [0.01, 0.1, 0.2]

# Random Forest
```

```
hp_grid['rf__n_estimators'] = [100]
hp_grid['rf__max_features'] = [3, 6, 9, 12, 15]

# GradBoost
hp_grid['gb__n_estimators'] = [10, 50, 100,200,500]
hp_grid['gb__max_depth'] = [1, 3, 5]
hp_grid['gb__learning_rate'] = [0.01, 0.1, 0.2, 1.0]
hp_grid['gb__subsample'] = [0.5, 0.75, 1.0]

start_time = time.time()
grid = RandomizedSearchCV(ensemble_model, hp_grid, cv=5, scoring='accuracy', verbose = True,
                         n_iter = 100, n_jobs=-1).fit(Xtrain, ytrain)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

```
grid.best_estimator_.score(Xtest, ytest)
```

```
0.8120805369127517
```

13.3 Ensembling models based on different sets of predictors

Generally, tree-based models such as CatBoost, and XGBoost are the most accurate, while other models, such as bagging, random forests, KNN, and linear models, may not be as accurate. Thus, sometimes, the weaker models, despite bringing-in diversity in the model ensemble may deteriorate the ensemble accuracy due to their poor individual performance *(check slides for technical details)*. Thus, sometimes, another approach to bring-in model diversity is to develop strong models based on different sets of predictors, and ensemble them.

Different feature selection methods *(such as Lasso, feature importance returned by tree-based methods, stepwise k-fold feature selection, etc.)*, may be used to obtain different sets of important features, strong models can be tuned on these sets, and then ensembled. Even though the models may be of the same type, the different sets of predictors will help bring-in the element of diversity in the ensemble.

```
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

We will create polynomial interactions to develop two sets of predictors - first order predictors, and second order predictors.

```
poly_set = PolynomialFeatures(2, include_bias = False)
X_poly = poly_set.fit_transform(X)
X_poly = pd.DataFrame(X_poly, columns=poly_set.get_feature_names_out())
X_poly.columns = X_poly.columns.str.replace("^", "_", regex=True)
Xtest_poly = poly_set.fit_transform(Xtest)
Xtest_poly = pd.DataFrame(Xtest_poly, columns=poly_set.get_feature_names_out())
Xtest_poly.columns = Xtest_poly.columns.str.replace("^", "_", regex=True)
```

Let us use 2 different sets of predictors to introduce diversity in the ensemble.

```
col_set1 = ['mileage','mpg', 'year','engineSize']
col_set2 = X_poly.columns
```

Let us use two types of strong tree-based models.

```
cat = CatBoostRegressor(verbose=False)
gb = GradientBoostingRegressor(loss = 'huber')
```

We will use the `Pipeline()` function along with `ColumnTransformer()` to map a predictor set to each model.

```
cat_pipe1 = Pipeline([
    ('column_transformer', ColumnTransformer([('cat1_transform', 'passthrough', col_set1)], r
    ('cat1', cat)
```

```
])

cat_pipe2 = Pipeline([
    ('column_transformer', ColumnTransformer([('cat2_transform', 'passthrough', col_set2)], r
    ('cat2', cat)
])

gb_pipe1 = Pipeline([
    ('column_transformer', ColumnTransformer([('gb1_transform', 'passthrough', col_set1)], re
    ('gb1', gb)
])

gb_pipe2 = Pipeline([
    ('column_transformer', ColumnTransformer([('gb2_transform', 'passthrough', col_set2)], re
    ('gb2', gb)
])
```

We will use Linear regression to ensemble the models.

```
en_new.final_estimator_.coef_
```

```
array([ 0.30127482,  0.79242981, -0.07168258, -0.01781781])
```

```
en_new = StackingRegressor(estimators = [('cat1', cat_pipe1),('cat2', cat_pipe2),
                                        ('gb1', gb_pipe1), ('gb2', gb_pipe2)],
                     final_estimator=LinearRegression(),
                    cv = KFold(n_splits = 15, shuffle = True, random_state=1))
```

```
en_new.fit(X_poly, y)
```

```
StackingRegressor(cv=KFold(n_splits=15, random_state=1, shuffle=True),
                  estimators=[('cat1',
                               Pipeline(steps=[('column_transformer',
                                                ColumnTransformer(transformers=[('cat1_trans
                                                                                 'passthroug
                                                                                 ['mileage',
                                                                                  'mpg',
                                                                                  'year',
                                                                                  'engineSiz
                                               ('cat1',
                                            <catboost.core.CatBoostRegressor object at 0x
                              ('cat2',
                               Pipeline(steps=[('column_transformer',...
                               Pipeline(steps=[('column_transformer',
                                            ColumnTransformer(transformers=[('gb2_transfo
                                                                           'passthrough
                                                                           Index(['mile
     'mileage year', 'mileage engineSize', 'mpg_2', 'mpg year',
     'mpg engineSize', 'year_2', 'year engineSize', 'engineSize_2'],
    dtype='object'))])),
                                           ('gb2',
                                            GradientBoostingRegressor(loss='huber'))]))]
            final_estimator=LinearRegression())
```

```
mean_squared_error(en_new.predict(Xtest_poly), ytest, squared = False)
```

```
5185.376722607323
```

Note that the above model does better on test data than all the models developed so far. Using different sets of predictors introduces diversity in the ensemble, as an alternative to including "weaker" models in the ensemble to add diversity.

Check the idea being used in the Spring 2023 prediction problem in the appendix.

A Assignment 1

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
3. Use Quarto to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. The assignment is worth 100 points, and is due on **Thursday, 11th April 2025 at 11:59 pm**.
5. **Five points are properly formatting the assignment**. The breakdown is as follows:
   - Must be an HTML file rendered using Quarto **(2 points)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
   - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
   - Final answers to each question are written in the Markdown cells. **(1 point)**
   - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**

A.1 1) Bias-Variance Trade-off for Regression (50 points)

The main goal of this question is to understand and visualize the bias-variance trade-off in a regression model by performing repetitive simulations.

The conceptual clarity about bias and variance will help with the main logic behind creating many models that will come up later in the course.

A.1.1 a) Define the True Relationship (Signal)

First, you need to implement the underlying true relationship (Signal) you want to sample data from. Assume that the function is the Bukin function. Implement it as a user-defined function and run it with the test cases below to make sure it is implemented correctly. **(5 points)**

Note: It would be more useful to have only one input to the function. You can treat the input as an array of two elements.

```
print(Bukin(np.array([1,2]))) # The output should be 141.177
print(Bukin(np.array([6,-4]))) # The output should be 208.966
print(Bukin(np.array([0,1]))) # The output should be 100.1
```

A.1.2 b) Generate Test Set (No Noise)

Generate a **noiseless** test set with **100 observations** sampled from the true underlying function. This test set will be used to evaluate **bias and variance**, so make sure it follows the correct data generation process.

(5 points)

Instructions:

- **Do not use loops** for this question.
- `.apply` will be especially helpful (and often simpler).

Data generation assumptions:

- Use `np.random.seed(100)` for reproducibility.
- The first predictor, x_1, should be drawn from a **uniform distribution** over the interval $[-15, -5]$, i.e., $x_1 \sim U[-15, -5]$.
- The second predictor, x_2, should be drawn from a **uniform distribution** over the interval $[-3, 3]$, i.e., $x_2 \sim U[-3, 3]$.
- Compute the true function values using the underlying model as your response y

A.1.3 c) Initialize Results DataFrame

Create an empty DataFrame with the following columns:

- **degree**: the degree of the polynomial model

- **bias_sq**: estimated squared bias (averaged over test points)

- **var**: estimated variance of predictions

- **bias_var_sum**: sum of bias squared and variance

- **empirical_mse**: mean squared error calculated using sklearn's `mean_squared_error()` on model predictions vs. true function values

This DataFrame will be used to store the results of your bias–variance tradeoff analysis and for generating comparison plots.

(3 points)

A.1.4 d) Generate Training Sets (With Noise)

To estimate the **bias**, **variance**, and **total error (MSE)** of a Linear Regression model trained on noisy data from the underlying Bukin function, follow the steps below.

Step 1: Generate 100 Training Sets

- Create **100 independent training datasets**, each with **100 observations** (same size as the test set).
- For each training dataset:
  - Use `np.random.seed(i)` to ensure reproducibility, where `i` is the dataset index (0 to 99).
  - Sample predictors from the **same distributions** used to generate the test set.
  - Add **Gaussian noise** with mean 0 and standard deviation 10:
    $\varepsilon \sim \mathcal{N}(0, 10)$

Step 2: Train Polynomial Models (Degrees 1 to 7)

- For each training dataset, train polynomial models with degrees **1 through 7**.
- Use polynomial feature transformations that include both:
  - **Higher-order terms** (e.g., x_1^2, x_2^3)
  - **Interaction terms** (e.g., $x_1 \cdot x_2$)
- Make predictions on the **fixed, noiseless test set** for each trained model.

Step 3: Estimate Bias², Variance, and MSE

- For each **degree**, and each **test point**, collect the 100 predicted values from the models trained on the different training sets.
- Using these predictions, compute:
  - **Bias squared**: squared difference between the mean prediction and the true value.
  - **Variance**: variance of the predictions.
  - **Theoretical MSE**: sum of bias squared and variance.
  - **Empirical MSE**: compute using `sklearn.metrics.mean_squared_error` between each model's prediction and the true values, then average over the 100 training runs.
- Store all four quantities for each degree in your results DataFrame:
  - `degree`
  - `bias_sq`
  - `var`
  - `bias_var_sum` (bias squared + variance)
  - `empirical_mse`

(25 points)

Reminder: Comparing Theoretical vs. Empirical MSE

When evaluating model performance on the **noiseless test set**:

- The **irreducible error** (i.e., noise in training data) does **not** affect the test targets.
- Therefore, the test error (MSE) can be decomposed as:

  $MSE = Bias^2 + Variance$
- The **empirical MSE** (from sklearn) should closely match the **sum of bias² and variance**, since the test data contains no noise.

A.1.5 e) Visualize Bias–Variance Decomposition

Using the results stored in your DataFrame, create a plot with **four lines**, each plotted against the polynomial **degree**:

1. **Bias squared**
2. **Variance**
3. **Bias squared + Variance** (i.e., the theoretical decomposition of MSE)
4. **Empirical MSE** calculated using `sklearn.metrics.mean_squared_error()` (computed from the predicted values vs. true function values on the noiseless test set)

Plot requirements: - Use a single line plot with the polynomial degree on the x-axis and error values on the y-axis. - Include a **legend** to clearly label each line. - Use different line styles or markers for easy visual comparison.

Goal: - Compare the **empirical MSE** to the **sum of bias squared and variance**. - If everything is implemented correctly, the two lines should be very close (or even identical, up to numerical precision).

A.1.6 f) Identify the Optimal Model

- What is the **optimal polynomial degree** based on the **lowest empirical MSE** (calculated using sklearn)?
  (2 points)
- Report the corresponding values of:
  - **Bias squared**
  - **Variance**
  - **Bias squared + Variance**
  - **Empirical MSE**
    for that degree.
    (3 points)

A.2 2) Building a Low-Bias, Low-Variance Model via Regularization (50 points)

The main goal of this question is to further reduce the **total prediction error** by applying **regularization**.
Specifically, you'll use **Ridge regression** to build a **low-bias, low-variance** model for data generated from the underlying Bukin function with noise.

A.2.1 a) Why Regularization?

Explain why the model with the optimal polynomial degree (as identified in Question 1) is **not guaranteed** to be the true low-bias, low-variance model.

Why might **regularization** still be necessary to improve generalization performance, even after selecting the degree that minimizes MSE?

(5 points)

A.2.2 b) Which Degrees to Exclude?

Based on your plot and results from **1e** and **1f**, identify which polynomial degrees should be **excluded** from regularization experiments because they are already too simple (high bias) or too complex (high variance).

Explain which degrees you will exclude and **why**, using your understanding of how **regularization affects bias and variance**.

(10 points)

A.2.3 c) Apply Ridge Regularization

Repeat the steps from **1c** and **1d**, but this time use **Ridge regression** instead of ordinary least squares.

- Use only the degrees **not excluded** in 2b (and also exclude degree 7 to avoid extreme overfitting).
- Use **5-fold cross-validation** to tune the Ridge regularization strength.
- Use `neg_root_mean_squared_error` as the scoring metric for cross-validation.
- Tune over a range of regularization strengths (e.g., from 1 to 100).
- For each retained degree, compute:
  - **Bias squared**
  - **Variance**
  - **Bias squared + Variance**
  - **Empirical MSE** (from `sklearn.metrics.mean_squared_error`)

Store your results in a new DataFrame with the same structure as in Question 1.

(10 points)

A.2.4 d) Visualize Regularized Results

Repeat the visualization from **1e**, but using the results from **2c** (Ridge regression).

Your plot should include **four lines** plotted against polynomial degree:

1. **Bias squared**
2. **Variance**
3. **Bias squared + Variance**
4. **Empirical MSE** (computed using sklearn)

Include a clear **legend** and label your axes.
This will help you visually assess how regularization impacts bias, variance, and overall model error.

(10 points)

A.2.5 e) Evaluate the Regularized Model

- What is the **optimal polynomial degree** for the Ridge Regression model, based on the **lowest empirical MSE**?
  (3 points)
- Report the corresponding values of:
  - **Bias squared**
  - **Variance**
  - **Empirical MSE**
    for that optimal Ridge model.
    (3 points)
- Compare these results to those of the optimal **Linear Regression** model from Question 1.
  Discuss how **regularization** influenced the **bias**, **variance**, and **overall prediction error (MSE)**.
  (4 points)

A.2.6 f) Interpreting the Impact of Regularization

- Was **regularization successful** in reducing the **total prediction error (MSE)** compared to the unregularized model?
  (2 points)
- Based on your results from **2e**, explain how **bias** and **variance** changed as a result of regularization.
  How did these changes affect the final total error?
  Support your explanation with values or observations from your analysis.
  (3 points)

B Assignment 2

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
3. Use Quarto to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. The assignment is worth 100 points, and is due on **Monday, 18th April 2025 at 11:59 pm**.
5. **Five points are properly formatting the assignment**. The breakdown is as follows:
   - Must be an HTML file rendered using Quarto **(1 point)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
   - No name can be written on the assignment, nor can there be any indicator of the student's identity—e.g. printouts of the working directory should not be included in the final submission. **(1 point)**
   - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
   - Final answers to each question are written in the Markdown cells. **(1 point)**
   - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**
6. The maximum possible score in the assigment is 103+5 = 108 out of 100.

B.1 Optimizing KNN for Classification (71 points)

In this question, you will use **classification_data.csv**. Each row is a loan and the each column represents some financial information as follows:

- `hi_int_prncp_pd`: Indicates if a high percentage of the repayments went to interest rather than principal. **This is the classification response.**
- `out_prncp_inv`: Remaining outstanding principal for portion of total amount funded by investors
- `loan_amnt`: The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
- `int_rate`: Interest Rate on the loan
- `term`: The number of payments on the loan. Values are in months and can be either 36 or 60.

As indicated above, `hi_int_prncp_pd` is the response and all the remaining columns are predictors. You will tune and train a K-Nearest Neighbors (KNN) classifier throughout this question.

B.1.1 a) Load the Dataset (1 point)

Read the dataset into your notebook.

B.1.2 b) Define Predictor and Response Variables (1 point)

Create the **predictor** (features) and **response** (target) variables from the dataset.

B.1.3 c) Split the Data into Training and Test Sets (1 points)

Create the training and test datasets using the following specifications:

- Use a **75%-25% split**.
- Ensure that the **class ratio is preserved** in both training and test sets (i.e., stratify the split).
- Set `random_state=45` for reproducibility.

B.1.4 d) Check Class Ratios (2 points)

Print the **class distribution** (ratios) for:

- The entire dataset
- The training set
- The test set

This is to verify that the **class ratio is preserved** after splitting.

B.1.5 e) Scale the Dataset (2 points)

Use `StandardScaler` to scale the dataset in order to prepare it for KNN modeling.

Scaling ensures that all features contribute equally to the distance calculations used by the KNN algorithm.

B.1.6 f) Set Up Cross-Validation (2 points)

Before creating and tuning your model, you need to define cross-validation settings to ensure consistent and accurate evaluation across folds.

Please follow these specifications:

- Use **5 stratified folds** to preserve class distributions in each split.
- **Shuffle** the data before splitting to introduce randomness.
- Set `random_state=14` for reproducibility.

Note: You must use these exact cross-validation settings throughout the rest of this question to maintain consistency.

B.1.7 g) Tune K for KNN Using Cross-Validation (12 points)

Tune a **KNN Classifier** using cross-validation with the following specifications:

- Use **every odd K value from 1 to 50** (inclusive).
- Keep all other model settings at their defaults.
- Use the **cross-validation settings** defined in part (f).
- Evaluate performance using the **F1 score** as the metric.

(4 points)

Then, complete the following tasks:

- Create a **plot of K values vs. cross-validation F1 scores** to visualize how K balances overfitting and underfitting. **(2 points)**
- Print the **best average cross-validation F1 score**. **(1 points)**
- Report the **K value corresponding to the best F1 score**. **(1 points)**
- Determine whether this is the **only K value** that results in the best F1 score. Use code to justify your answer. **(2 points)**
- Reflect on whether **accuracy** is a good metric for tuning the model in this case. Explain your reasoning. **(2 points)**

Hint:

In addition to reporting the best `K` and best F1 score, you may also want to examine the full cross-validation results to check if other K values achieved the same F1 score.

B.1.8 h) Optimize the Classification Threshold (4 points)

Using the **optimal K value** you identified in part (g), optimize the classification **threshold** to maximize the cross-validation F1 score.

B.1.8.1 Specifications:

- Search across all possible threshold values using a **step size of 0.05**.
- Use the **cross-validation settings** defined in part (f).
- Evaluate performance using the **F1 score**, consistent with part (g).

B.1.8.2 Tasks:

- Visualize the **F1 score vs. different threshold values**. **(2 points)**
- Identify and report the **best threshold** that yields the highest F1 score. **(1 points)**
- Output the **best cross-validation F1 score**. **(1 points)**

B.1.9 i) Evaluate the Tuning Method (2 points)

Is the method we used in parts (g) and (h) **guaranteed** to find the best combination of **K** and **threshold**, i.e., to tune the classifier to its optimal values?
(1 point)

Justify your answer.
(1 point)

B.1.10 j) Evaluate Tuned Classifier on Test Set (3 points)

Using the **tuned KNN classifier** and the **optimal threshold** you identified, evaluate the model on the **test set**. Report the following metrics:

- F1 Score

- Accuracy

- Precision

- Recall

- AUC

B.1.11 k) Jointly Tune K and Threshold (6 points)

Now, tune **K** and the **classification threshold simultaneously**, rather than sequentially.

- Use the same settings from the previous parts (i.e., odd K values from 1 to 50, threshold step size of 0.05, F1 score as the metric, and the same cross-validation strategy).
- Identify the **best F1 score**, along with the **K value and threshold** that produce it.

B.1.12 l) Visualize Cross-Validation Results with a Heatmap (3 points)

Create a **heatmap** to visualize the cross-validation results in two dimensions.

- The **x-axis** should represent the **K values**.
- The **y-axis** should represent the **threshold values**.
- The color should represent the **F1 score**.

Note: This question only requires **one line of code**. You'll need to recall a **data visualization function** and a **data reshaping method** from 303-1.

B.1.13 m) Compare Joint vs. Sequential Tuning Results (4 points)

- How does the **best cross-validation F1 score** from part (k) compare to the scores from parts (g) and (h)? **(1 point)**
- Did the **optimal K value** and **threshold** change when tuning them jointly? **(1 point)**
- Explain **why or why not**. Consider how tuning the two parameters together might impact the result. **(2 points)**

B.1.14 n) Evaluate Final Tuned Model on Test Set (3 points)

Using the **tuned classifier and threshold** from part (k), evaluate the model on the **test set**. Report the following metrics:

- F1 Score

- Accuracy

- Precision

- Recall

- AUC

B.1.15 o) Compare Tuning Strategies and Computational Cost (3 points)

Compare the tuning approach used in parts **(g) & (h)** (separate tuning of K and threshold) with the approach in **part (k)** (joint tuning of K and threshold) in terms of **computational cost**.

- How many **K and threshold combinations** did you evaluate in each approach? **(2 points)**
- Based on this comparison and your answer from part (l), explain the **main trade-off** involved in model tuning (e.g., between computation and performance). **(2 points)**

B.1.16 p) Tune K Using Multiple Metrics (5 points)

`GridSearchCV` and `cross_val_score` are designed to optimize based on a **single metric**.
In this section, you'll practice tuning hyperparameters while evaluating **multiple metrics** simultaneously using `cross_validate`.

For this imbalanced classification task, instead of optimizing the F1 score directly, we'll focus on **precision** and **recall** together.

Keep in mind that the F1 score is the **harmonic mean** of precision and recall—it balances the trade-off between the two.

Cross-validate a **KNN classifier** using the following specifications:

- Use the **same cross-validation setting** and **hyperparameter grid** as before
- Evaluate the model using **precision**, **recall**, and **f1-score**, as metrics **at the same time**.

Save the cross-validation results into a **DataFrame**, and compute the **average score for each metric**, and visualize how these metrics change with different values of K.

B.1.17 q) Optimize for Recall with Precision Constraint (4 point)

Identify the **K value** that yields the **highest recall**, while maintaining a **precision of at least 75%**.
(3 points)

Then, print the **average cross-validation metrics** (f1-score, precision, recall) for that K value.
(1 point)

B.1.18 r) Tune Threshold for Maximum Recall (3 point)

Using the **optimal K value** identified in part (q), find the **threshold** that maximizes **cross-validation Recall**, following the specifications below:

- Evaluate all possible threshold values with a **step size of 0.05**.
- Use the **cross-validation settings** from part (f).

Then: - Print the **best cross-validation recall**. - Report the **threshold value** that achieves this recall.

Note: This task is very similar to part (h), but it's important for the next part.

B.1.19 s) Evaluate Precision-Optimized Model on Test Set (2 points)

Using the **tuned classifier and threshold** from parts (q) and (r), evaluate the model on the **test set**. Report the following metrics:

- F1 Score

- Accuracy

- Precision
- Recall
- AUC

B.1.20 t) Final Reflection: Comparing Tuning Strategies (3 points)

You have now tuned your KNN classifier using **three different strategies**:

1. **Sequential tuning** of K and threshold based on **F1 score** (parts g–h)
2. **Joint tuning** of K and threshold using **F1 score** (part k)
3. Tuning based on **multiple metrics**, selecting the K with the **highest recall** while maintaining **precision 75%** (parts p–r)

Reflect on the following:

- Which tuning strategy led to the **best overall performance on the test set**, based on the metrics you care about most?
- Which strategy would you choose in a real-world application, and why?
- What are the **trade-offs** between tuning for F1 score versus prioritizing precision or recall individually?

Note: This is an open-ended question. As long as your reasoning makes sense, you will receive full credit.

B.2 Tuning a KNN Regressor on Bank Loan Data (32 points)

In this question, you will use **bank_loan_train_data.csv** to tune *(the model hyperparameters)* and train the model. Each row is a loan and the each column represents some financial information as follows:

- `money_made_inv`: Indicates the amount of money made by the bank on the loan. **This is the regression response.**
- `out_prncp_inv`: Remaining outstanding principal for portion of total amount funded by investors
- `loan_amnt`: The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
- `int_rate`: Interest Rate on the loan

- `term`: The number of payments on the loan. Values are in months and can be either 36 or 60
- `mort_acc`: The number of mortgage accounts
- `application_type_Individual`: 1 if the loan is an individual application or a joint application with two co-borrowers
- `tot_cur_bal`: Total current balance of all accounts
- `pub_rec`: Number of derogatory public records

As indicated above, `money_made_inv` is the response and all the remaining columns are predictors. You will tune and train a K-Nearest Neighbors (KNN) regressor throughout this question.

B.2.1 a) Split, Scale, and Tune a KNN Regressor (15 point)

Create the **training and test datasets** using the following specifications:

- Use an **80%-20% split**.
- Set `random_state=1` for reproducibility.

Then, **scale all the predictors**, as KNN is sensitive to the scale of input features.

Next, you will **tune a KNN Regressor** by searching for the optimal hyperparameters using three search approaches: **Grid Search**, **Random Search**, and **Bayesian Search**.

B.2.1.1 Cross-Validation Setting

You should use **5-fold cross-validation**, with the following specifications:

- The data should be **shuffled** before splitting
- Use `random_state=1` to ensure **reproducibility**

B.2.1.2 Hyperparameters to Tune:

You will tune the following hyperparameters for the KNN Regressor:

You will tune the following hyperparameters for the **K-Nearest Neighbors Regressor**, using **Minkowski** as the distance metric:

1. `n_neighbors`: Number of nearest neighbors
   - Tune over the range: `np.arange(1, 25, 1)`
2. `p`: Power parameter for the Minkowski distance
   - Use values: `np.arange(1, 4, 1)`
   - `p = 1` corresponds to Manhattan distance
   - `p = 2` corresponds to Euclidean distance
   - **Note**: Set the distance `metric` to `"minkowski"`
3. `weights`: Weight function used in prediction
   You must consider the following **5 types of weights**:
   - `'uniform'`: All neighbors are weighted equally
   - `'distance'`: Weight is inversely proportional to distance
   - Custom weight functions:
     - $\propto \frac{1}{\text{distance}^2}$
     - $\propto \frac{1}{\text{distance}^3}$
     - $\propto \frac{1}{\text{distance}^4}$

For **each search method** (Grid Search, Random Search, Bayesian Search), report the following:

- `best_params_`: The best combination of hyperparameters
- `best_score_`: Cross-validated RMSE on the training set
- **Test RMSE** obtained from the best model
- **Execution time** for the search process

Hint:

Define **three custom weight functions** as shown below:

```
def dist_power_2(distance):
    return 1 / (1e-10 + distance**2)

def dist_power_3(distance):
    return 1 / (1e-10 + distance**3)

def dist_power_4(distance):
    return 1 / (1e-10 + distance**4)
```

Note the small constant `1e-10` helps avoid division by zero and numerical instability.

B.2.2 b) Compare Tuning Approaches (1 point)

Compare the results from part (2a) in terms of **execution time** and **model performance**. Briefly discuss the **main trade-offs** among the three hyperparameter tuning approaches: Grid Search, Random Search, and Bayesian Search.

B.2.3 c) Feature Selection and Hyperparameter Tuning with GridSearchCV (15 point)

KNN performance can **deteriorate significantly** if irrelevant or noisy predictors are included. In this part, you will explore **feature selection** to improve model performance, followed by **hyperparameter tuning** using `GridSearchCV` (with `refit=True`).

Try the following **three different feature selection approaches**:

1. **Correlation-based filtering**:
    - Select features with an absolute correlation of at least **0.1** with the target variable.
2. **Lasso regression for feature selection**:
    - Use `Lasso(alpha=50)` to select important features based on non-zero coefficients.
3. **SelectKBest**:
    - Use `SelectKBest` with `f_regression`, selecting the **top 4** features.

For **each approach**, perform hyperparameter tuning using **GridSearchCV**, and report:

- The **best score** (cross-validated RMSE) on the **training set**
- The **test RMSE** from the best model
- The **best hyperparameters**

B.2.4 d) Compare Feature Selection Approaches (1 point)

Create a **DataFrame** that summarizes the model performance from each feature selection method, including:

- **Training RMSE**
- **Test RMSE**

Be sure to also include the results from the model trained **without any feature selection** for comparison.

Then, briefly explain what you learned from this experiment.
For example: Did feature selection improve performance? Which method worked best?

C Datasets, assignment and project files

Datasets used in the book, assignment files, project files, and prediction problems report tempate can be found here