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 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
```

1.1 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
```

```
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)
```

```
# 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()
```

```
# 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
```

```
# 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'].transforme

# 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'].transfor
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']
```

1.2 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
```

1.3 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.

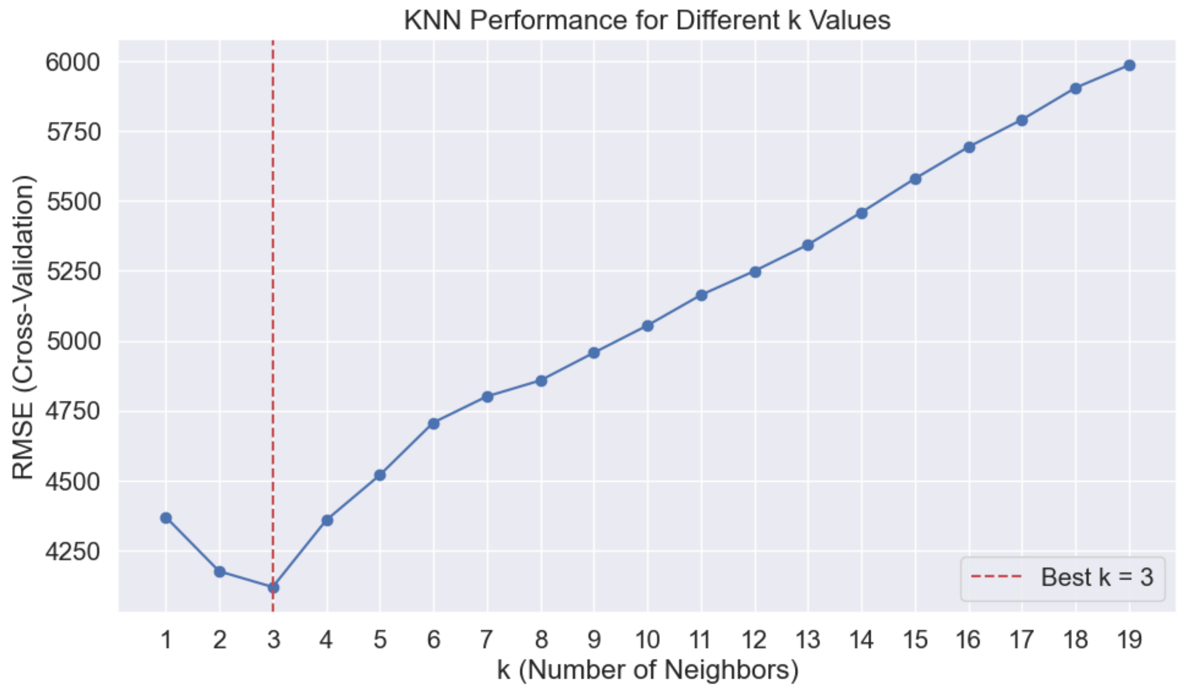
1.3.1 Tuning *k* in KNN

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

1.3.2 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'}
```

```
# 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
```

```python
# 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.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_knn__metric	param_knn
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.

```python
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_knn
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

1.4 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.

1.4.0.1 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
```

```python
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.

1.4.0.2 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
```

2 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.

```python
# 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)
```

2.1 Simple model (Less flexible)

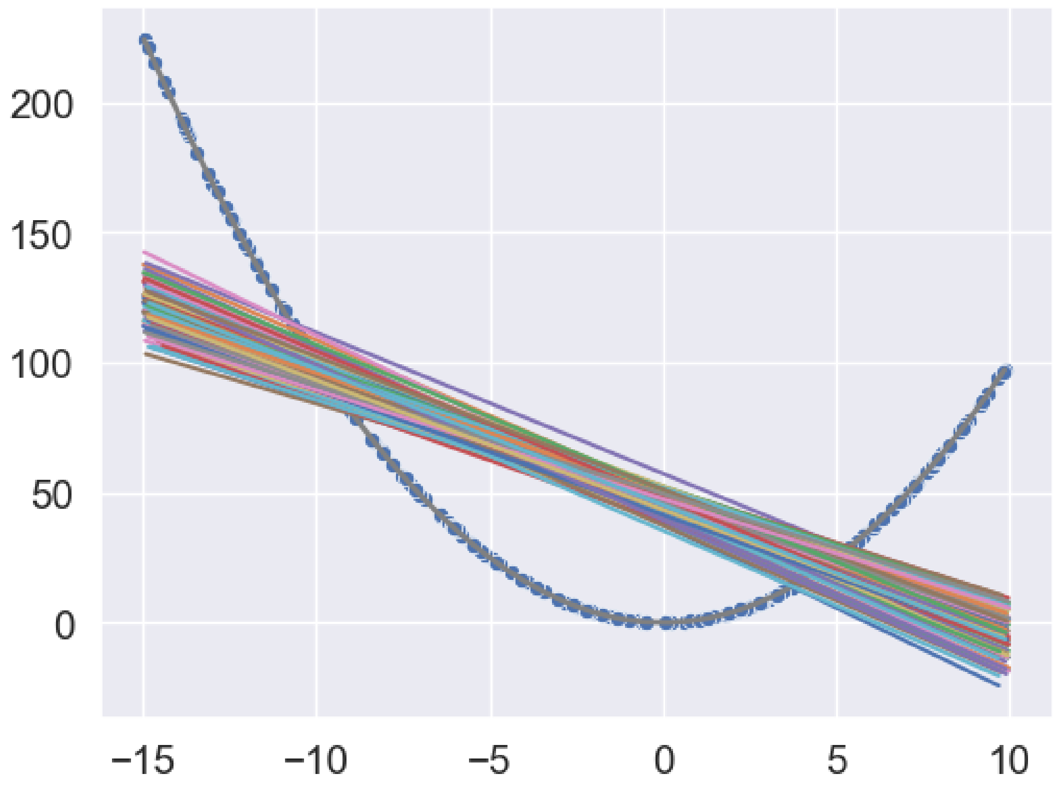
Let us consider a linear regression model as the less-flexible *(or relatively simple)* model.

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

```python
np.random.seed(101)

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

```python
# 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()
```

```python
# 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
```

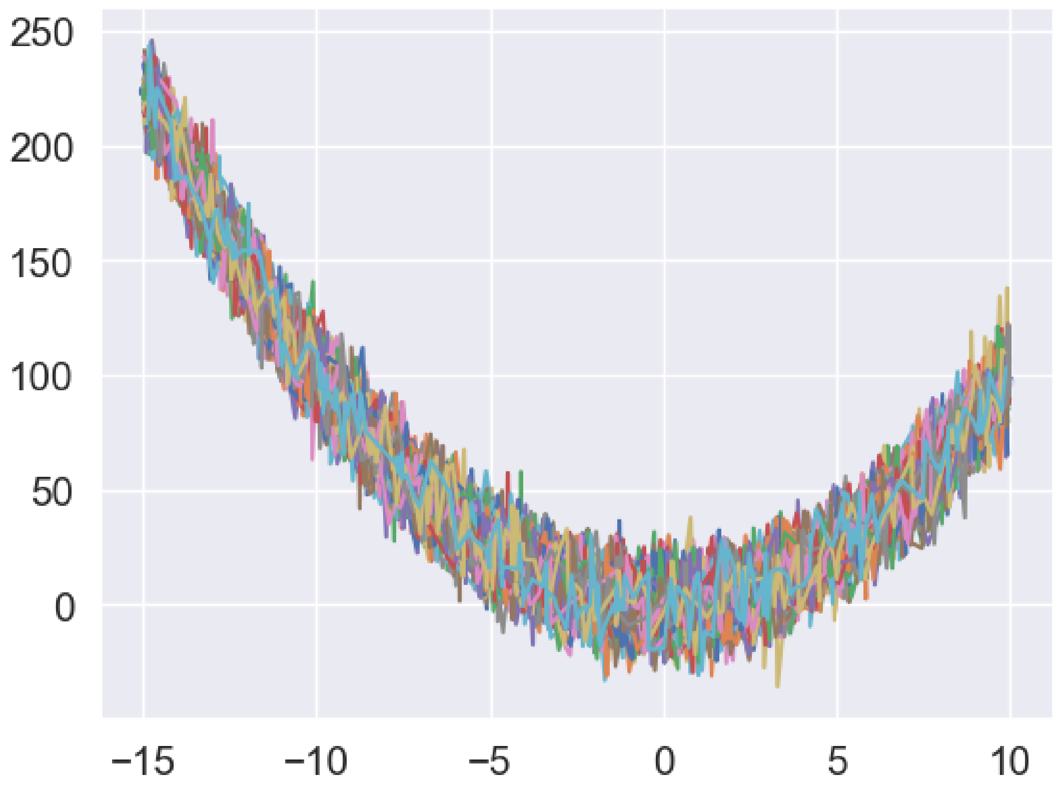
Note that this is exactly the same as the mean squared error computed above as we are developing a finite number of models, and making predictions on a finite number of test data points.

2.2 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.

3 Basic 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 tl
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_neighbo
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_neighbo
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 distributi
                                                                #for continuous hyperparamet

# 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 tl
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)
```

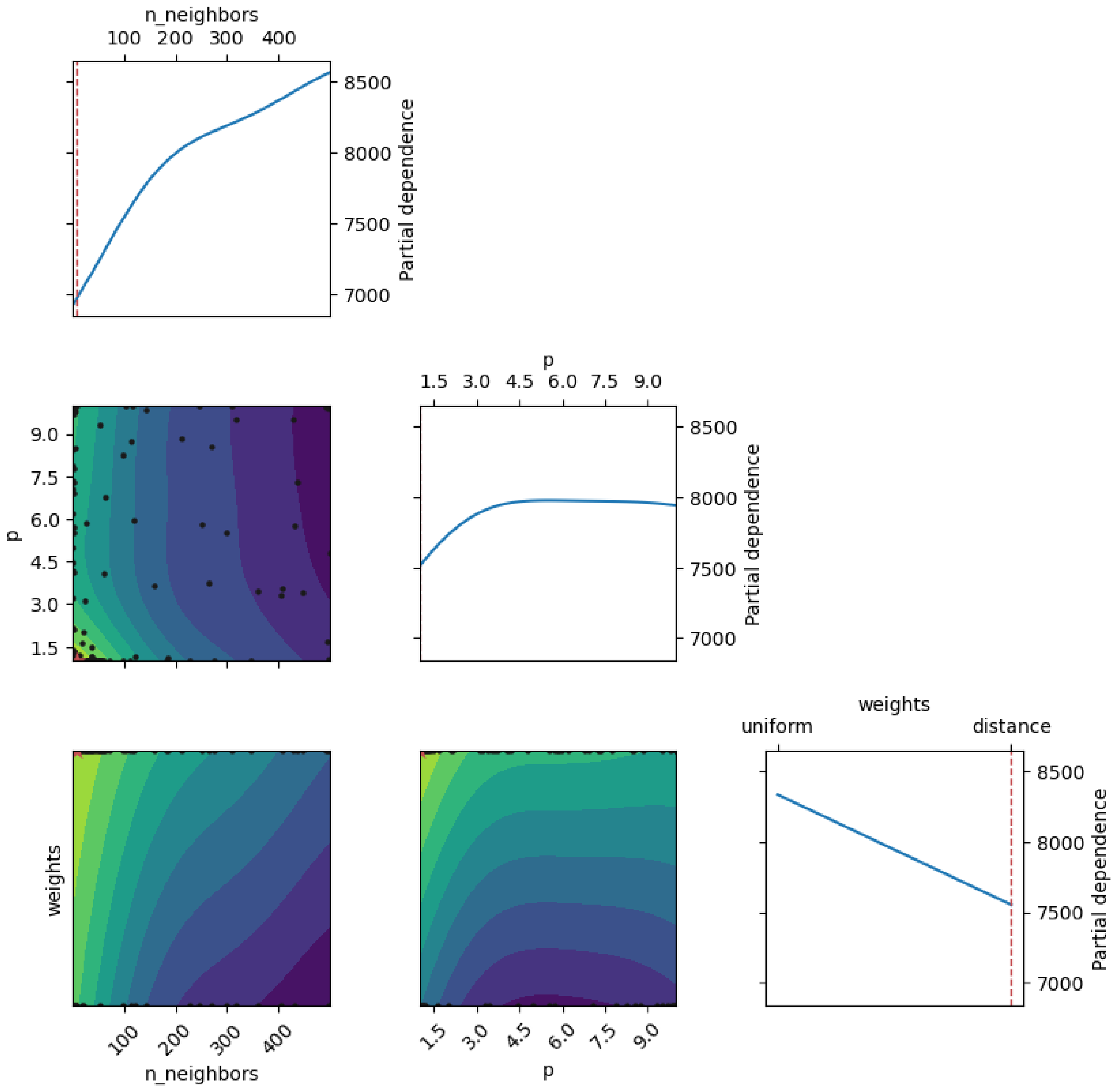
```
5740.432278861367
```

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.

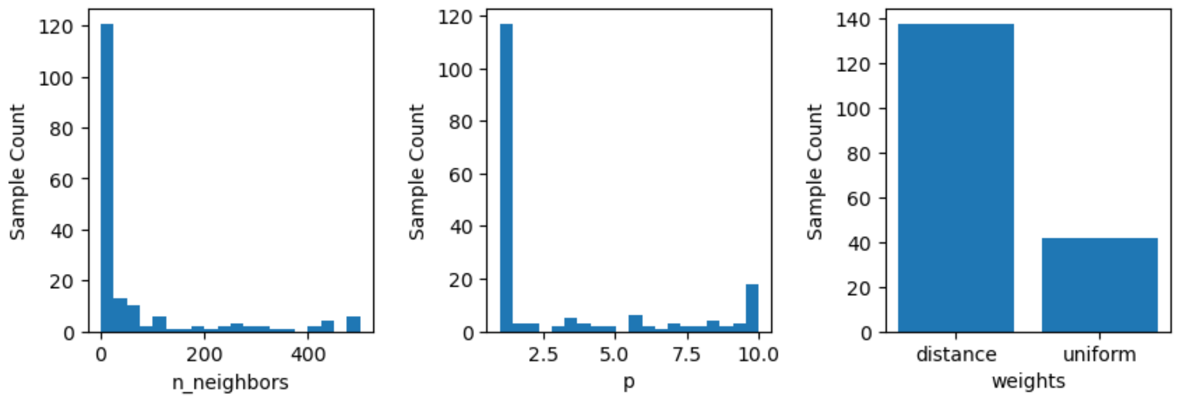
Check the `plot_objective()` documentation to interpret the plots.

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



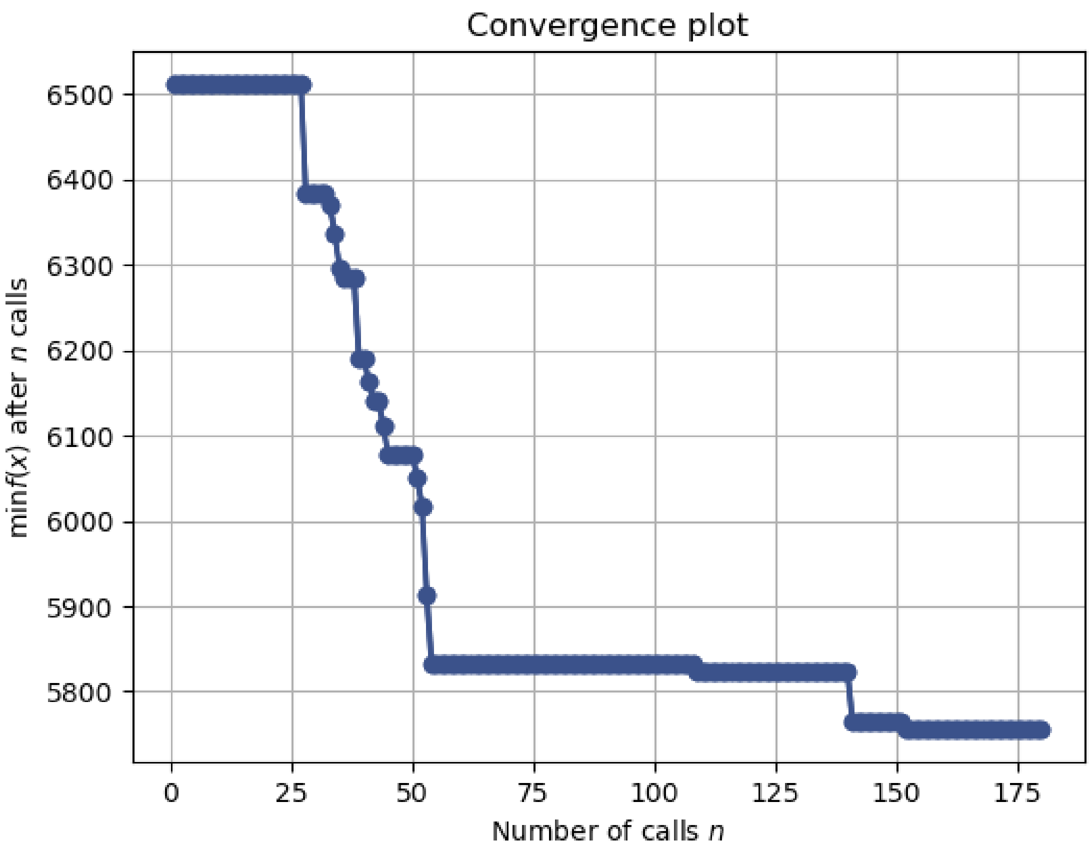
The frequence of individual hyperparameter values considered can also be visualized as below.

```
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:

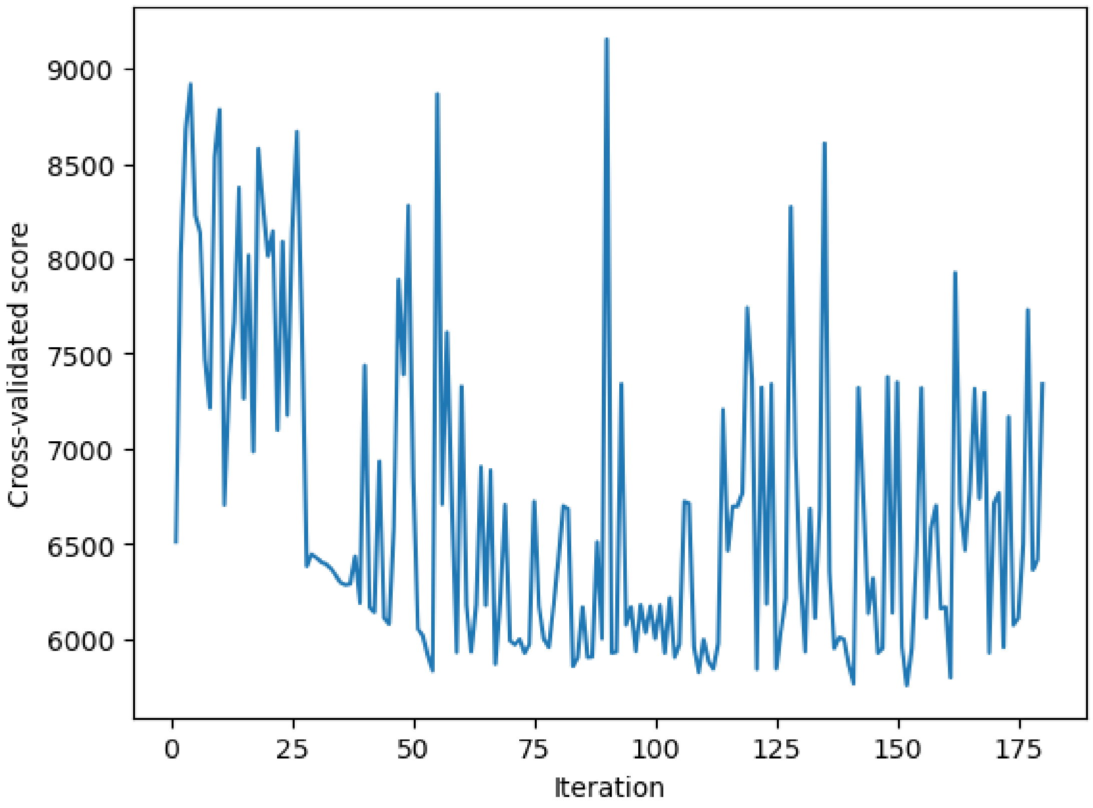
```
gcv.optimizer_results_[0]['x_iters'][53]
```

```
[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

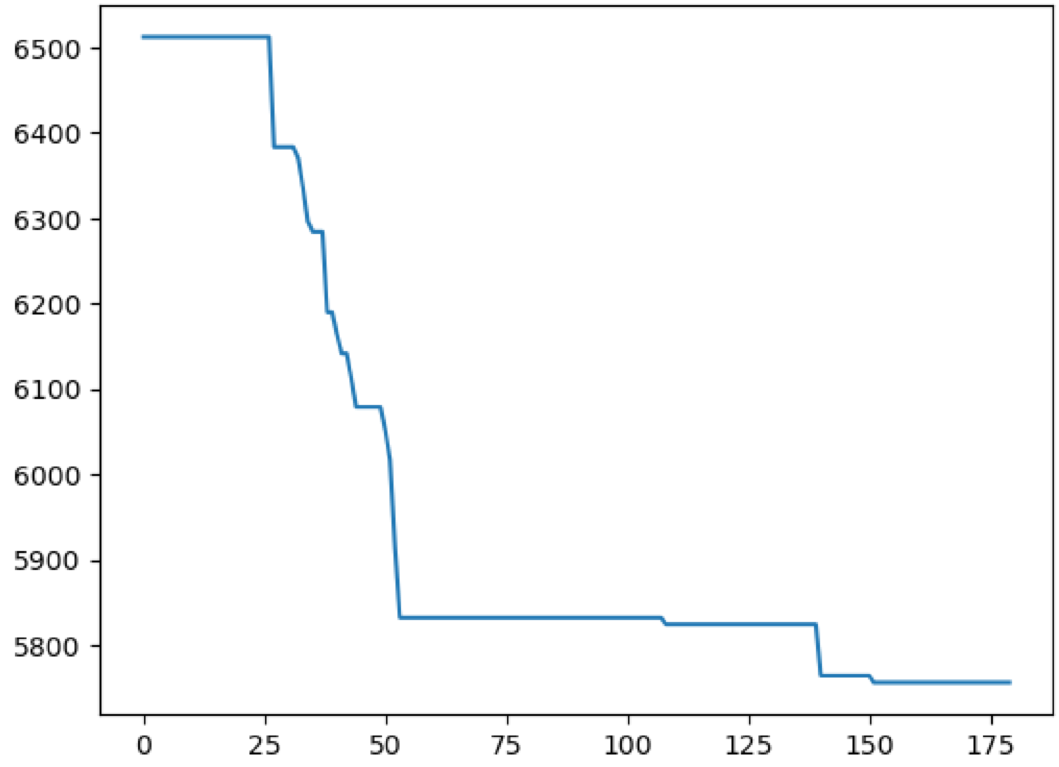
Note that it will be useful monitor the cross-validated score while the Bayesian Search CV code is running, and stop the code as soon as the desired accuracy is reached, or the optimal cross-validated score doesn't seem to improve. The `fit()` method of the `BayesSeaerchCV()` object has a `callback`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

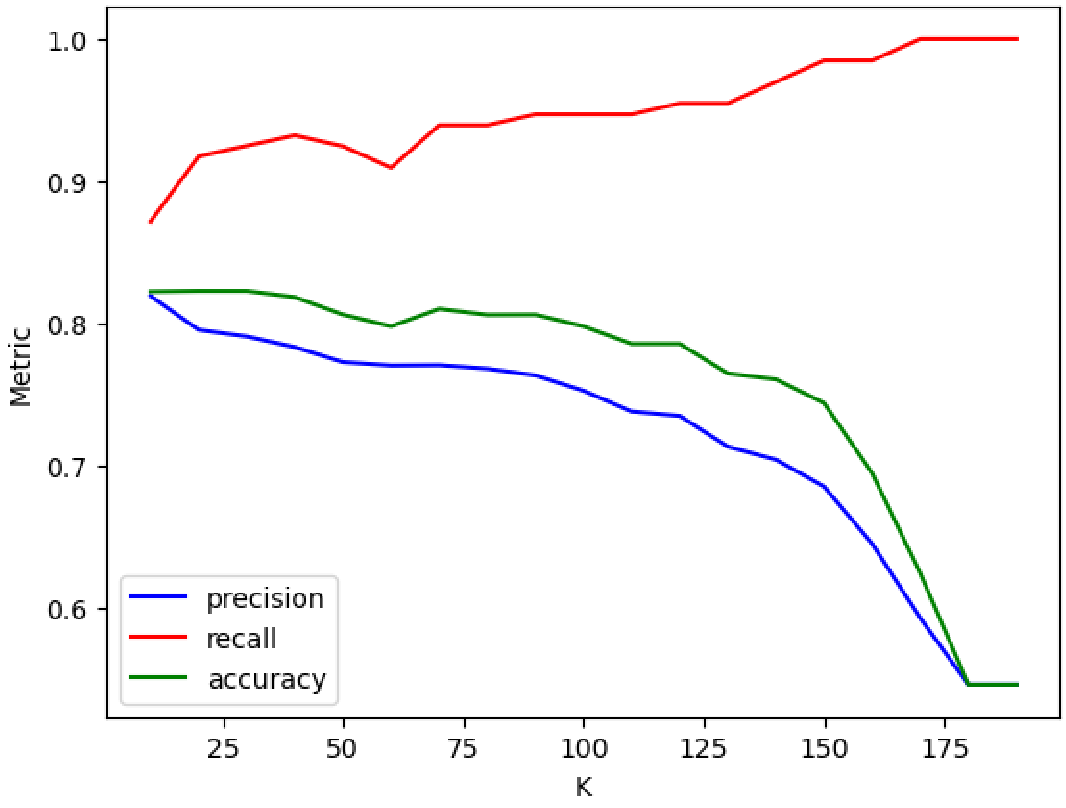
```
120
```

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

```python
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
```

```python
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:

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

```
1.6.1
```

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

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

```
# 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

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

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

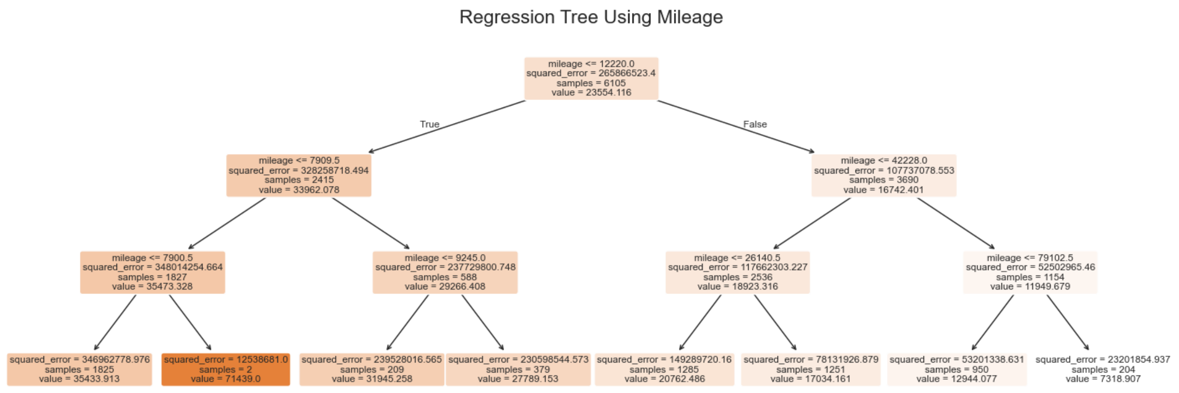
4.2.1 Using only mileage feature

```python
# 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
```

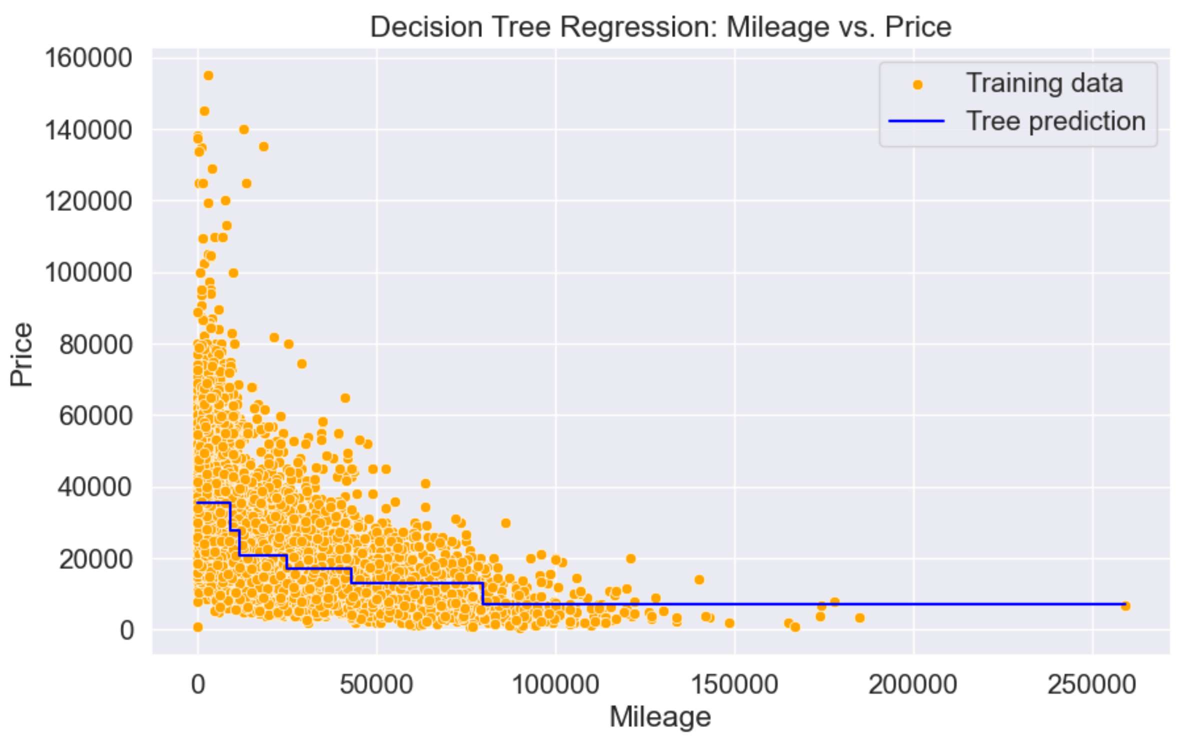
Let's visualize the tree structure

```
# 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 dat

# 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

```
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

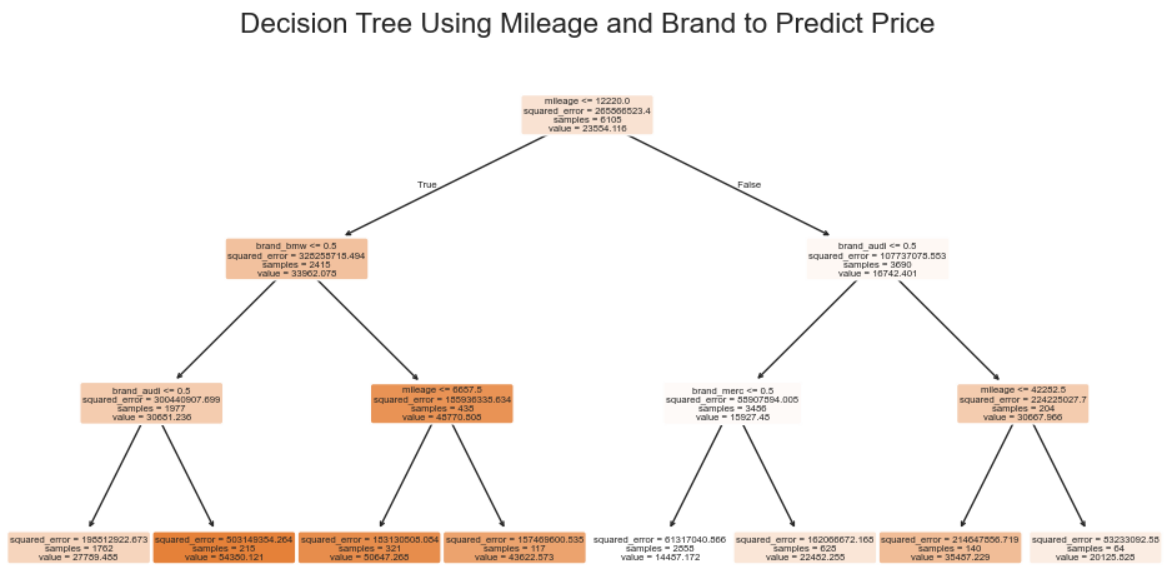
```
# Select features and target
X_train_tree = X_train[['mileage', 'brand']]

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}")
```

```python
# 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.

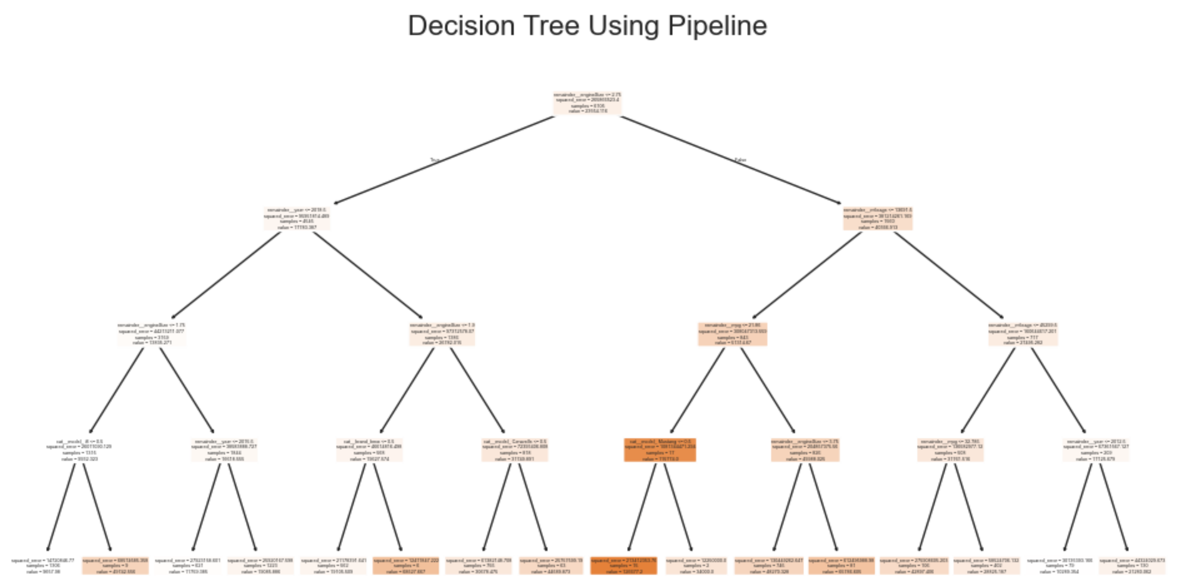
We'll construct a **pipeline** that handles necessary preprocessing steps (e.g., categorical encoding) and fits a **Decision Tree Regressor** in a streamlined and reproducible way.

```python
# 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

```python
# 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)
```

```python
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%
```

```python
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%}")
```

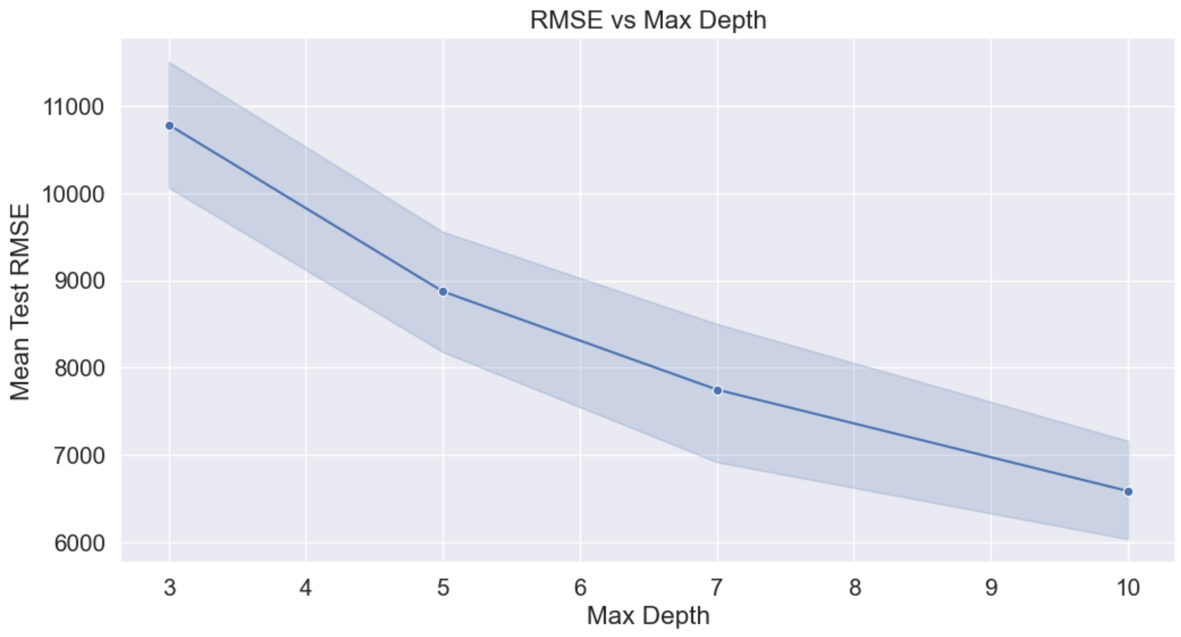
```
=== Best Parameters ===
{'regressor__ccp_alpha': 0.001, 'regressor__max_depth': None, 'regressor__max_features': None

=== 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.

```python
#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	pa
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');
```

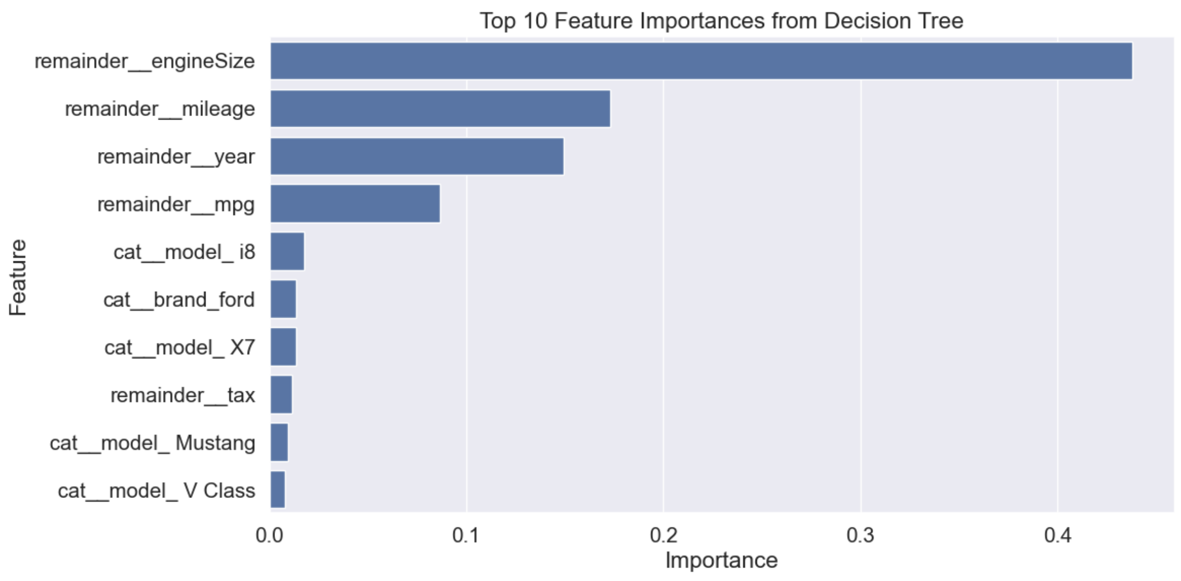


4.3.7 Output feature importance

```
# 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)
```

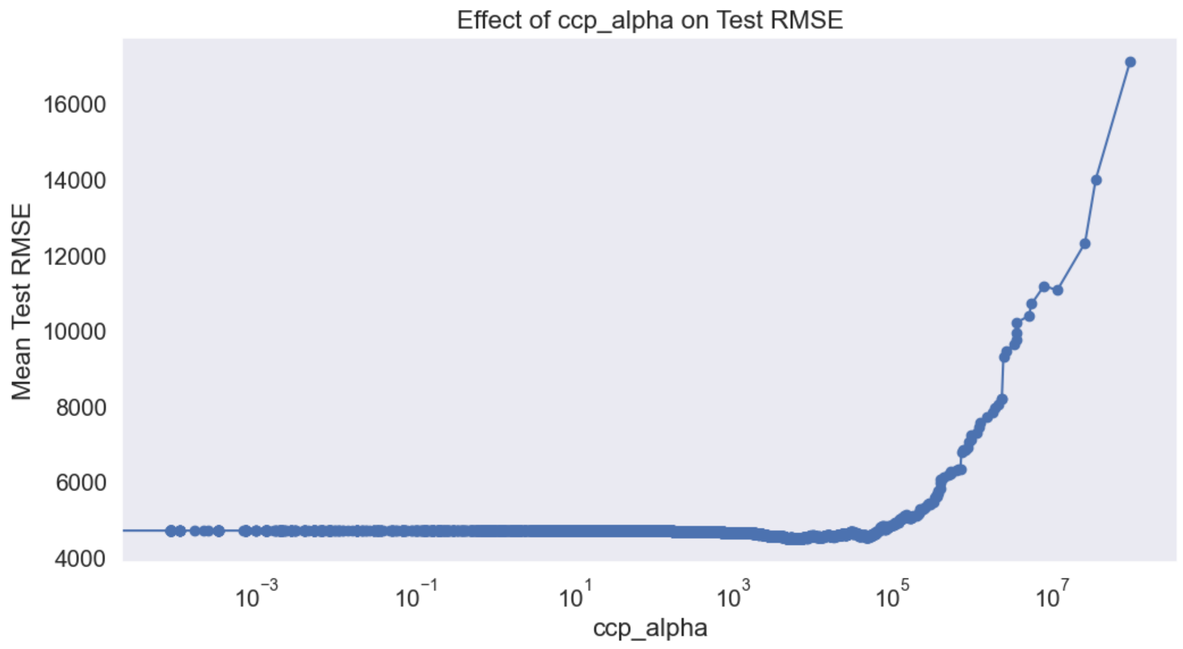
```
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)

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(categ

# 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

# 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	b
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	

```python
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
```

```python
# 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

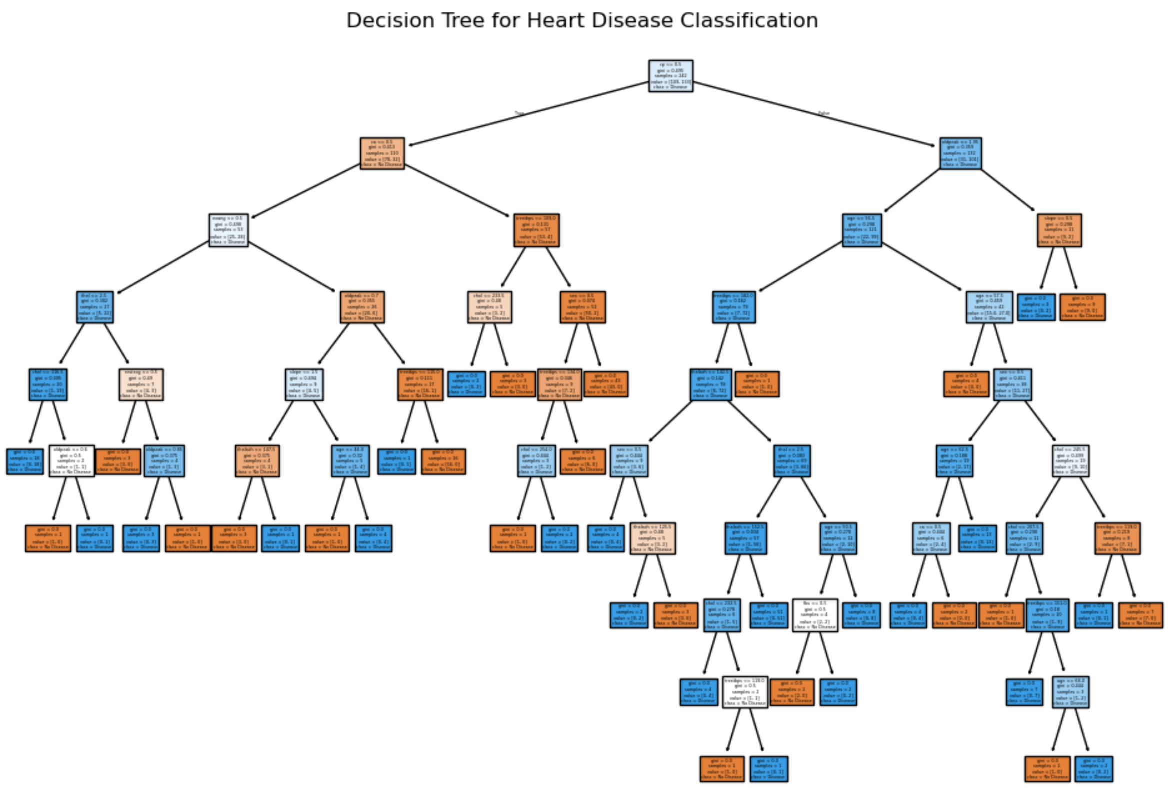
We will build a classification tree to predict whether a person has heart disease, using the default parameters of the decision tree classifier.

```
# 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
```

```
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
```

```
# 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');
```



```
# 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.

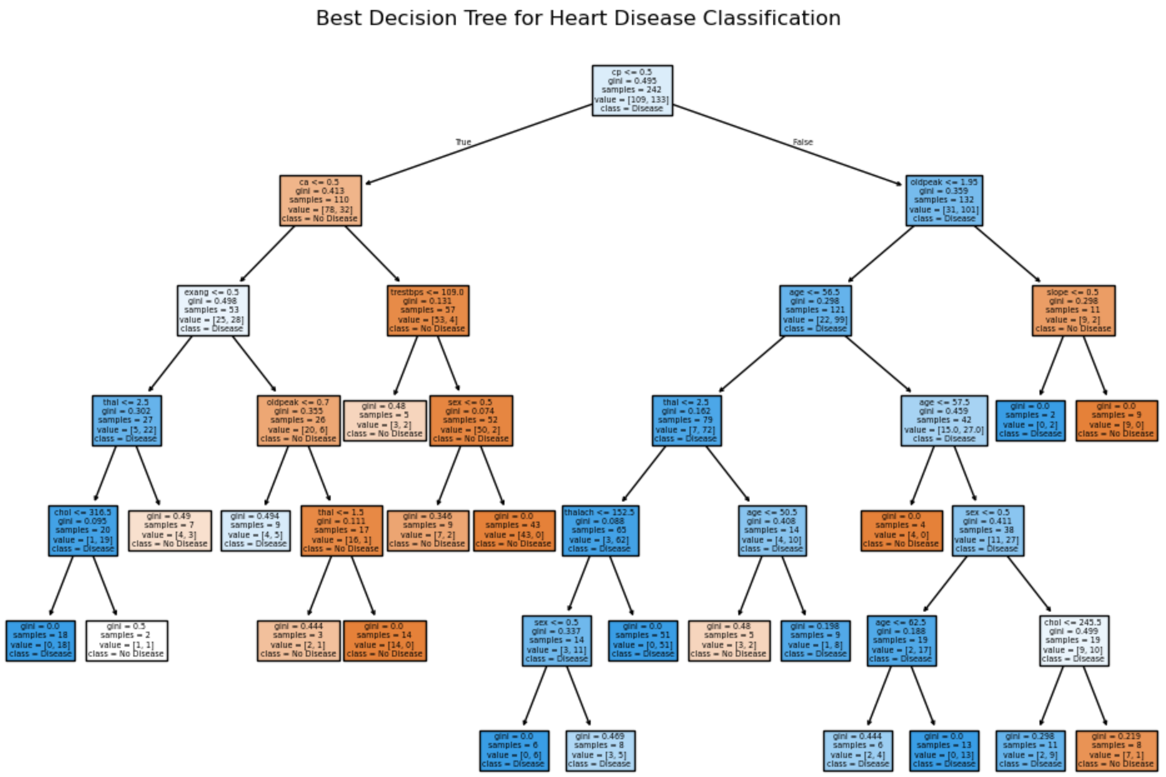
```
# hyperparameter tuning

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_)
```

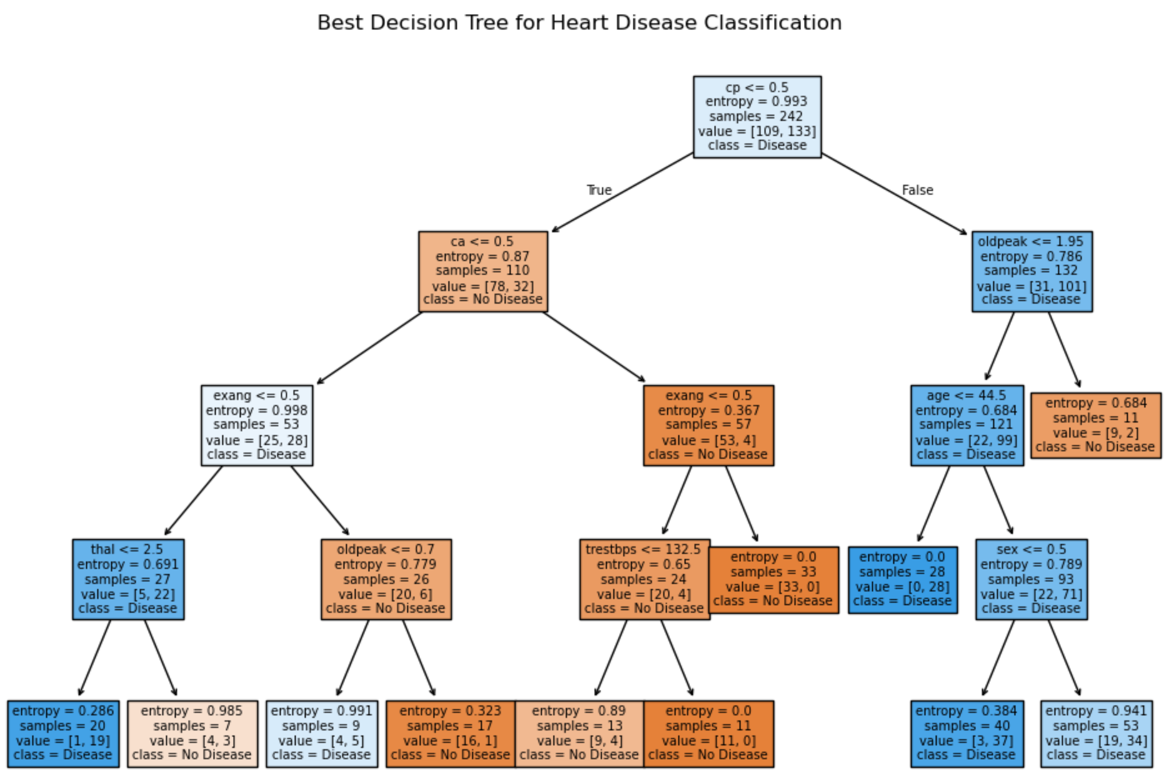
```
# print the best score
print("Best score: ", grid_search_metric.best_score_)
# get the best estimator
best_tree = grid_search_metric.best_estimator_

# 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

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

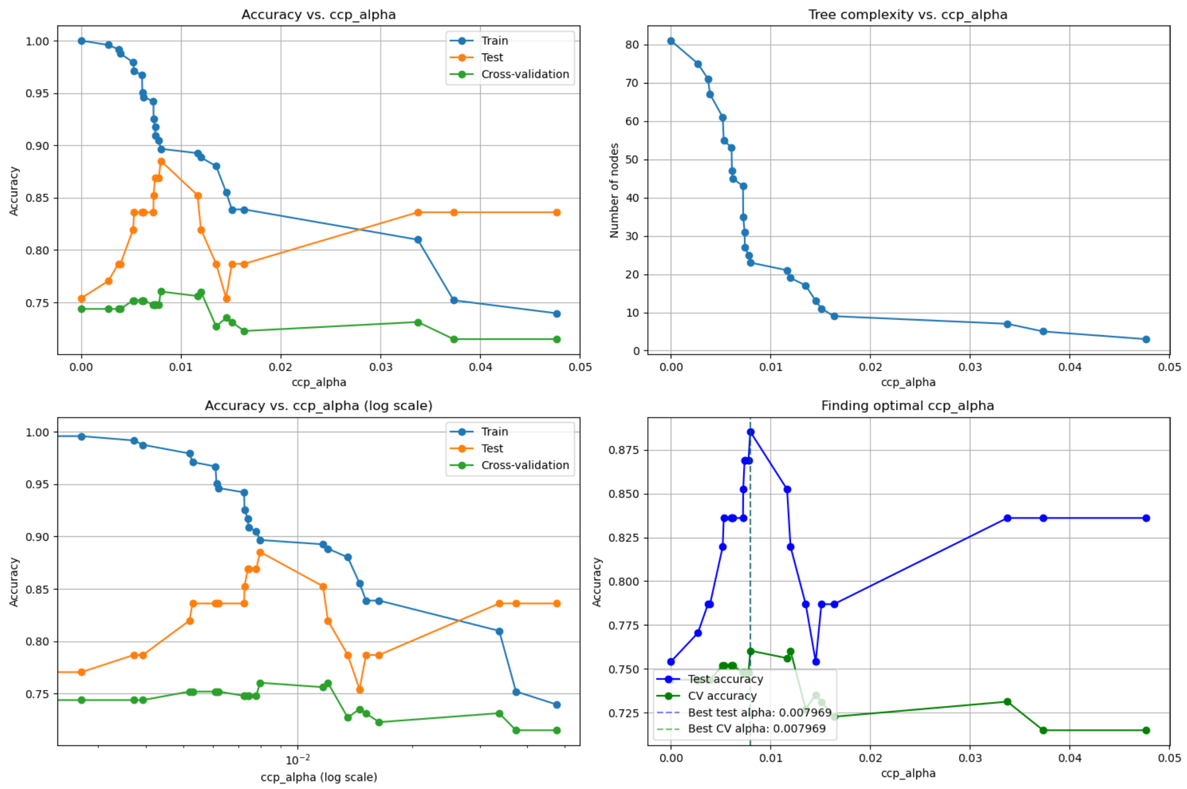
5.3.2 step 2: Create trees with different ccp_alpha values and evaluate their performance

```
# 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
```

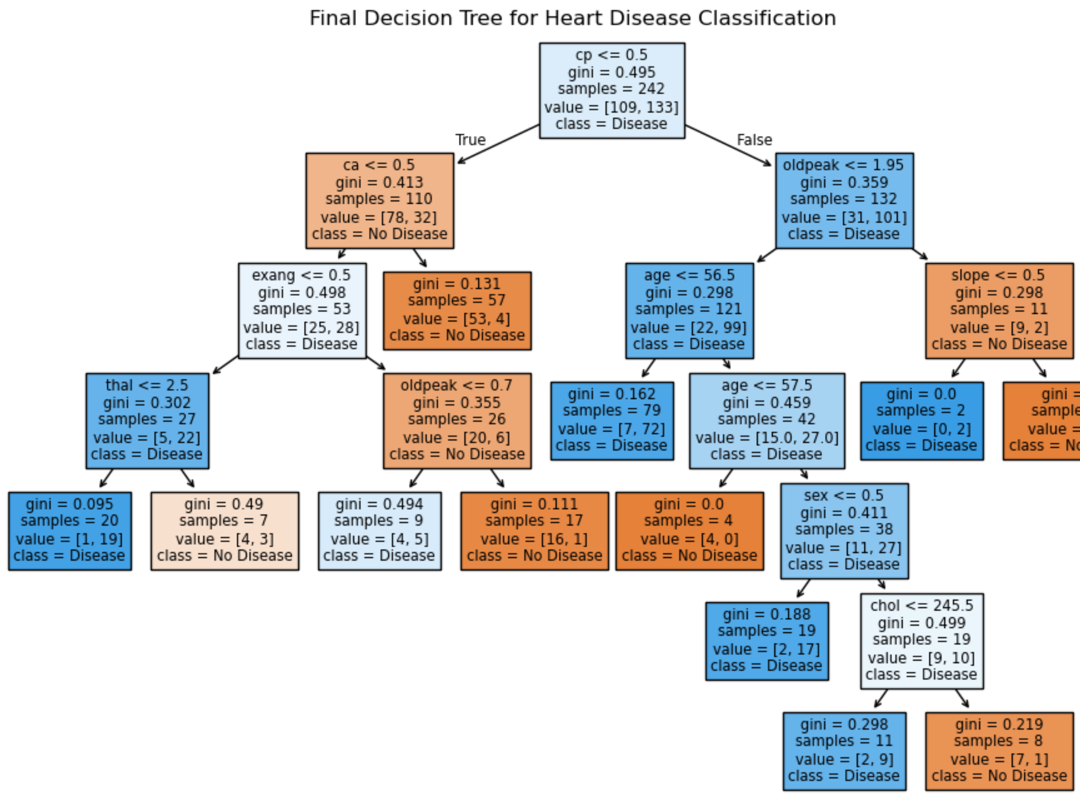


5.3.4 Step 4: Create the final model with the optimal alpha

```
# 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', 'Dis
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**).

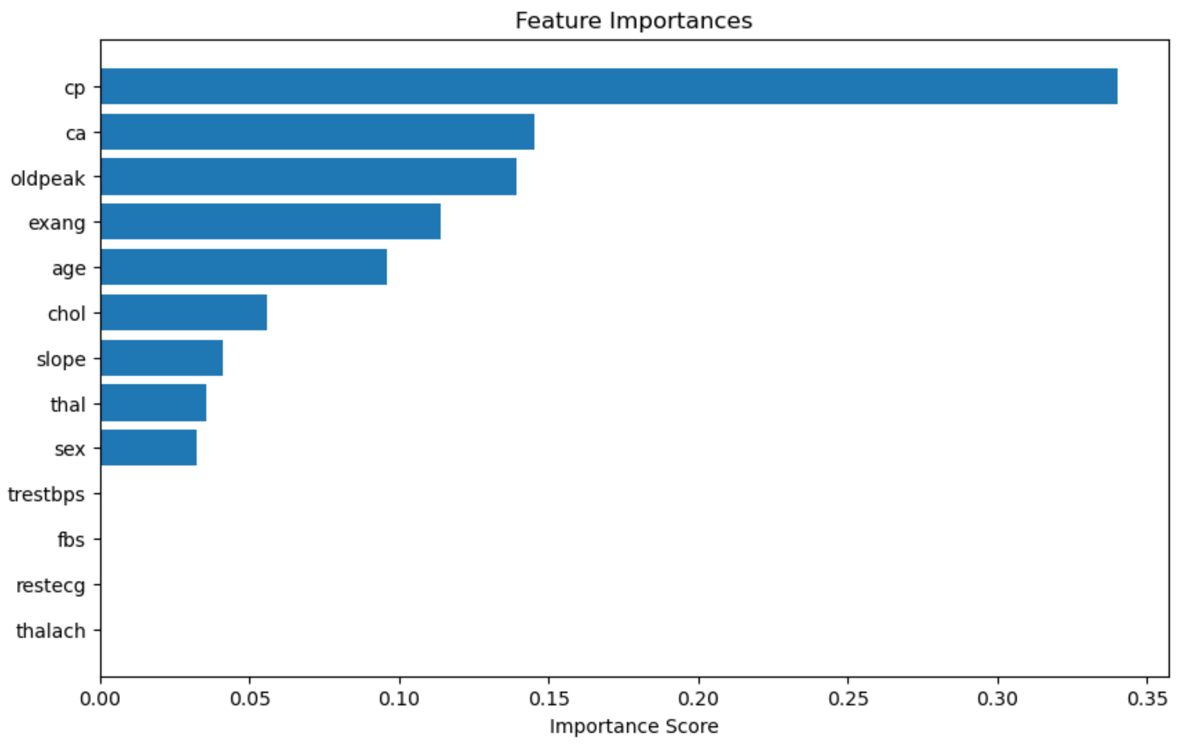
This methodology extends naturally to **ensemble models** like **Random Forests** and **Gradient Boosting**, which average feature importance across all trees in the ensemble.

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()
```



5.4.1 Do We Need Feature Selection or Regularization with Tree Models?

When using **tree-based models** (e.g., Decision Trees, Random Forests, Gradient Boosting) and you have a **small number of predictors**, you typically **don't need to worry much about feature selection or regularization**. Here's why:

- **Tree models inherently ignore uninformative features**. They only split on features that reduce impurity (e.g., Gini or entropy), so irrelevant features tend to receive **zero or very low importance**.
- **Ensemble methods** like Random Forest or Boosting average over many trees. Features that don't help prediction are rarely used across the ensemble.
- **Including unimportant features won't significantly hurt model performance** in small feature spaces. The model will usually ignore them during training.
- **However**, even in small feature sets, irrelevant features may slightly:
- Increase model variance
- Increase training time
- Reduce interpretability

5.4.2 Bottom Line

> If the number of predictors is small, you can typically skip feature selection and regularization when using tree-based models — they will handle irrelevant features gracefully.

5.5 Next Lecture

A single decision tree is highly susceptible to **overfitting**, especially on noisy or complex datasets.

In this notebook, we explored two techniques to reduce variance and improve generalization:

- **Pre-pruning** (e.g., setting `max_depth`, `min_samples_split`, etc)
- **Post-pruning** (e.g., cost-complexity pruning)

In the next lecture, we'll introduce another powerful method to combat overfitting:
Bagging (Bootstrap Aggregating) — an ensemble technique that builds multiple trees and averages their predictions to reduce variance and improve robustness.

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: A Variance Reduction Technique

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

Split the predictors and target, then perform the train-test split

```
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()
```

Encode categorical predictors

```
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)
```

By default, a single decision tree grows to its full depth, which often leads to overfitting as shown below

```
# build a decision tree regressor using the default parameters
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(X_train_final, y_train)
y_pred = tree_reg.predict(X_test_final)
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Test RMSE: {rmse:.2f}, test R^2: {r2:.2f}")

# training rmse and r2
y_train_pred = tree_reg.predict(X_train_final)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f"Train RMSE: {train_rmse:.2f}, train R^2: {train_r2:.2f}")

# print the depth of the tree
print(f"Depth of the tree: {tree_reg.get_depth()}")
# print the number of leaves in the tree
print(f"Number of leaves in the tree: {tree_reg.get_n_leaves()}")
```

```
Test RMSE: 6219.96, test R^2: 0.87
Train RMSE: 0.00, train R^2: 1.00
Depth of the tree: 34
Number of leaves in the tree: 5925
```

As observed, the model achieves an RMSE of 0.00 and an R^2 of 100% on the training data with default parameters, indicating overfitting

To address this, we've previously explored pre-pruning and post-pruning techniques. Another effective approach is bagging, which helps reduce overfitting by lowering model variance.

Next, we'll explore how bagging can improve the performance of unpruned decision trees by reducing variance

```
#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 settings, bagging unpruned trees improves performance, reducing the RMSE from 6219.96 to 3758.10 and increasing the R² score from 0.87 to 0.95.

What about bagging pruned trees? Since pruning improves the performance of a single decision tree, does that mean bagging pruned trees will also outperform bagging unpruned trees? Let's find out through implementation.

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

```
# fit the decision tree regressor
pruned_tree_reg = DecisionTreeRegressor(max_depth=None, min_samples_leaf=1, min_samples_spli
pruned_tree_reg.fit(X_train_final, y_train)

# make predictions on the test set
y_pred = pruned_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))

#print the depth of the tree
print(f"Depth of the tree: {pruned_tree_reg.get_depth()}")
#print the number of leaves in the tree
print(f"Number of leaves in the tree: {pruned_tree_reg.get_n_leaves()}")
```

```
Test RMSE: 4726.17
Test R^2 score: 0.92
Depth of the tree: 33
Number of leaves in the tree: 2558
```

Next, let's apply bagging to these pruned trees using the default settings.

```
#Bagging the results of 10 decision trees to predict car price
bagging_reg = BaggingRegressor(estimator=pruned_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))
```

```python
# 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 bagging the unpruned trees, the performance is slightly worse, with bagging pruned trees the RMSE is 3806.7, bagging the unpruned trees lead to RMSe 3758.1.

Why is bagging tuned trees worse than bagging untuned trees?

In the pruned tree, limiting the maximum depth reduces variance but increases bias, as reflected by the smaller depth and fewer leaves compared to the unpruned tree. Since bagging only reduces variance and does not affect bias, applying it to pruned trees—which have slightly higher bias—results in slightly worse performance than bagging unpruned trees

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 Bagging Doesn't Reduce Bias

Bagging high-variance models can effectively lower overall variance, as long as the individual models are not highly correlated. However, Bagging high-bias models will still produce a high-bias ensemble.

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.

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

# Predict and evaluate on test set
y_pred_single = shallow_tree_reg.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_reg.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))
```

```
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
```

Let's bag these shallow trees

```python
# 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))
```

```
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:

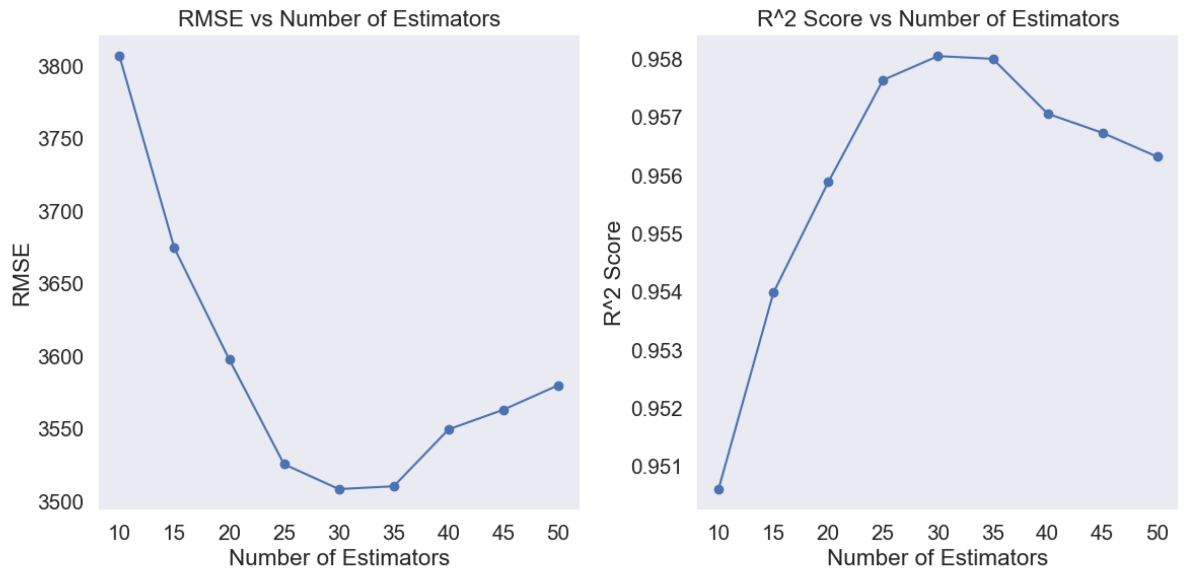
- **Both models show low R² and high RMSE due to the shallow depth (`max_depth=2`).**
- **Bagging cannot fix the high bias inherent in a shallow decision tree.**

6.4 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]
rmse_scores = []
r2_scores = []

# iterate through the number of estimators and fit the model
for n in n_estimators:
    bagging_reg = BaggingRegressor(estimator=pruned_tree_reg, n_estimators=n, random_state=1
                        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))
```

```python
# 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.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.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()
```



Quick Takeaway

- **Increasing the number of estimators initially improves performance**, as seen from the decreasing RMSE and increasing R^2 scores.
- **Performance stabilizes around 30–35 estimators**. Beyond this point, additional trees provide minimal gains.

- Due to the small and possibly noisy test set, performance may appear to **peak and then decline slightly**. However, this is not typical—under normal circumstances, performance **levels off and forms a plateau**.

```
# 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))
```

```
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
```

6.5 OOB Sample and OOB Score in Bagging

When training a **Bagging ensemble**, such as `BaggingClassifier` or `BaggingRegressor`, each base learner is trained on a **bootstrap sample**—a random sample *with replacement* from the original dataset.

6.5.1 What is an OOB Sample?

For each base learner, the data points **not selected** in the bootstrap sample form its **Out-of-Bag (OOB) sample**. On average, about **1/3 of the original data points** are not included in each bootstrap sample. These unused samples are called **OOB samples**.

6.5.2 What is OOB Score?

Each base learner can be evaluated on its corresponding OOB sample—i.e., the instances it did *not* see during training. This provides a **built-in validation mechanism** without needing an explicit validation set or cross-validation.

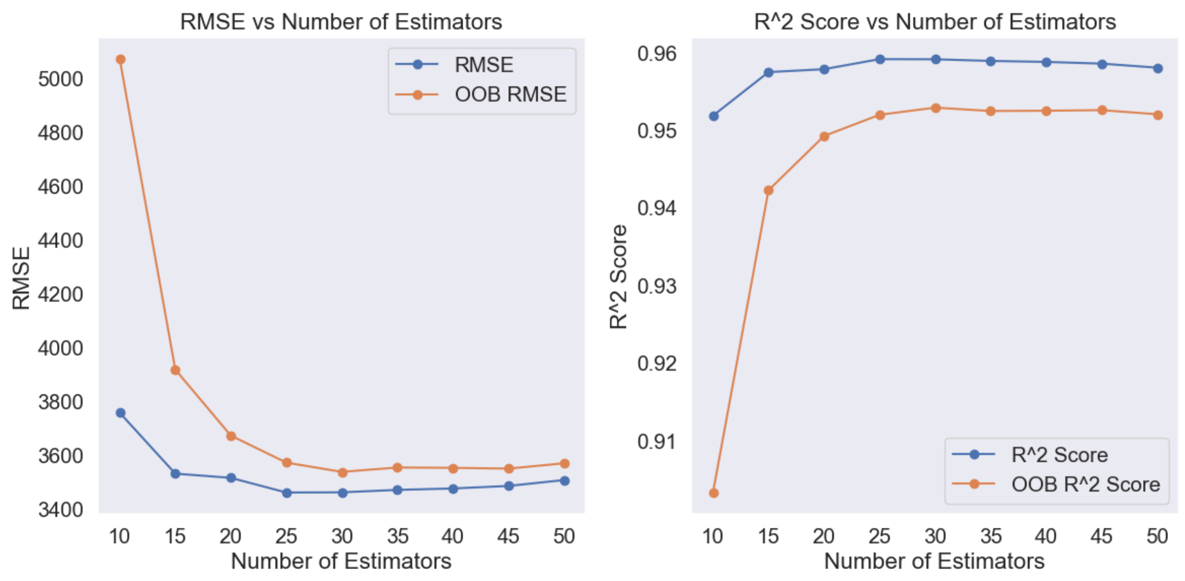
The OOB score is the **average performance** (e.g., accuracy for classifiers, R^2 for regressors) of the ensemble evaluated on OOB samples.

> **Note:** By default, the `oob_score` option is turned **off** in scikit-learn. You must explicitly set `oob_score=True` to enable it, as shown below.

```
from sklearn.ensemble import BaggingClassifier

bagging_clf = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=100,
    oob_score=True,
    random_state=42
)
bagging_clf.fit(X_train, y_train)

# Access the OOB score
print(f"OOB Score: {bagging_clf.oob_score_:.4f}")
```

```
n_estimators = [ 10, 15, 20, 25,30, 35, 40, 45, 50]
rmse_scores = []
r2_scores = []
oob_scores = []
oob_rmse_scores = []
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

- **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**.

```
# 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 OOB RMSE: 30
Best OOB RMSE value: 3539.1
```

6.6 Bagging Hyperparameter Tuning

To further improve the performance of our bagging model, we can tune key hyperparameters such as:

- `n_estimators`: the number of base estimators in the ensemble,
- `max_features`: the maximum number of features considered at each split,
- `max_samples`: the size of each bootstrap sample used to train base estimators,
- `bootstrap`: whether sampling is performed with replacement (`True`) or without (`False`),
- `bootstrap_features`: whether features are sampled with replacement when selecting subsets of features for each estimator.

By systematically exploring different combinations of these parameters, we aim to identify the optimal settings that enhance predictive accuracy while maintaining good generalization.

There are two common approaches for tuning these hyperparameters: - **Cross-validation**, which provides a robust estimate of model performance, and
- **Out-of-Bag (OOB) score**, which offers an efficient built-in alternative without needing a separate validation set.

6.6.1 Tuning with Cross-Validation

Next, let's use `GridSearchCV` to tune these hyperparameters and identify the best combination for improved model performance.

```
# 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 R^2 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.

6.6.2 Tuning with Out-of-Bag (OOB) Score

As an alternative to cross-validation, we can use the **Out-of-Bag (OOB) score** to evaluate model performance during training.
This method is more efficient for large datasets, as it avoids the need to split data or run multiple folds.

By enabling `oob_score=True`, we can monitor performance on unseen data points (those not included in each bootstrap sample) and use this score to guide hyperparameter tuning.

```python
from sklearn.model_selection import ParameterGrid
# tune the hyperparameters of the decision tree regressor using oob score
# Hyperparameter grid
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],  # Required for OOB
    'bootstrap_features': [True, False]
}

# Track best parameters and OOB score
best_score = -np.inf
best_params = {}

# Iterate over all hyperparameter combinations
```

```python
for params in ParameterGrid(param_grid):
    # Train model with current params and OOB score enabled
    model = BaggingRegressor(
        estimator=tree_reg,
        **params,
        oob_score=True,
        random_state=42,
        n_jobs=-1
    )
    model.fit(X_train_final, y_train)

    # Get OOB score (higher is better for R², lower for RMSE)
    current_score = model.oob_score_

    # Update best params if current score is better
    if current_score > best_score:
        best_score = current_score
        best_params = params

# Best model
best_model = BaggingRegressor(
    estimator=tree_reg,
    **best_params,
    oob_score=True,
    random_state=42,
    n_jobs=-1
)
best_model.fit(X_train_final, y_train)

print("Best Hyperparameters:", best_params)
print("Best OOB Score:", best_score)
```

```
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(
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(
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(
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(
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(
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(
```

```
Best Hyperparameters: {'bootstrap': True, 'bootstrap_features': False, 'max_features': 0.75,
Best OOB Score: 0.9587800605892783
```

```python
# output the test RMSE and R^2 score with the best model
y_pred_best_model = best_model.predict(X_test_final)
rmse_best_model = root_mean_squared_error(y_test, y_pred_best_model)
r2_best_model = r2_score(y_test, y_pred_best_model)
print("Test RMSE with best model:", round(rmse_best_model, 2))
print("Test R^2 score with best model:", round(r2_best_model, 2))
```

```
Test RMSE with best model: 3418.55
Test R^2 score with best model: 0.96
```

6.6.3 Comparing Hyperparameter Tuning: Cross-Validation vs. OOB Score

Aspect	Cross-Validation (CV)	Out-of-Bag (OOB) Score
Mechanism	Splits training data into multiple folds to validate	Uses unused (out-of-bag) samples in each bootstrap
Requires Manual Splits?	Yes	No — internal to bagging process
Efficiency	Slower, especially for large datasets	Faster and more efficient for large datasets
Bias-Variance Tradeoff	More stable and less biased performance estimate	Slightly more variable and can underestimate accuracy
Availability	Available for all models	Only available when `bootstrap=True` in bagging
Integration in Sklearn	Built-in via `GridSearchCV`	Must be implemented manually for tuning
Flexibility	Works with any model type	Only works with bagging-based models
Use Case	Ideal for robust model comparison	Great for quick tuning on large datasets
Scoring Access	`.best_score_` from `GridSearchCV`	`.oob_score_` from trained model

6.6.3.1 Best Practices for Imbalanced Classification

- **Prefer Cross-Validation**, especially with:
  - `StratifiedKFold` to maintain class distribution in each fold.
  - Custom metrics (e.g., F1-score, ROC-AUC, balanced accuracy) using `scoring=`.
- **Be cautious using OOB score:**
  - OOB score may be misleading for **rare classes**, especially when `n_estimators` is small.
  - Only use it for **rough estimates** or **early tuning** when computational efficiency is critical.

Summary

- **Use Cross-Validation** when:
  - You want robust, model-agnostic performance evaluation.
  - You need precise comparisons between different model types or pipelines.
  - You work on imbalanced classification task
- **Use OOB Score** when:

  - You're working with large datasets and want faster tuning.
  - Your model is based on bagging (e.g., `BaggingClassifier`, `RandomForestClassifier`).
  - You want to avoid manual train/validation splits.

Note: Scikit-learn's `GridSearchCV` **does not use OOB score** for tuning—even if `oob_score=True` is set. To use OOB for tuning, you must loop over hyperparameters manually and evaluate using `.oob_score_`.

6.7 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 Random Forests

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.

```
# 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, RandomForestRegressor, Rando

# 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
```

7.1 Motivation: Bagging Revisited

In Bagging (Bootstrap Aggregating), we:

- Train many trees on different bootstrap samples of the training data.

- Aggregate their predictions by averaging (regression) or voting (classification).

  Bagging helps reduce variance But if the trees are too similar (i.e., highly correlated), averaging won't help as much.

```python
# 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

```python
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()
```

```python
preprocessor = ColumnTransformer(
    transformers=[
        ('num', FunctionTransformer(), numerical_feature),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_feature)
    ],
    remainder='passthrough'
)
```

7.1.1 Let's build a single decision tree and output its performance

```
# build a single decsision tree regressor
single_tree_regressor = DecisionTreeRegressor(random_state=0)

# pipeline for the single decision tree regressor
single_tree_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('tree', single_tree_regressor)
])
# fit the pipeline to the training data
single_tree_pipeline.fit(X_train, y_train)
# make predictions on the test data
y_pred_single_tree = single_tree_pipeline.predict(X_test)
# calculate the RMSE and R^2 score
rmse_single_tree = root_mean_squared_error(y_test, y_pred_single_tree)
r2_single_tree = r2_score(y_test, y_pred_single_tree)
print(f'Single Tree RMSE: {rmse_single_tree:.2f}')
print(f'Single Tree R^2: {r2_single_tree:.2f}')

# calculate the RMSE and R^2 score for the training data
y_pred_train_single_tree = single_tree_pipeline.predict(X_train)
rmse_train_single_tree = root_mean_squared_error(y_train, y_pred_train_single_tree)
r2_train_single_tree = r2_score(y_train, y_pred_train_single_tree)
print(f'Single Tree Train RMSE: {rmse_train_single_tree:.2f}')
print(f'Single Tree Train R^2: {r2_train_single_tree:.2f}')
```

```
Single Tree RMSE: 5073.81
Single Tree R^2: 0.91
Single Tree Train RMSE: 0.00
Single Tree Train R^2: 1.00
```

```
# single tree depth
tree_depth = single_tree_pipeline.named_steps['tree'].get_depth()
print(f"Depth of the single decision tree: {tree_depth}")
```

```
Depth of the single decision tree: 34
```

7.1.2 Let's Build a Bagging Tree with Bootstrap Sampling to Reduce Variance

By default, `bootstrap=True`, meaning each training set is created by sampling **with replacement** from the original dataset.

```
# bagging with bootstrap
bagging_with_bootstrap_regressor = BaggingRegressor(
    estimator=DecisionTreeRegressor(random_state=0),
    n_estimators=50,
    random_state=42
)

# create a pipeline with the preprocessor and the bagging regressor
bootstrap_bagging_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('bagging', bagging_with_bootstrap_regressor)
])

# fit the pipeline to the training data
bootstrap_bagging_pipeline.fit(X_train, y_train)
# make predictions on the test data
y_pred = bootstrap_bagging_pipeline.predict(X_test)
# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f'RMSE using bootstraping: {rmse:.2f}')
print(f'R^2 using bootstraping: {r2:.2f}')
# calculate the training rmse and r^2 score
y_train_pred = bootstrap_bagging_pipeline.predict(X_train)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f'Training RMSE using bootstraping: {train_rmse:.2f}')
print(f'Training R^2 using bootstraping: {train_r2:.2f}')
```

```
RMSE using bootstraping: 3756.85
R^2 using bootstraping: 0.95
Training RMSE using bootstraping: 1395.75
Training R^2 using bootstraping: 0.99
```

The test RMSE improved significantly from 5073 to 3756, and the R^2 score increased from 0.91 to 0.95 after applying bagging.

7.1.3 Let's Build a Bagging Tree Without Bootstrap Sampling

Now, we'll turn off bootstrap sampling (`bootstrap=False`) and observe how it affects the bagging model's performance.

```
bagging_without_bootstrap_regressor = BaggingRegressor(
    estimator=DecisionTreeRegressor(random_state=0),
    bootstrap=False,
    n_estimators=50,
    random_state=42
)

# create a pipeline with the preprocessor and the bagging regressor
without_bootstrap_bagging_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('bagging', bagging_without_bootstrap_regressor)
])

# fit the pipeline to the training data
without_bootstrap_bagging_pipeline.fit(X_train, y_train)
# make predictions on the test data
y_pred = without_bootstrap_bagging_pipeline.predict(X_test)

# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f'RMSE without bootstrap sampling: {rmse:.2f}')
print(f'R^2 without bootstrap sampling: {r2:.2f}')

# calculate the training rmse and r^2 score
y_train_pred = without_bootstrap_bagging_pipeline.predict(X_train)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f'Training RMSE without bootstrap sampling: {train_rmse:.2f}')
print(f'Training R^2 without bootstrap sampling: {train_r2:.2f}')
```

```
RMSE: 4667.43
R^2: 0.93
Training RMSE: 0.00
Training R^2: 1.00
```

As observed from the results, the performance of bagging without bootstrap sampling is worse (RMSE: 4667) compared to using bootstrap sampling (RMSE: 3756).

7.1.4 Why Does Bagging Without Bootstrap Perform Worse?

When `bootstrap=True`, each tree in the ensemble is trained on a different bootstrap sample — a random sample drawn **with replacement** from the training data. This process has two key effects:

- It **introduces diversity** among the individual trees.
- It **reduces correlation** between trees.

This diversity is the **core strength** of bagging: even though individual trees may overfit, their errors tend to cancel out when their predictions are averaged, leading to improved generalization.

7.1.5 Why Can Bagging Without Bootstrap Still Show Slight Improvement?

When `bootstrap=False`, all trees are trained on the **same full dataset**, removing the primary source of diversity in bagging. As a result, the **variance reduction benefit** from averaging is significantly weakened.

However, even when trees are trained on the same data, slight variations can still arise due to **internal randomness** in how decision trees are constructed. For example:

- When multiple splits yield the same information gain, one split may be selected **randomly**.
- **Ties** between split candidates can be broken differently.
- **Minor numerical differences** can occur due to floating-point operations.

These small variations cause the trees to differ slightly, allowing the ensemble to achieve **some variance reduction**, which can **slightly improve generalization** compared to a single decision tree.

> However, this improvement is typically **much smaller** than the improvement achieved when using full bootstrap sampling (`bootstrap=True`).

7.2 Random Forest

While **diversity** is the core strength of bagging, **Random Forest** further improves upon bagging by introducing **even more diversity among the individual trees**.

The goal of Random Forest is to **further decorrelate the trees**, which leads to improved generalization and predictive performance.

7.2.1 Idea Behind Random Forest

Random Forest introduces an additional source of randomness:

- At each split in a tree, instead of considering **all predictors**, Random Forest randomly selects a **subset of predictors** to evaluate.

This approach:

- Increases **diversity** among the trees.
- Decreases **correlation** between trees.
- Further **reduces the variance** of the aggregated model.

  Result: Random Forest generally achieves better performance than standard bagging, especially on high-dimensional datasets.

7.2.2 Key Hyperparameter Comparison

Hyperparameter	**Bagging**	**Random Forest**
`bootstrap`	Yes	Yes
`max_features`	All features considered at each split	Random subset of features at each split
`oob_score`	Often used for evaluation	Often used for evaluation
`n_estimators`	Number of trees	Number of trees

7.2.3 Let's Build a Random Forest Model Using the Default Settings

The `max_features` hyperparameter controls the number of features considered when searching for the best split at each node.
By default, `max_features=1.0`, meaning **all features** are considered at every split, similar to standard bagging.

```
from sklearn.ensemble import RandomForestRegressor

rf_regressor = RandomForestRegressor(
    n_estimators=50,
    random_state=42
)

# create a pipeline with the preprocessor and the bagging regressor
rf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('rf', rf_regressor)
])

# fit the pipeline to the training data
rf_pipeline.fit(X_train, y_train)
# make predictions on the test data
y_pred = rf_pipeline.predict(X_test)

# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f'RMSE using random forest: {rmse:.2f}')
print(f'R^2 using random forest: {r2:.2f}')

# calculate the training rmse and r^2 score
y_train_pred = rf_pipeline.predict(X_train)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f'Training RMSE using random forest: {train_rmse:.2f}')
print(f'Training R^2 using random forest: {train_r2:.2f}')
```

```
RMSE: 3747.58
R^2: 0.95
Training RMSE: 1395.03
Training R^2: 0.99
```

This result is close to that of the bagging model with bootstrap sampling (RMSE: 3756 vs. 3747).
To further decorrelate the trees, we can adjust the `max_features` parameter. Reducing `max_features` limits the number of features considered at each split, which increases diversity among the trees and helps further reduce variance.

7.2.4 Let's Build a Random Forest Model with `sqrt` max_features

There are two common options to reduce the number of features considered at each split: `sqrt` and `log2`.
Here, we will set `max_features='sqrt'` and observe how it affects the model's performance.

```python
rf_sqrt_regressor = RandomForestRegressor(
    n_estimators=50,
    max_features='sqrt',
    random_state=42
)

# create a pipeline with the preprocessor and the bagging regressor
rf_sqrt_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('bagging', rf_sqrt_regressor)
])

# fit the pipeline to the training data
rf_sqrt_pipeline.fit(X_train, y_train)
# make predictions on the test data
y_pred = rf_sqrt_pipeline.predict(X_test)

# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f'RMSE: {rmse:.2f}')
print(f'R^2: {r2:.2f}')

# calculate the training rmse and r^2 score
y_train_pred = rf_sqrt_pipeline.predict(X_train)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f'Training RMSE: {train_rmse:.2f}')
print(f'Training R^2: {train_r2:.2f}')
```

```
RMSE: 3424.28
R^2: 0.96
Training RMSE: 1279.85
Training R^2: 0.99
```

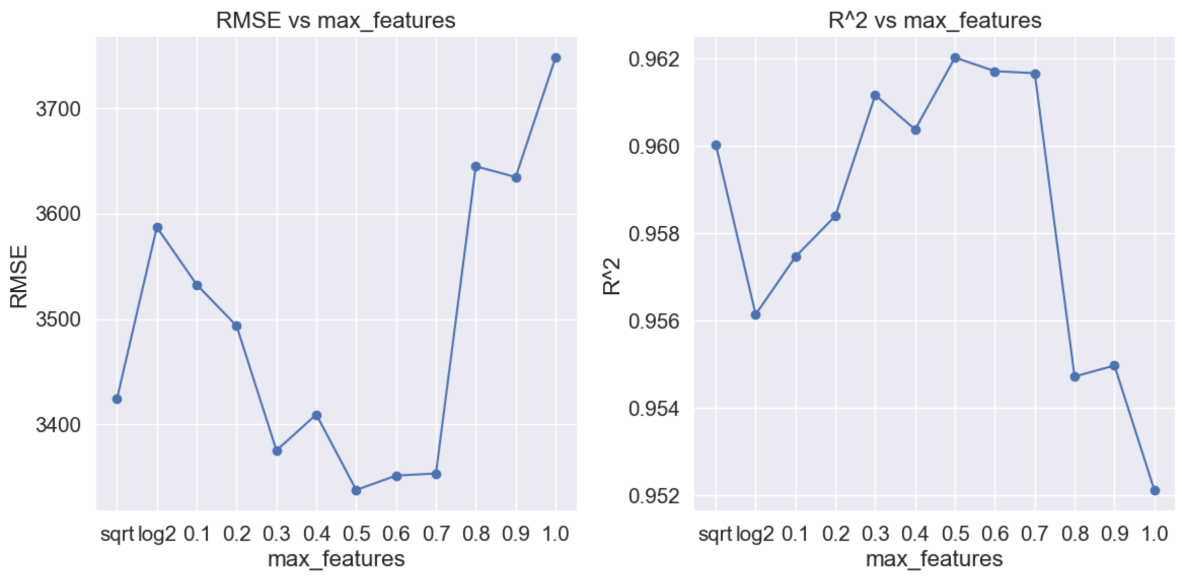
By using `sqrt` for `max_features`, we further decorrelate the trees, resulting in a lower RMSE of 3424 compared to 3747 when using all features at each split.

7.3 Let's Explore How `max_features` Affects Performance

The `max_features` parameter controls the degree of feature decorrelation among trees. Let's explore different values:

```
# explore how the max_features parameter affects the model performance
max_features = ['sqrt', 'log2', 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
rmse_list = []
r2_list = []
for max_feature in max_features:
    rf_regressor = RandomForestRegressor(
        n_estimators=50,
        max_features=max_feature,
        random_state=42
    )

    # create a pipeline with the preprocessor and the bagging regressor
    rf_pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('bagging', rf_regressor)
    ])

    # fit the pipeline to the training data
    rf_pipeline.fit(X_train, y_train)
    # make predictions on the test data
    y_pred = rf_pipeline.predict(X_test)

    # calculate the RMSE and R^2 score
    rmse = root_mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    rmse_list.append(rmse)
    r2_list.append(r2)
```

```
# plot the RMSE and R^2 score against the max_features parameter
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(max_features, rmse_list, marker='o')
plt.xlabel('max_features')
plt.ylabel('RMSE')
plt.title('RMSE vs max_features')
plt.grid(True)
plt.subplot(1, 2, 2)
plt.plot(max_features, r2_list, marker='o')
plt.xlabel('max_features')
plt.ylabel('R^2')
plt.title('R^2 vs max_features')
plt.grid(True)
plt.tight_layout()
plt.show()
```



As observed from the plots, **Random Forest performs best when tree decorrelation is balanced**.
Setting `max_features` too low or too high hurts the model's generalization ability.

- **R²** peaks when `max_features` is around **0.5 to 0.6**, consistent with the lowest RMSE values.
- **R²** drops at both extremes:
  - Using **too few features** (`sqrt`, `log2`, or very small proportions) leads to **underfitting**.
  - Using **all features** (`max_features=1.0`) increases correlation between trees, leading to **overfitting**.

> **Key takeaway**: Carefully tuning `max_features` is critical for achieving the best balance between bias and variance in Random Forest models.

Let's get the minimum RMSE and the corresponding `max_features` value from the result

```
min_rmse = min(rmse_list)
min_rmse_index = rmse_list.index(min_rmse)
best_max_feature = max_features[min_rmse_index]
print(f'Minimum RMSE: {min_rmse:.2f}')
print(f'Best max_features: {best_max_feature}')
# get the maximum R^2 and the corresponding max_features parameter
max_r2 = max(r2_list)
max_r2_index = r2_list.index(max_r2)
best_max_feature_r2 = max_features[max_r2_index]
print(f'Maximum R^2: {max_r2:.2f}')
print(f'Best max_features: {best_max_feature_r2}')
```

```
Minimum RMSE: 3338.02
Best max_features: 0.5
Maximum R^2: 0.96
Best max_features: 0.5
```

Adjusting `max_features` from ‘sqrt’ to 0.5 led to a slight improvement in RMSE, reducing it from 3424 to 3338

7.4 Other Hyperparameters in Random Forest

7.4.1 Why Bagging Uses Unpruned Trees

- Bagging’s main strength lies in **reducing variance**, not bias.
- Deep, unpruned decision trees tend to **overfit** (high variance), but bagging effectively reduces this variance through aggregation.
- Using pruned trees reduces variance but **increases bias** — and since bagging does not correct bias, this would weaken overall performance.

  Therefore, in bagging, it is common to let each tree **grow fully** to preserve low bias and rely on bagging to reduce variance.

7.4.2 Hyperparameters That Control Tree Complexity in Random Forest

Setting	Effect	Applies to
`max_depth=None`	Full trees, low bias, high variance	Bagging, RF

Setting	Effect	Applies to
`max_depth=some int`	Pruned trees, more bias, less variance	Especially helpful in RF
`min_samples_split/leaves`	Prevents small, unreliable branches	Both

7.4.3 Why Random Forest Often Limits Tree Depth

In Random Forest, only a **subset of features** is considered at each split.
As a result, fully growing trees without any depth constraint can cause them to **overfit** to noise within these smaller subsets.

Limiting tree complexity in Random Forest:

- **Prevents deep trees** from chasing noise and overfitting.
- **Improves generalization**, especially in high-dimensional or noisy datasets.
- **Balances** the bias-variance tradeoff more effectively than using full trees.

  Careful tuning of tree depth and other complexity-controlling hyperparameters is critical to maximizing Random Forest performance.

7.4.4 Let's Tune Multiple Hyperparameters Simultaneously Using Cross-Validation

Given the number of hyperparameters involved, we will use `BayesSearchCV` to efficiently perform tuning.
This approach helps reduce computational cost while exploring a wide range of hyperparameter combinations.

```python
# hyperparameter tuning for the random forest regressor

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

# Rename the pipeline step for clarity (recommended)
random_forest_regressor = RandomForestRegressor(
    random_state=42
)

randome_forest_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('rf', random_forest_regressor)  # Renamed from 'bagging' to 'rf'
])

param_space = {
    # Tree structure (control complexity)
    "rf__max_depth": Integer(5, 35),
    "rf__min_samples_split": Integer(2, 20),
    "rf__min_samples_leaf": Integer(1, 10),
    "rf__max_features": Real(0.1, 1.0),

    # Ensemble settings
    "rf__n_estimators": Integer(20, 60),

    # Advanced
    "rf__max_samples": Real(0.1, 1.0),
}

opt = BayesSearchCV(
    randome_forest_pipeline,
    param_space,
    n_iter=50,  # Adjust based on computational resources
    cv=5,
    n_jobs=-1,
    random_state=42
)
opt.fit(X_train, y_train)
```

```
BayesSearchCV(cv=5,
              estimator=Pipeline(steps=[('preprocessor',
                                         ColumnTransformer(remainder='passthrough',
                                                           transformers=[('num',
                                                                          FunctionTransforme
                                                                          ['year',
                                                                           'mileage',
                                                                           'tax',
                                                                           'mpg',
                                                                           'engineSize']),
                                                                         ('cat',
                                                                          OneHotEncoder(hand
                                                                          ['brand',
                                                                           'model',
                                                                           'transmission',
                                                                           'fuelType'])])),
                                        ('rf',
                                         RandomForestRegressor(random_state=42))]),
              n_jobs=-1, random_...
                             'rf__max_features': Real(low=0.1, high=1.0, prior='uniform', tra
                             'rf__max_samples': Real(low=0.1, high=1.0, prior='uniform', tran
                             'rf__min_samples_leaf': Integer(low=1, high=10, prior='uniform'
                             'rf__min_samples_split': Integer(low=2, high=20, prior='uniform
                             'rf__n_estimators': Integer(low=20, high=60, prior='uniform', t
```

```
# get the best parameters
best_params = opt.best_params_
print("Best parameters found: ", best_params)
# get the best score
best_score = opt.best_score_
print("Best score found: ", best_score)
# get the best estimator
best_estimator = opt.best_estimator_
print("Best estimator found: ", best_estimator)
```

```
Best parameters found:  OrderedDict({'rf__max_depth': 28, 'rf__max_features': 0.581672949922
Best score found:  0.9583182725211966
Best estimator found:  Pipeline(steps=[('preprocessor',
                 ColumnTransformer(remainder='passthrough',
                                   transformers=[('num', FunctionTransformer(),
                                                  ['year', 'mileage', 'tax',
                                                   'mpg', 'engineSize']),
                                                 ('cat',
                                                  OneHotEncoder(handle_unknown='ignore'),
                                                  ['brand', 'model',
                                                   'transmission',
                                                   'fuelType'])])),
                ('rf',
                 RandomForestRegressor(max_depth=28,
                                       max_features=0.5816729499221629,
                                       max_samples=1.0, n_estimators=60,
                                       random_state=42))])
```

```
# make predictions on the test data
y_pred = best_estimator.predict(X_test)
# calculate the RMSE and R^2 score
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f'RMSE: {rmse:.2f}')
print(f'R^2: {r2:.2f}')
# calculate the training rmse and r^2 score
y_train_pred = opt.predict(X_train)
train_rmse = root_mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)
print(f'Training RMSE: {train_rmse:.2f}')
print(f'Training R^2: {train_r2:.2f}')
```

```
RMSE: 3278.42
R^2: 0.96
Training RMSE: 1220.88
Training R^2: 0.99
```

As observed in the results, RMSE was further reduced after simultaneously tuning multiple hyperparameters compared to only tuning `max_features` (from 3338 to 3278).
However, due to the small and simple nature of the dataset, the performance improvement is relatively marginal.
On larger and more complex datasets, the performance gains from comprehensive hyperparameter tuning would likely be much more substantial.

7.5 Feature Importance in Random Forest

Random Forest provides a natural way to estimate **feature importance**.

Each time a feature is used to split a node, it contributes to reducing impurity (such as Gini impurity for classification or variance for regression).
By averaging these contributions over all trees in the forest, we can rank features by how important they are to the model's predictive performance.

7.5.1 How Feature Importance Is Calculated

- A feature's importance is based on the **total reduction of the criterion** (e.g., variance for regression) it brings across all splits it is used in.
- Features that result in larger reductions in impurity are assigned **higher importance scores**.
- The importance scores are **normalized** so that they sum to 1 across all features.

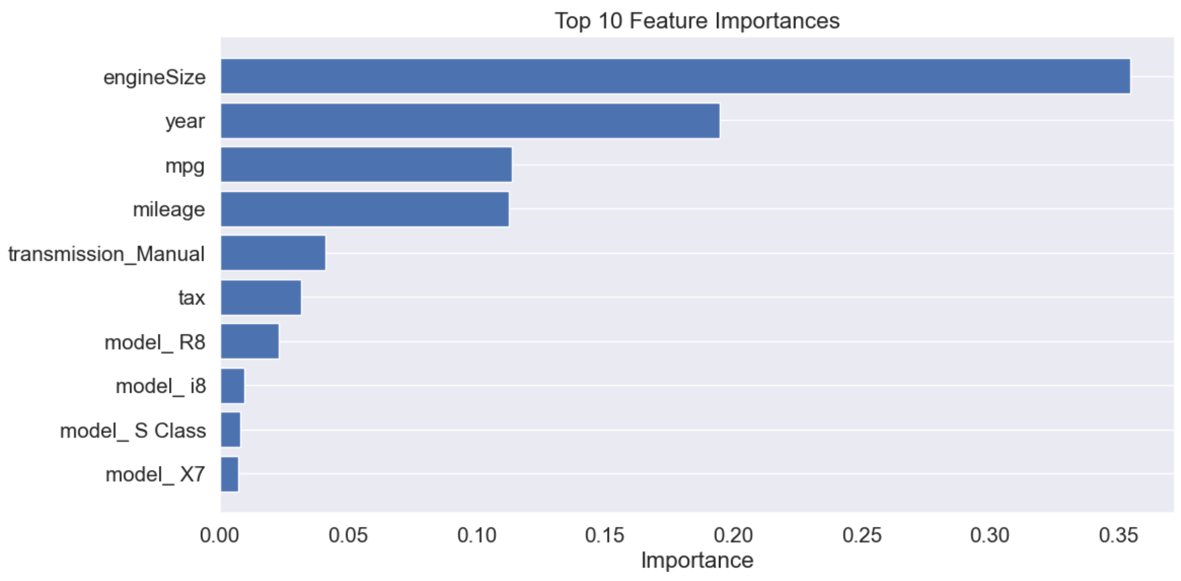
7.5.2 Accessing Feature Importances

After fitting a Random Forest model, feature importances can be accessed through the attribute:

```
model.feature_importances_
```

```
# get numerical_feature and categorical_feature from the pipeline
num_features = numerical_feature
cat_transformer = opt.best_estimator_.named_steps['preprocessor'].named_transformers_['cat']
cat_features = cat_transformer.get_feature_names_out(categorical_feature)

# concatenate all feature names
feature_names = np.concatenate([num_features, cat_features])
```

```
# output feature importances

importances = opt.best_estimator_.named_steps['rf'].feature_importances_
feature_importances = pd.DataFrame(importances, index=feature_names, columns=['importance'])

# select top 10 features
top_10 = feature_importances.head(10)

# plot the top 10 feature importances
plt.figure(figsize=(12, 6))
plt.barh(top_10.index[::-1], top_10['importance'][::-1])  # reverse for top-to-bottom order
plt.xlabel('Importance')
plt.title('Top 10 Feature Importances')
plt.grid(axis='x')
plt.tight_layout()
plt.show()
```



7.6 In Summary

Random Forest is a special case of bagging.
The `n_estimators` and `oob_score` hyperparameters function similarly in both methods, helping to aggregate multiple decision trees into a strong ensemble.

In this notebook, we focused on the key differences between **Random Forest** and **standard bagging**.
Random Forest generally outperforms bagging by introducing an additional layer of randomness:
at each split, only a **random subset of features** is considered.
This strategy **decorrelates** the individual trees, **increases diversity** within the ensemble, and **further reduces variance**, leading to stronger generalization performance.

> **Key takeaway**: Bagging reduces variance by aggregating independent models, while Random Forest improves further by strategically injecting feature-level randomness to strengthen ensemble diversity.

7.7 Next Step

In the next section, we will explore **Boosting methods**,
where models are built **sequentially**, each one focusing on correcting the errors made by the previous models.

Boosting shifts the focus from reducing variance to **reducing bias**, offering another powerful strategy for improving model performance.

8 Adaptive Boosting

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

After learning how Bagging and Random Forest reduce variance by aggregating many trees, we now turn to a different strategy: **Boosting**.

Boosting builds trees **sequentially**, with each new tree focusing on correcting the mistakes made by the previous ones.

Read section 8.2.3 of the book before using these notes.

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

```python
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 root_mean_squared_error,r2_score,roc_curve,auc,precision_recall_
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.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer
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
```

8.1 What is AdaBoost?

AdaBoost stands for **Adaptive Boosting**.
It was one of the first boosting algorithms developed.

The core idea behind AdaBoost:

- Train a **weak learner** (usually a shallow decision tree) on the original data.
- Increase the weights of examples that the learner misclassified.
- Train the next learner on this updated, reweighted data.
- Repeat this process, focusing more and more on hard-to-predict examples.

The final prediction is a **weighted combination** of all the weak learners.

8.2 AdaBoost Intuition

- Easy-to-classify points are **de-emphasized**.
- Hard-to-classify points are **emphasized**.
- Each learner **adapts** to the mistakes made by previous learners — hence "adaptive" boosting.
- **Better-performing learners** are given **higher weight** in the final prediction.

  Over time, the model becomes better at handling difficult cases.

8.3 How AdaBoost Works (High-Level Steps)

1. Initialize equal weights for all training examples.
2. Train a weak learner (e.g., decision stump).
3. Evaluate its performance:
   - Increase weights for misclassified points.
   - Decrease weights for correctly classified points.
4. Train the next learner using the updated weights.
5. Repeat for a set number of learners (`n_estimators`).
6. Combine all learners into a final weighted model.

8.4 Key Hyperparameters in AdaBoost

Hyperparameter	Meaning	Typical Values
`n_estimators`	Number of weak learners	50–500
`learning_rate`	Shrinks each learner’s contribution	0.01–1.0
`estimator`	Type of weak learner (default: decision stump)	Shallow trees

- **Lowering `learning_rate`** typically requires **more estimators** but improves generalization.

8.5 AdaBoost for Regression

We will revisit the car dataset we used earlier and evaluate how AdaBoost performs compared to a single decision tree and a bagging ensemble

```
# 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

```
print(car.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7632 entries, 0 to 7631
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   brand         7632 non-null   object
 1   model         7632 non-null   object
 2   year          7632 non-null   int64
 3   transmission  7632 non-null   object
 4   mileage       7632 non-null   int64
 5   fuelType      7632 non-null   object
 6   tax           7632 non-null   int64
```

 7   mpg   7632 non-null   float64
 8   engineSize   7632 non-null   float64
 9   price   7632 non-null   int64
dtypes: float64(2), int64(4), object(4)
memory usage: 596.4+ KB
None
```

```
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=Tr
X_test_final = pd.concat([X_test_encoded_df, X_test[numerical_feature].reset_index(drop=True)
```

8.5.1 Let's build a adaboost regressor model with default settings

```
# build a adaboost regressor model with default parameters
adaboost_regressor = AdaBoostRegressor(random_state=0)

# fit the model
adaboost_regressor.fit(X_train_final, y_train)
```
```

```python
# predict the test set
y_pred = adaboost_regressor.predict(X_test_final)

# calculate the mean squared error
rmse = root_mean_squared_error(y_test, y_pred)
print(f'RMSE: {rmse:.2f}')
print(f'R2 Score: {r2_score(y_test, y_pred)}')


# calculate the RMSE and R^2 score for the training data
y_pred_train = adaboost_regressor.predict(X_train_final)
rmse_train = root_mean_squared_error(y_train, y_pred_train)
r2_train = r2_score(y_train, y_pred_train)
print(f'Adaboost Train RMSE: {rmse_train:.2f}')
print(f'Adaboost Train R^2: {r2_train:.2f}')

# calculate the test score
```

```
RMSE: 10239.74
R2 Score: 0.6426025126917443
Adaboost Train RMSE: 10081.26
Adaboost Train R^2: 0.62
```

8.5.1.1 Why AdaBoost Perform Worse Here

- **Default base estimator is very weak**:
  By default, AdaBoost uses **Decision Stumps** (`DecisionTreeRegressor(max_depth=1)`), which are extremely shallow and tend to **underfit** the data badly.
- **Learning rate (`learning_rate=1.0`) is too aggressive**:
  When using very weak learners, a high learning rate can cause the boosting process to **fail to properly build up model strength**, leading to poor performance.
- **Dataset characteristics**:
  This dataset is **small** and **not very noisy**, using very shallow trees combined with a high learning rate can cause **severe underfitting**.

What should we do next to reduce bias

- Use deeper Trees as Base Learners
- Tune Learning Rate
- Increase the Number of Estimators

8.5.2 Impact of Tree Depth

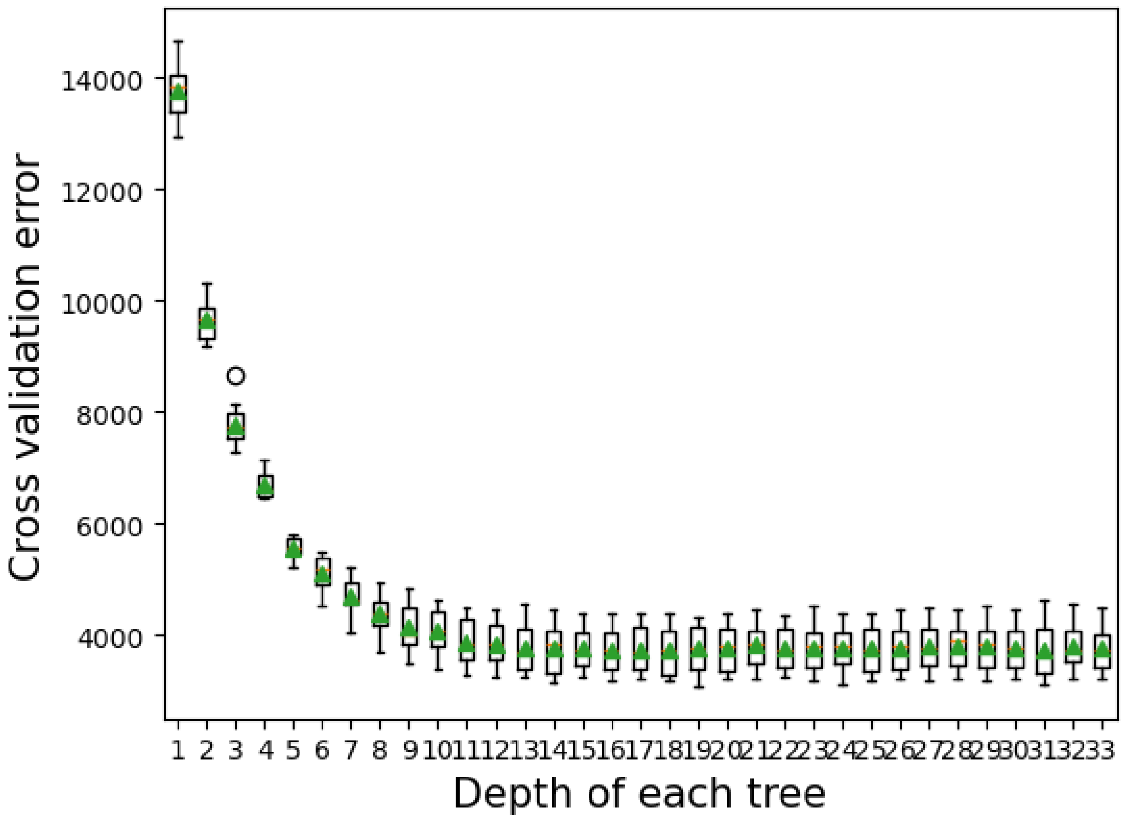
By default, AdaBoost uses shallow decision stumps (`max_depth=1`) as weak learners.
However, slightly increasing the tree depth can make each learner more expressive,
helping the ensemble capture more complex patterns in the data.

This often leads to a **reduction in cross-validation RMSE** and improved model performance,
especially when the underlying relationships in the data are non-linear.

From our previous exploration, we found that the fully grown decision tree has a depth of 34 on this dataset.
In this section, we'll experiment with limiting the tree depth and observe how it affects the model's RMSE.
The goal is to find a depth that balances **bias and variance**, leading to better generalization.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,34):
        # define base model
        base = DecisionTreeRegressor(max_depth=i)
        # define ensemble model
        models[str(i)] = AdaBoostRegressor(estimator=base,n_estimators=100, learning_rate=0.
    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_train_final, y_train)
    # 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 13770.872 (518.361)
>2 9673.586 (398.116)
>3 7783.875 (393.200)
>4 6686.293 (253.234)
>5 5575.918 (176.859)
>6 5106.235 (342.995)
>7 4695.491 (353.584)
>8 4395.372 (366.340)
>9 4143.296 (422.938)
>10 4064.871 (414.840)
>11 3880.994 (433.105)
>12 3831.714 (396.971)
>13 3773.891 (437.190)
>14 3771.442 (425.043)
>15 3769.082 (388.875)
>16 3741.356 (394.563)
>17 3740.153 (415.062)
>18 3721.954 (444.760)
>19 3765.976 (425.594)
>20 3777.496 (425.025)
>21 3827.491 (434.580)
>22 3761.119 (409.760)
>23 3773.776 (429.259)
>24 3763.150 (408.907)
>25 3763.396 (417.512)
>26 3782.507 (405.136)
>27 3791.885 (446.067)
>28 3812.705 (406.567)
>29 3792.121 (445.264)
>30 3780.123 (429.651)
>31 3739.596 (472.987)
>32 3797.480 (426.929)
>33 3754.727 (417.368)
```



As shown in the plot, very shallow trees (e.g., `max_depth=1` to `3`) result in high cross-validation error due to underfitting.
As tree depth increases, the model becomes more expressive, and the error drops sharply up to around depth 10.

Beyond this point, deeper trees offer diminishing returns, and performance stabilizes.
> This suggests that **slightly deeper trees (e.g., depth 5–10)** strike a good balance between model complexity and generalization in AdaBoost.

8.5.3 Impact of Learning Rate

In boosting algorithms such as **AdaBoost** or **Gradient Boosting**, the `learning_rate` controls **how much each new tree contributes** to the overall model.

Each new tree makes a correction to the current prediction, and the learning rate scales **how aggressively** that correction is applied.

- A **high learning rate** takes **large correction steps** — fast learning, but higher risk of overshooting or overfitting.

- A **low learning rate** takes **small correction steps** — more stable, but may underfit unless paired with enough trees.

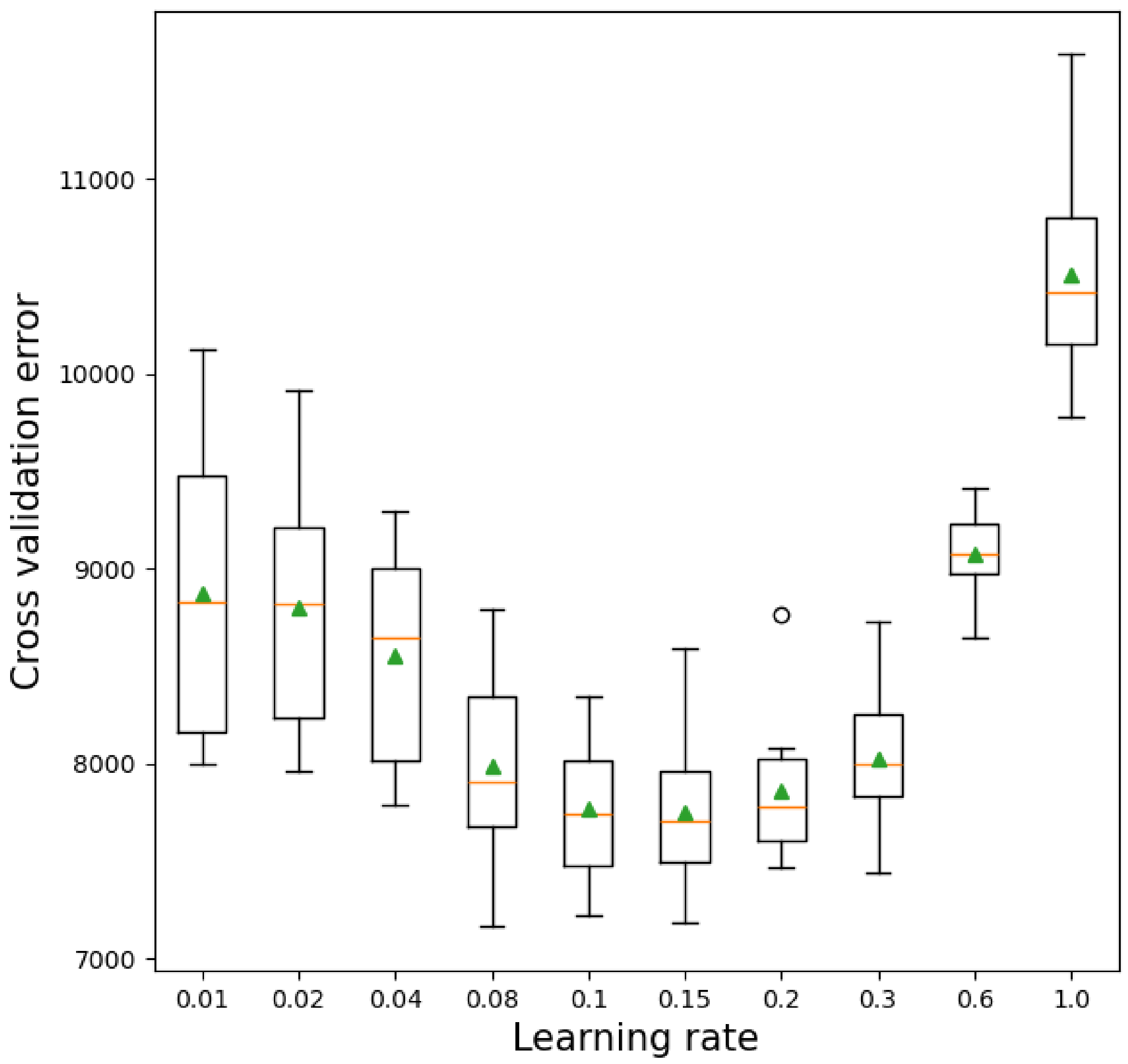
8.5.3.1 Effect on Performance

Learning Rate	Behavior	Risk
Very Small (e.g., 0.01)	Learns slowly, needs many trees	Underfitting if not enough trees
Moderate (e.g., 0.1–0.2)	Balanced correction, stable learning	Often optimal
Large (e.g., 0.5–1.0)	Learns quickly, may overshoot	Overfitting or unstable learning

> **Key takeaway**: Small learning rates usually generalize better — especially when combined with more estimators.

```
def get_models():
    models = dict()
    learning_rates = [0.01, 0.02, 0.04, 0.08, 0.1, 0.15, 0.2, 0.3, 0.6, 1.0]
    for i in range(len(learning_rates)):
        key = learning_rates[i]
        models[key] = AdaBoostRegressor(learning_rate=learning_rates[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_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_train_final, y_train)
    # 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);
```

```
>0.01 8877.6 (725.1)
>0.02 8797.2 (656.7)
>0.04 8554.4 (540.3)
>0.08 7988.1 (479.9)
>0.1 7763.7 (345.8)
>0.15 7754.0 (380.3)
>0.2 7862.9 (368.8)
>0.3 8024.6 (345.3)
>0.6 9078.9 (205.6)
>1.0 10508.4 (507.6)
```



The plot shows that moderate learning rates (0.1–0.2) yield the best and most stable model performance, while very small or very large values hurt generalization — likely due to underfitting or overfitting.

8.5.4 Impact of Number of Trees in Boosting

As the number of trees increases in a boosting model, the **prediction bias tends to decrease**, while the **variance may increase**.

This creates a trade-off:

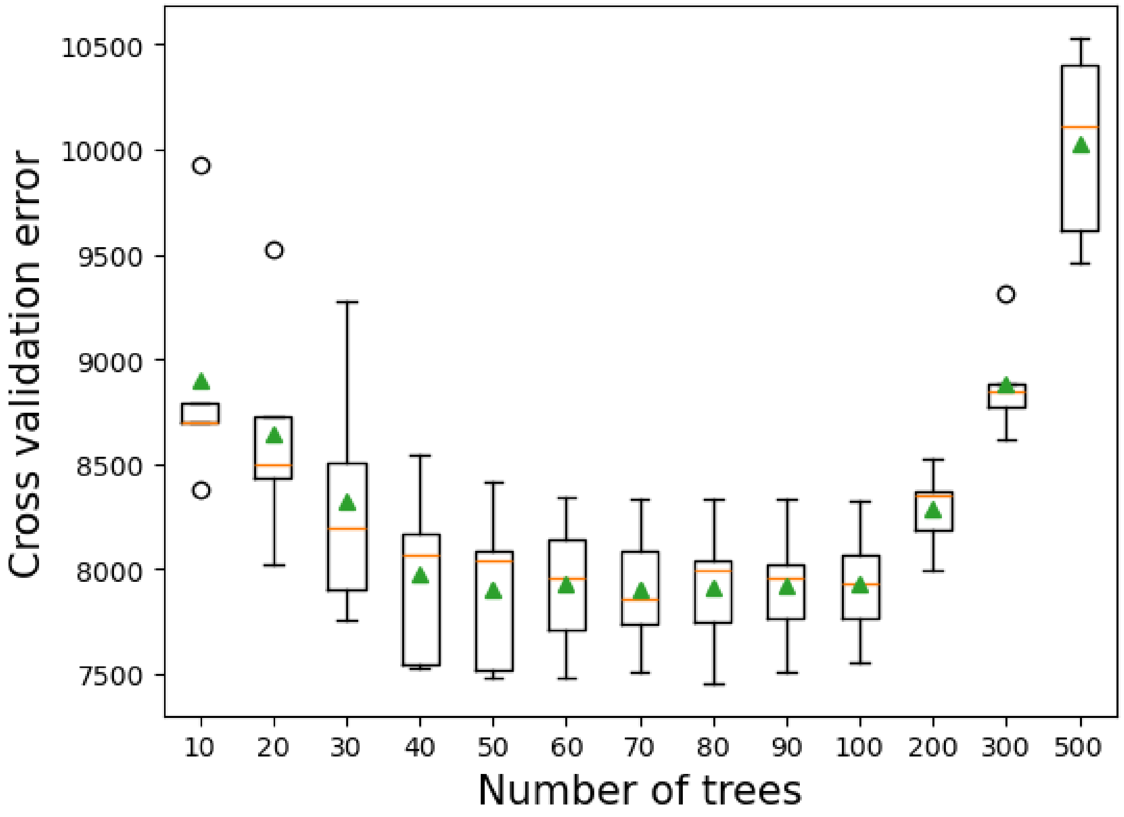
- Too few trees → underfitting (high bias)
- Too many trees → potential overfitting (high variance)

  There is typically an **optimal number of trees** that minimizes the overall prediction error, which can be identified using cross-validation.

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 20, 30, 40,  50, 60, 70,  80, 90, 100,  200, 300, 500]
    for n in n_trees:
        models[str(n)] = AdaBoostRegressor(n_estimators=n,random_state=1, learning_rate=0.1)
    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_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_train_final, y_train)
    # 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 8901.126 (529.620)
>20 8640.382 (495.164)
```

```
>30 8328.349 (539.563)
>40 7972.809 (387.803)
>50 7907.280 (359.779)
>60 7927.212 (305.995)
>70 7904.131 (281.108)
>80 7914.196 (295.777)
>90 7917.841 (274.357)
>100 7927.393 (260.542)
>200 8286.386 (180.913)
>300 8884.444 (230.006)
>500 10024.047 (421.340)
```



With a learning rate of 0.1, the validation error initially decreases, then levels off, and eventually starts to increase — indicating that overfitting is beginning to occur

8.5.5 Tuning Hyperparameters Simultaneously

In the following section, we will use **BayesSearchCV** instead of `GridSearchCV` to efficiently tune multiple hyperparameters at once.
Unlike grid search, which exhaustively evaluates all combinations, Bayesian optimization intelligently explores the hyperparameter space by learning from previous evaluations.
This allows us to find a high-performing model using **fewer iterations and less computation**.

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

# Define the base estimator search space (DecisionTreeRegressor)
base_estimator = DecisionTreeRegressor()

# AdaBoost model (wrapped for BayesSearchCV)
adaboost = AdaBoostRegressor(estimator=base_estimator, random_state=42)

# Search space for tuning
search_space = {
    'estimator__max_depth': Integer(5, 25),
    'n_estimators': Integer(100, 500),
    'learning_rate': Real(0.01, 2.0, prior='log-uniform')
}

# Set up the BayesSearchCV
opt = BayesSearchCV(
    estimator=adaboost,
    search_spaces=search_space,
    n_iter=50,
    scoring='neg_root_mean_squared_error',  # or use 'r2'
    cv=10,
    random_state=42,
    n_jobs=-1,
    verbose=1
)

# Fit on training data
opt.fit(X_train_final, y_train)

# Best parameters
print("Best parameters found:")
print(opt.best_params_)
```

```
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Fitting 10 folds for each of 1 candidates, totalling 10 fits
Best parameters found:
OrderedDict({'estimator__max_depth': 16, 'learning_rate': 1.3460276374020355, 'n_estimators'
```

```python
# Best score
print("Best score (RMSE):")
print(-opt.best_score_)
```

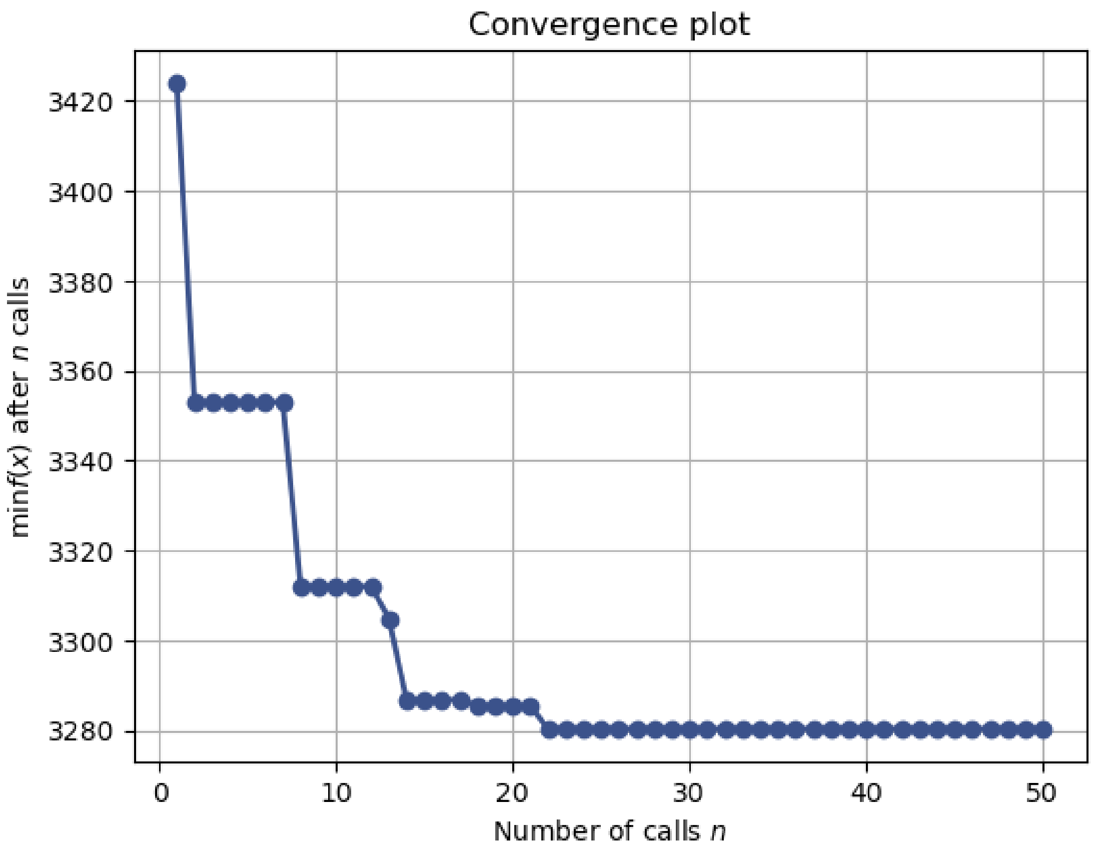
```
Best score (RMSE):
3280.273309604182
```

```python
# evaluate the best model on the test set
best_model = opt.best_estimator_
y_pred_test = best_model.predict(X_test_final)
rmse_test = root_mean_squared_error(y_test, y_pred_test)
print(f'Test RMSE: {rmse_test:.2f}')
print(f'Test R^2: {r2_score(y_test, y_pred_test):.2f}')
```

```
Test RMSE: 3989.52
Test R^2: 0.95
```

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

```python
# import plot_convergence from skopt
from skopt.plots import plot_convergence

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

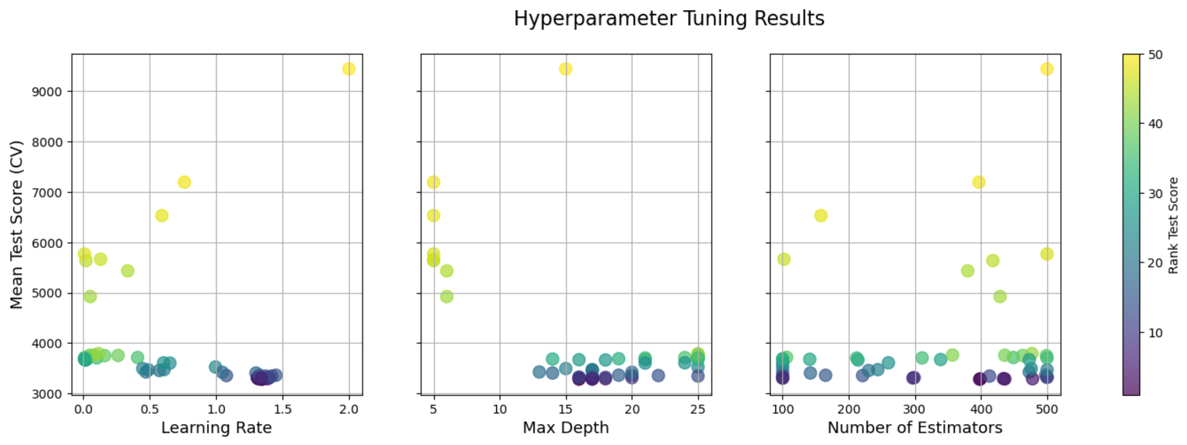


```
# access the full results
results_df = pd.DataFrame(opt.cv_results_)
results_df['mean_test_score'] = -results_df['mean_test_score']
results_df.head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_estimator__max_depth	p
0	27.288841	0.743828	0.243362	0.085644	13	
1	17.667935	0.272929	0.097709	0.037637	22	
2	8.346380	0.144302	0.054507	0.011088	14	
3	30.282255	0.495589	0.178685	0.054628	21	
4	27.381998	0.592005	0.195445	0.069875	21	

8.5.5.1 Analyzing `BayesSearchCV` Results

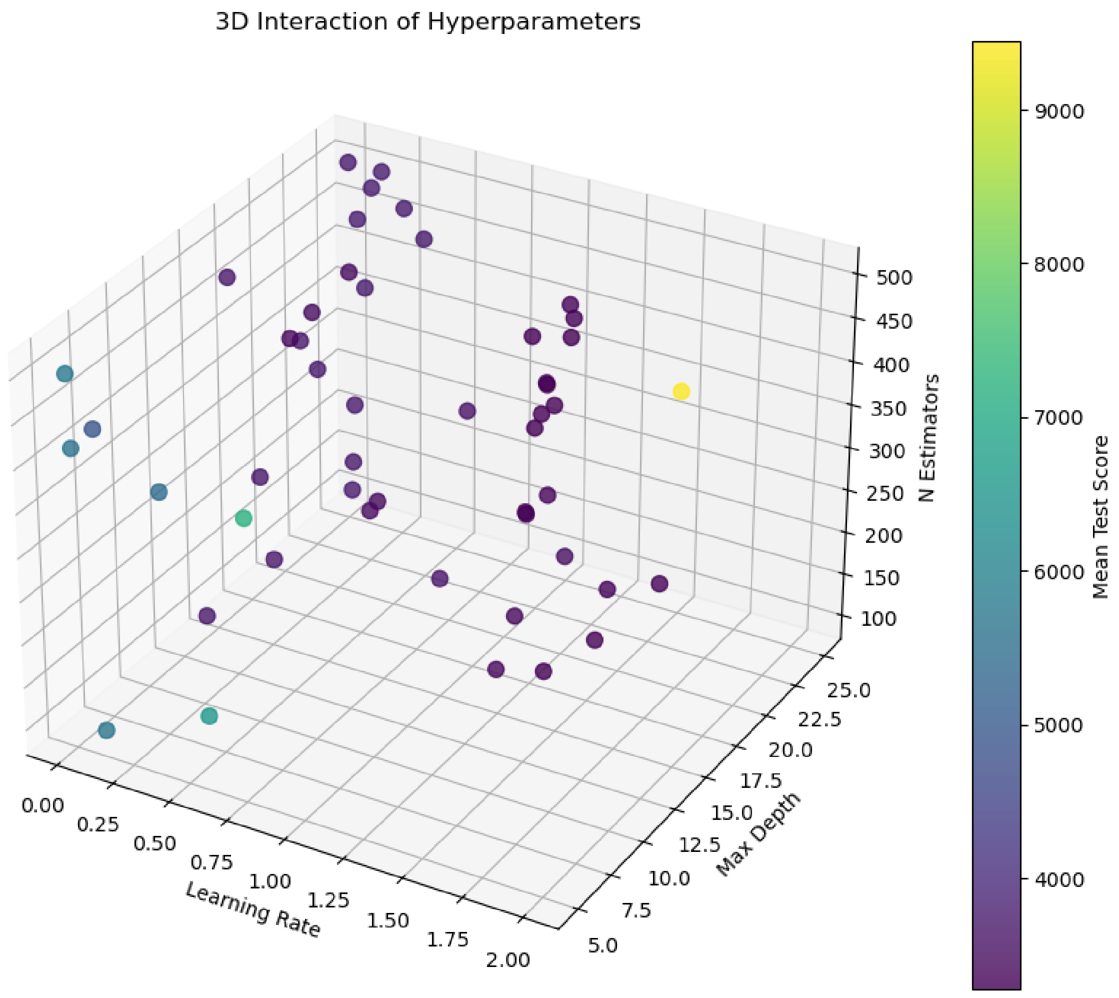
```python
# Create 1x3 subplots
fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

# List of hyperparameters and axis labels
params = ['param_learning_rate', 'param_estimator__max_depth', 'param_n_estimators']
labels = ['Learning Rate', 'Max Depth', 'Number of Estimators']

# Plot each subplot
for ax, param, label in zip(axes, params, labels):
    sc = ax.scatter(
        results_df[param],
        results_df['mean_test_score'],
        c=results_df['rank_test_score'],
        cmap='viridis',
        s=100,
        alpha=0.7
    )
    ax.set_xlabel(label, fontsize=13)
    ax.grid(True)

# Set shared y-axis label and title
axes[0].set_ylabel('Mean Test Score (CV)', fontsize=13)
fig.suptitle('Hyperparameter Tuning Results', fontsize=16)

# Add shared colorbar
cbar = fig.colorbar(sc, ax=axes.ravel().tolist(), label='Rank Test Score');
```



3D scatterplot

- Each point is a combination of the 3 hyperparameters.
- Color indicates performance (darker = better).
- You can rotate the 3D plot in Jupyter interactively!

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

p = ax.scatter(
    results_df['param_learning_rate'],
    results_df['param_estimator__max_depth'],
    results_df['param_n_estimators'],
    c=results_df['mean_test_score'],
    cmap='viridis',
    s=60,
    alpha=0.8
)

ax.set_xlabel('Learning Rate')
ax.set_ylabel('Max Depth')
ax.set_zlabel('N Estimators')
fig.colorbar(p, label='Mean Test Score')
plt.title('3D Interaction of Hyperparameters')
plt.tight_layout()
plt.show()
```



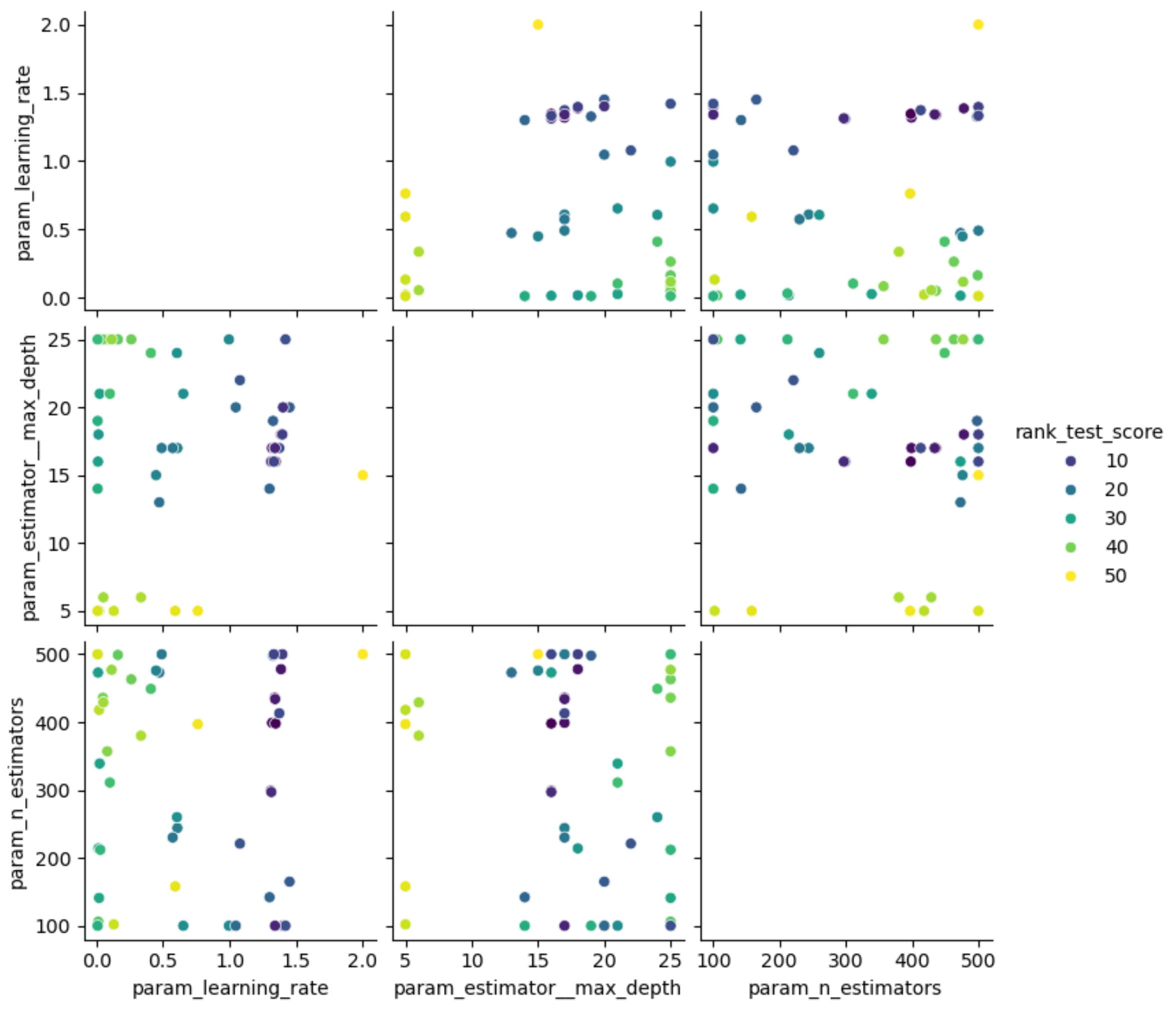
```
sorted_results = results_df.copy()
sorted_results = sorted_results[['param_learning_rate', 'param_estimator__max_depth', 'param_
sorted_results = sorted_results.sort_values(by='rank_test_score')
sorted_results.reset_index(drop=True, inplace=True)
sorted_results[:10] # Display the top 10 results
```

	param_learning_rate	param_estimator__max_depth	param_n_estimators	mean_test_score	std_t
0	1.346028	16	398	3280.273310	368.
1	1.339271	17	436	3285.379694	347.
2	1.318424	17	399	3286.757311	349.
3	1.386113	18	478	3289.156528	363.
4	1.342019	17	434	3291.786531	354.

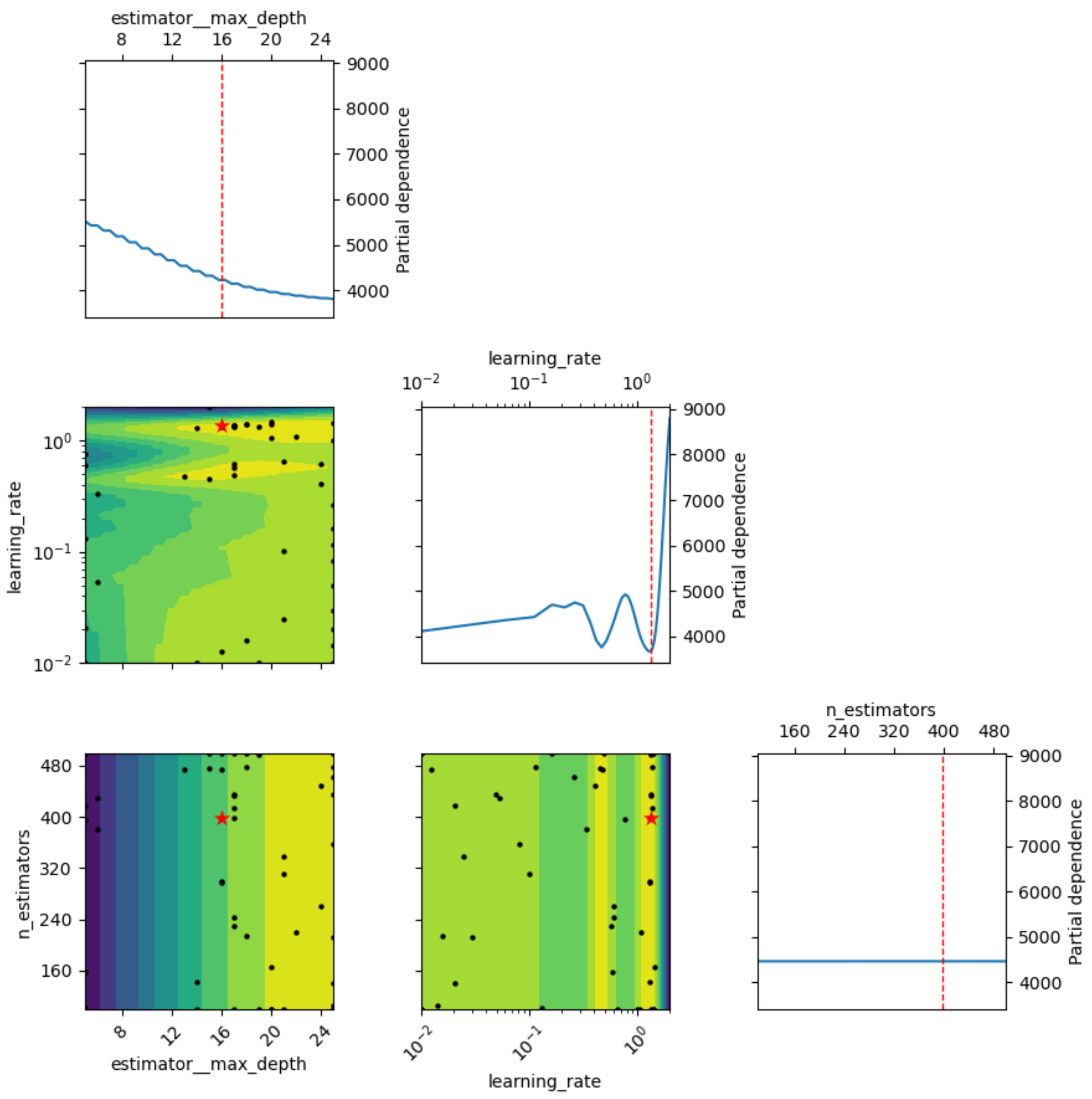
	param__learning__rate	param__estimator___max__depth	param__n__estimators	mean__test__score	std__t
5	1.340962	17	100	3300.807523	343.
6	1.313334	16	297	3304.849191	342.
7	1.401063	20	100	3309.832144	341.
8	1.309088	16	299	3312.005185	332.
9	1.396356	18	500	3317.350680	334.

Let's analyze radeoffs/interactions

```
sns.pairplot(
    results_df,
    vars=[
        'param_learning_rate',
        'param_estimator__max_depth',
        'param_n_estimators'
    ],
    hue='rank_test_score',
    palette='viridis'
);
```



```python
from skopt.plots import plot_objective

plot_objective(opt.optimizer_results_[0], dimensions=None, size = 3)
plt.tight_layout()
plt.show()
```



8.5.6 Using Optuna for Hyperparameter Tuning

```
pip install optuna
```

```
Collecting optuna
  Downloading optuna-4.3.0-py3-none-any.whl.metadata (17 kB)
```

```
Collecting alembic>=1.5.0 (from optuna)
  Downloading alembic-1.15.2-py3-none-any.whl.metadata (7.3 kB)
Collecting colorlog (from optuna)
  Downloading colorlog-6.9.0-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: numpy in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pa
Requirement already satisfied: packaging>=20.0 in c:\users\lsi8012\appdata\roaming\python\py
Requirement already satisfied: sqlalchemy>=1.4.2 in c:\users\lsi8012\appdata\local\anaconda3\
Requirement already satisfied: tqdm in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pac
Requirement already satisfied: PyYAML in c:\users\lsi8012\appdata\local\anaconda3\lib\site-p
Collecting Mako (from alembic>=1.5.0->optuna)
  Downloading mako-1.3.10-py3-none-any.whl.metadata (2.9 kB)
Collecting typing-extensions>=4.12 (from alembic>=1.5.0->optuna)
  Downloading typing_extensions-4.13.2-py3-none-any.whl.metadata (3.0 kB)
Requirement already satisfied: greenlet!=0.4.17 in c:\users\lsi8012\appdata\local\anaconda3\
Requirement already satisfied: colorama in c:\users\lsi8012\appdata\roaming\python\python312\
Requirement already satisfied: MarkupSafe>=0.9.2 in c:\users\lsi8012\appdata\local\anaconda3\
Downloading optuna-4.3.0-py3-none-any.whl (386 kB)
   ---------------------------------------- 0.0/386.6 kB ? eta -:--:--
   ------------------ --------------------- 174.1/386.6 kB 3.5 MB/s eta 0:00:01
   ---------------------------------------- 386.6/386.6 kB 4.8 MB/s eta 0:00:00
Downloading alembic-1.15.2-py3-none-any.whl (231 kB)
   ---------------------------------------- 0.0/231.9 kB ? eta -:--:--
   ---------------------------------------- 231.9/231.9 kB 7.2 MB/s eta 0:00:00
Downloading colorlog-6.9.0-py3-none-any.whl (11 kB)
Downloading typing_extensions-4.13.2-py3-none-any.whl (45 kB)
   ---------------------------------------- 0.0/45.8 kB ? eta -:--:--
   ---------------------------------------- 45.8/45.8 kB 2.4 MB/s eta 0:00:00
Downloading mako-1.3.10-py3-none-any.whl (78 kB)
   ---------------------------------------- 0.0/78.5 kB ? eta -:--:--
   ---------------------------------------- 78.5/78.5 kB 4.6 MB/s eta 0:00:00
Installing collected packages: typing-extensions, Mako, colorlog, alembic, optuna
  Attempting uninstall: typing-extensions
    Found existing installation: typing_extensions 4.11.0
    Uninstalling typing_extensions-4.11.0:
      Successfully uninstalled typing_extensions-4.11.0
Successfully installed Mako-1.3.10 alembic-1.15.2 colorlog-6.9.0 optuna-4.3.0 typing-extensi
Note: you may need to restart the kernel to use updated packages.

ERROR: pip's dependency resolver does not currently take into account all the packages that a
streamlit 1.32.0 requires packaging<24,>=16.8, but you have packaging 24.2 which is incompat
```

Step 1: Import

```
# import optuna
import optuna
```

Step 2: Define the Objective Function

```
def objective(trial):
    # Suggest hyperparameters
    learning_rate = trial.suggest_float("learning_rate", 0.01, 2.0)
    max_depth = trial.suggest_int("max_depth", 5, 25)
    n_estimators = trial.suggest_int("n_estimators", 100, 500)

    # Define model with trial parameters
    base_estimator = DecisionTreeRegressor(max_depth=max_depth)
    model = AdaBoostRegressor(
        estimator=base_estimator,
        learning_rate=learning_rate,
        n_estimators=n_estimators,
        random_state=42
    )

    # Cross-validation score (negative RMSE)
    score = cross_val_score(model, X_train_final, y_train, scoring="neg_root_mean_squared_er
    return -np.mean(score)
```

Step 3: Run the study

```
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=20, timeout=600)  # 50 trials or 10 min
```

```
[I 2025-04-29 18:13:37,366] A new study created in memory with name: no-name-8c1fd49f-a877-4
[I 2025-04-29 18:13:55,695] Trial 0 finished with value: 3604.108244895723 and parameters: {
[I 2025-04-29 18:14:39,369] Trial 1 finished with value: 3418.989299711666 and parameters: {
[I 2025-04-29 18:15:01,876] Trial 2 finished with value: 6675.343723783454 and parameters: {
[I 2025-04-29 18:15:51,382] Trial 3 finished with value: 3534.3761450527854 and parameters:
[I 2025-04-29 18:16:17,255] Trial 4 finished with value: 4146.601971284103 and parameters: {
[I 2025-04-29 18:16:51,329] Trial 5 finished with value: 3373.6473210617305 and parameters:
[I 2025-04-29 18:17:22,661] Trial 6 finished with value: 4645.349739417665 and parameters: {
[I 2025-04-29 18:17:48,173] Trial 7 finished with value: 3783.098791065398 and parameters: {
[I 2025-04-29 18:18:07,203] Trial 8 finished with value: 3642.6233815254373 and parameters:
[I 2025-04-29 18:19:28,631] Trial 9 finished with value: 3815.355472103091 and parameters: {
[I 2025-04-29 18:19:52,259] Trial 10 finished with value: 3460.264767659689 and parameters:
```

```
[I 2025-04-29 18:20:24,557] Trial 11 finished with value: 3354.0174995460206 and parameters:
[I 2025-04-29 18:20:51,602] Trial 12 finished with value: 3424.5702876072514 and parameters:
[I 2025-04-29 18:21:42,335] Trial 13 finished with value: 3428.9633976083956 and parameters:
[I 2025-04-29 18:22:07,861] Trial 14 finished with value: 3363.7031695680153 and parameters:
[I 2025-04-29 18:22:38,820] Trial 15 finished with value: 3467.0742139036493 and parameters:
[I 2025-04-29 18:23:03,761] Trial 16 finished with value: 4339.523251732014 and parameters:
[I 2025-04-29 18:23:32,216] Trial 17 finished with value: 3429.3383627414196 and parameters:
[I 2025-04-29 18:23:46,880] Trial 18 finished with value: 3432.187531277779 and parameters:
```

Step 4: Review Best Result

```python
print("Best RMSE:", study.best_value)
print("Best hyperparameters:", study.best_params)
```

```
Best RMSE: 3354.0174995460206
Best hyperparameters: {'learning_rate': 1.4283231452601248, 'max_depth': 17, 'n_estimators':
```

```python
# make a prediction using the best hyperparameters
best_params = study.best_params
base_estimator = DecisionTreeRegressor(max_depth=best_params['max_depth'])
model = AdaBoostRegressor(
    estimator=base_estimator,
    learning_rate=best_params['learning_rate'],
    n_estimators=best_params['n_estimators'],
    random_state=42
)
# fit the model
model.fit(X_train_final, y_train)
# predict the test set
y_pred = model.predict(X_test_final)
# calculate the mean squared error
rmse = root_mean_squared_error(y_test, y_pred)
print(f'RMSE: {rmse:.2f}')
print(f'R2 Score: {r2_score(y_test, y_pred)}')
```

```
RMSE: 3524.38
R2 Score: 0.9576611696103872
```

Step 5: Visualize the Search

```
optuna.visualization.plot_optimization_history(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
optuna.visualization.plot_param_importances(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
optuna.visualization.plot_parallel_coordinate(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

Insights:

- Best Hyperparameter Region:
  - `learning_rate`: ~0.2–0.5
  - `max_depth`: ~7–10
  - `n_estimators`: ~300–450
- Trade-offs:
  - Increasing `n_estimators` improves performance but increases computation time.
  - Lower `learning_rate` values require more estimators to achieve good performance.
- Next Steps:
  - Focus on fine-tuning within the identified ranges for `learning_rate`, `max_depth`, and `n_estimators`.
  - Use these insights to narrow the search space for further optimization.

9 Gradient Boosting

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

Gradient Boosting is a **boosting technique** that builds an additive model in a forward stage-wise manner. Unlike AdaBoost, which adjusts weights on training instances, Gradient Boosting fits new models to the **residual errors** made by prior models using the gradient of a specified loss function.

At each stage, a new weak learner is trained to minimize the loss function by correcting the errors of the current ensemble.

9.1 Gradient Boosting Intuition

Gradient Boosting can be understood as **functional gradient descent**:

- We start with an initial prediction (e.g., the mean of the targets).
- At each iteration, we fit a new model to the **negative gradient** of the loss function with respect to the current predictions.
- This negative gradient plays a similar role to residuals in squared loss regression—it points in the direction that most reduces the loss.
- The new model's predictions are then added to the current model, scaled by a learning rate.

By sequentially adding models that reduce the remaining error, the ensemble gradually improves.

9.2 How Gradient Boosting Works (Regression Example)

1. **Initialize** the model with a constant prediction:

$$\hat{f}^{(0)}(x) = \arg\min_c \sum_{i=1}^{n} L(y_i, c)$$

2. **For** $m = 1$ to M (number of boosting rounds):

- Compute the **negative gradient** (pseudo-residuals):

$$r_i^{(m)} = -\left[\frac{\partial L(y_i, \hat{f}(x_i))}{\partial \hat{f}(x_i)}\right]_{\hat{f}(x)=\hat{f}^{(m-1)}(x)}$$

- Fit a **base learner** $h^{(m)}(x)$ to the residuals $r_i^{(m)}$.
- Determine the **optimal step size** (line search):

$$\gamma^{(m)} = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, \hat{f}^{(m-1)}(x_i) + \gamma \cdot h^{(m)}(x_i)\right)$$

- **Update the model**:

$$\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \eta \cdot \gamma^{(m)} h^{(m)}(x)$$

where η is the learning rate.

3. **Final prediction**:

$$\hat{f}^{(M)}(x)$$

```
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 root_mean_squared_error, mean_squared_error,r2_score,roc_curve,au
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.preprocessing import OneHotEncoder, FunctionTransformer
import itertools as it
import time as time

import optuna
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
```

9.3 Gradient Boosting in Scikit-Learn

Scikit-learn offers a standard implementation of Gradient Boosting through two primary estimators:

- `GradientBoostingClassifier` for classification tasks
- `GradientBoostingRegressor` for regression tasks

These estimators build an additive model in a forward stage-wise fashion, allowing for the optimization of arbitrary differentiable loss functions. They are suitable for small to medium-sized datasets and provide flexibility in model tuning.

For larger datasets (typically with `n_samples >= 10,000`), consider using the histogram-based variants:

- `HistGradientBoostingClassifier`
- `HistGradientBoostingRegressor`

9.4 Core Hyperparameters Categories

The primary hyperparameters for `GradientBoostingClassifier` and `GradientBoostingRegressor` can be grouped into the following categories:

1. **Number of Trees** (`n_estimators`)

    - Use **early stopping** (via `n_iter_no_change` and `validation_fraction` in scikit-learn) to avoid overfitting.

    - Start with a large value (e.g., 500–1000) and let early stopping prune unnecessary trees.

2. **Early Stopping**

    - Prevents overfitting by halting training once the validation performance stops improving.

    - Controlled using:
        - `n_iter_no_change`: Number of rounds with no improvement before stopping (e.g., 10).
        - `validation_fraction`: Fraction of training data reserved as internal validation set (e.g., 0.1).
        - `tol`: Minimum improvement to be considered significant (e.g., `1e-4`).

- Set a large n_estimators, and let early stopping determine the optimal number of boosting iterations.

3. **Learning Rate** (learning_rate)
   - Shrinks the contribution of each tree to improve generalization.
   - *Typical range*: 0.01–0.2 (lower values require more trees).
4. **Tree Complexity**
   - max_depth: Depth of individual trees. Start shallow (3–6) to limit overfitting.
   - min_samples_split: Minimum samples required to split a node (e.g., 10–50).
   - min_samples_leaf: Minimum samples required in a leaf node (e.g., 5–20).
5. **Stochastic Gradient Boosting**
   - subsample: Fraction of training data sampled per tree (e.g., 0.5–1.0).
   - max_features: Fraction/absolute number of features used per split (e.g., sqrt(n_features) or 0.8).
6. **Loss Function** (loss)
   - Matches the problem type:
     - Regression: squared_error, absolute_error
     - Classification: log_loss (binary/multinomial deviance)

For a comprehensive list and detailed explanations of all hyperparameters, refer to the official Scikit-learn documentation:

- GradientBoostingClassifier Documentation
- GradientBoostingRegressor Documentation

9.5 Hyperparameter Tuning

Let's reuse the car dataset to evaluate how different hyperparameter settings affect the performance of gradient boosting

```
# 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

```python
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()
```

```python
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(categ

# 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)
```

9.5.1 Individual Hyperparameter Impact Analysis

9.5.1.1 Effect of Number of Trees on Cross-Validation Error

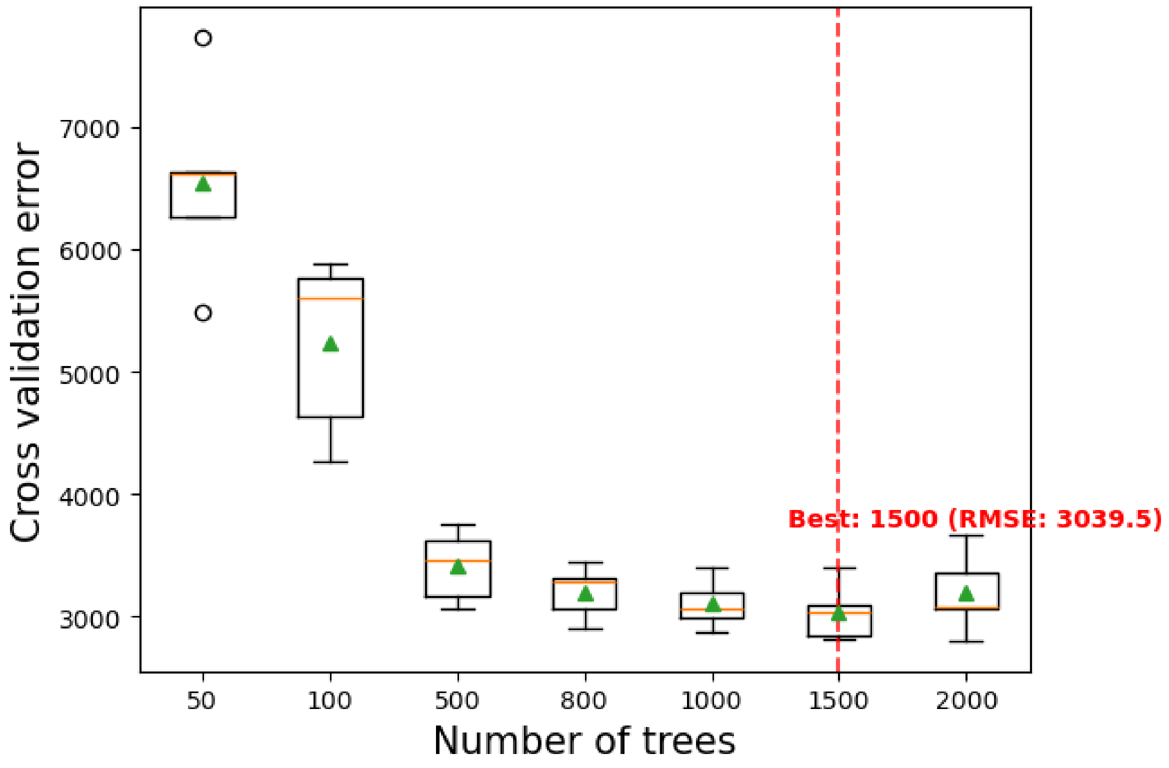
Effect of Number of Trees on Cross-Validation Error In Gradient Boosting, the number of trees (`n_estimators`) controls how many boosting rounds the model performs. Adding more trees can reduce bias and improve training accuracy, but it also increases the risk of overfitting, especially with a high learning rate.

The optimal number of trees is often found by balancing **model complexity** and **generalization performance** using cross-validation.

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [50, 100, 500, 800, 1000, 1500, 2000]
    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_train_final, y_train)
    # 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('Number of trees',fontsize=15);

# get the optimal number of trees
best_index = np.argmin([np.mean(r) for r in results])
best_n_trees = names[best_index]
best_score = np.mean(results[best_index])


# Highlight the best model on the plot
```

```
plt.axvline(x=best_index+1, color='red', linestyle='--', alpha=0.7)
plt.text(best_index + 1 - 0.4, best_score+700,
         f'Best: {best_n_trees} (RMSE: {best_score:.1f})',
         color='red', fontweight='bold')

print(f"Best number of trees: {best_n_trees} with RMSE: {best_score:.3f}")
```

```
>50 6549.576 (722.462)
>100 5232.949 (656.216)
>500 3419.467 (262.753)
>800 3202.489 (194.072)
>1000 3106.002 (184.799)
>1500 3039.520 (210.989)
>2000 3194.874 (293.134)
Best number of trees: 1500 with RMSE: 3039.520
```



9.5.1.2 Early stopping in Gradient Boosting

Why Early Stopping Matters

Specifying a fixed number of trees means deciding in advance how many boosting rounds (i.e., trees) the model will train.

This approach can be inefficient or risky:

- If **too few** trees are used, the model may **underfit**.
- If **too many**, the model may **overfit** or **waste computation**.

That's why **early stopping** is useful — it allows the model to **stop training once performance on a validation set no longer improves**, effectively selecting the optimal number of trees automatically.

How Early Stopping Works

Instead of specifying a fixed number of trees (`n_estimators`), the algorithm monitors performance on a **validation set** and stops adding new trees once the model's improvement has plateaued.

In scikit-learn, early stopping can be enabled using:

- `early_stopping=True`
- `validation_fraction`: The fraction of training data used as a validation set
- `n_iter_no_change`: Number of iterations to wait without improvement before stopping

This approach not only improves generalization but also reduces training time by avoiding unnecessary trees.

```
params = dict(n_estimators=2000, max_depth=5, learning_rate=0.1, random_state=42)

gbm_full = GradientBoostingRegressor(**params)
gbm_early_stopping = GradientBoostingRegressor(
    **params,
    validation_fraction=0.1,
    n_iter_no_change=10,
)

start_time = time.time()
gbm_full.fit(X_train_final, y_train)
training_time_full = time.time() - start_time
n_estimators_full = gbm_full.n_estimators_

start_time = time.time()
```

```
gbm_early_stopping.fit(X_train_final, y_train)
training_time_early_stopping = time.time() - start_time
estimators_early_stopping = gbm_early_stopping.n_estimators_
```

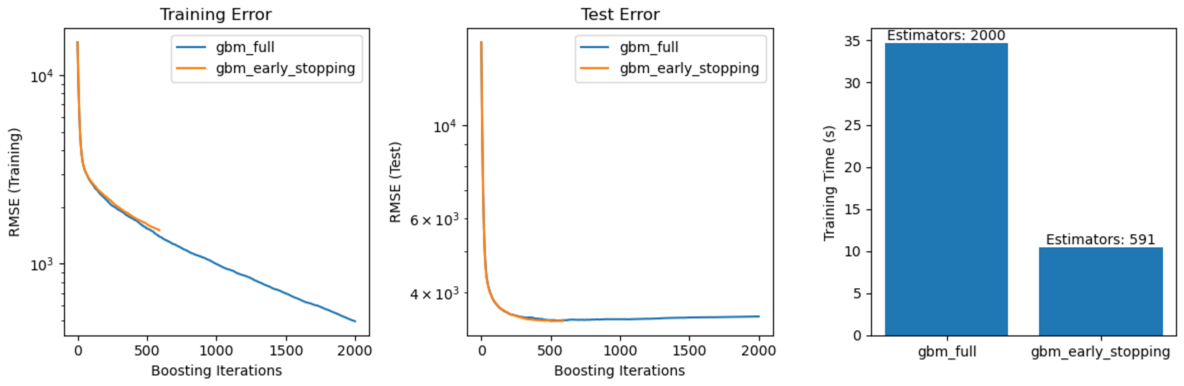
Let's calculate the RMSE on both the training and test datasets for each model, which will be used for later visualization.

```
# import root mean squared error function
from sklearn.metrics import root_mean_squared_error

train_errors_without = []
test_errors_without = []

train_errors_with = []
test_errors_with = []

for i, (train_pred, test_pred) in enumerate(
    zip(
        gbm_full.staged_predict(X_train_final),
        gbm_full.staged_predict(X_test_final),
    )
):
    train_errors_without.append(root_mean_squared_error(y_train, train_pred))
    test_errors_without.append(root_mean_squared_error(y_test, test_pred))

for i, (train_pred, test_pred) in enumerate(
    zip(
        gbm_early_stopping.staged_predict(X_train_final),
        gbm_early_stopping.staged_predict(X_test_final),
    )
):
    train_errors_with.append(root_mean_squared_error(y_train, train_pred))
    test_errors_with.append(root_mean_squared_error(y_test, test_pred))
```

Let's visulize Comparison. It includes three subplots:

1. Plotting training errors of both models over boosting iterations.
2. Plotting test errors of both models over boosting iterations.
3. Creating a bar chart to compare the training times and the number of estimators used by the models with and without early stopping.

```python
fig, axes = plt.subplots(ncols=3, figsize=(12, 4))

axes[0].plot(train_errors_without, label="gbm_full")
axes[0].plot(train_errors_with, label="gbm_early_stopping")
axes[0].set_xlabel("Boosting Iterations")
axes[0].set_ylabel("RMSE (Training)")
axes[0].set_yscale("log")
axes[0].legend()
axes[0].set_title("Training Error")

axes[1].plot(test_errors_without, label="gbm_full")
axes[1].plot(test_errors_with, label="gbm_early_stopping")
axes[1].set_xlabel("Boosting Iterations")
axes[1].set_ylabel("RMSE (Test)")
axes[1].set_yscale("log")
axes[1].legend()
axes[1].set_title("Test Error")

training_times = [training_time_full, training_time_early_stopping]
labels = ["gbm_full", "gbm_early_stopping"]
bars = axes[2].bar(labels, training_times)
axes[2].set_ylabel("Training Time (s)")

for bar, n_estimators in zip(bars, [n_estimators_full, estimators_early_stopping]):
    height = bar.get_height()
    axes[2].text(
        bar.get_x() + bar.get_width() / 2,
        height + 0.001,
        f"Estimators: {n_estimators}",
        ha="center",
        va="bottom",
    )

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



The difference in training error between the `gbm_full` and the `gbm_early_stopping` stems from the fact that
`gbm_early_stopping` sets aside `validation_fraction` of the training data as an internal validation set.

Early stopping is decided based on this internal validation score.

Benefits of Using Early Stopping in Boosting:

- **Preventing Overfitting**
  Early stopping helps avoid overfitting by monitoring the test error.
  When the error stabilizes or starts increasing, training stops — resulting in better generalization to unseen data.
- **Improving Training Efficiency**
  Models with early stopping often require **fewer estimators** while achieving similar accuracy.
  This reduces training time significantly compared to training without early stopping.

9.5.1.3 Effect of Learning Rate on Cross-Validation Error

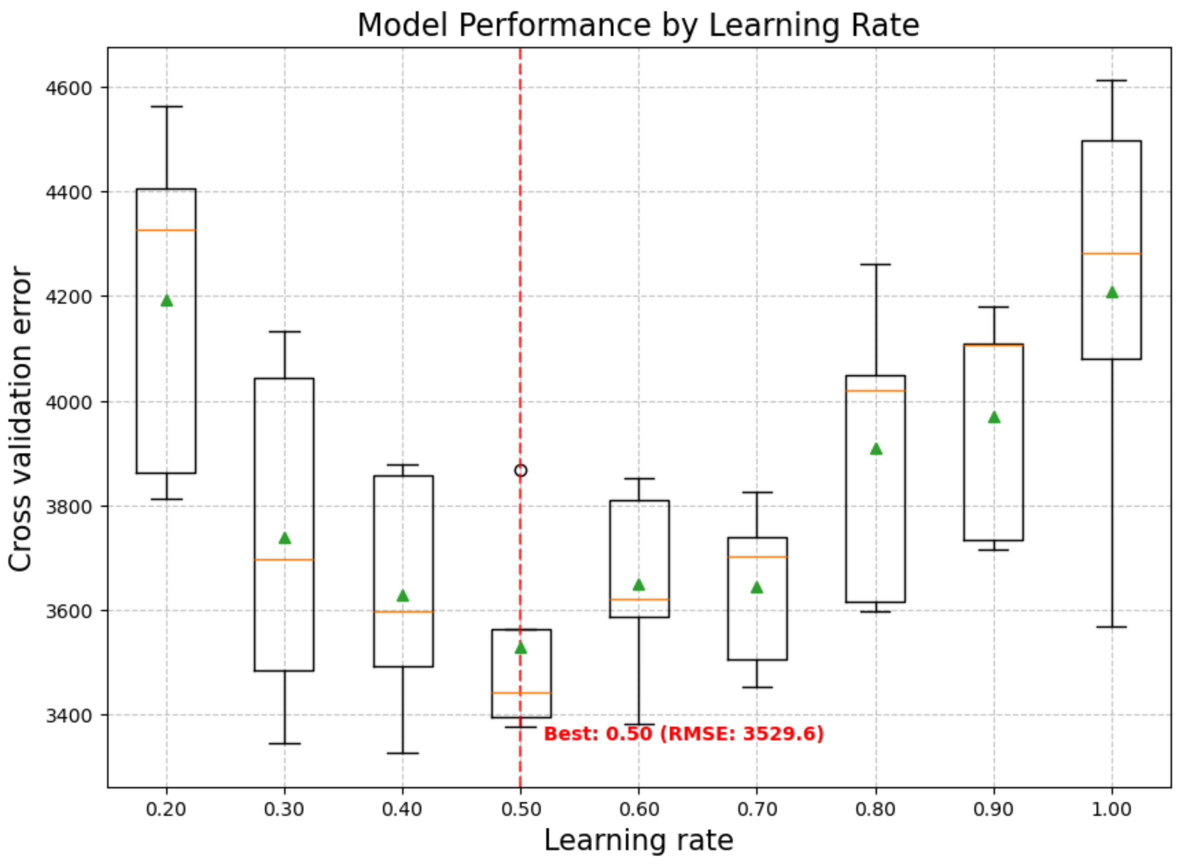
The learning rate (`learning_rate`) determines how much each new tree contributes to the overall model. A **smaller learning rate** results in slower learning and often requires more trees to achieve good performance. A **larger learning rate** speeds up learning but increases the risk of overfitting.

Finding the optimal learning rate involves balancing: - **High learning rate** $\rightarrow$ faster convergence, but higher risk of overfitting
- **Low learning rate** $\rightarrow$ better generalization, but requires more trees and longer training time

Cross-validation helps identify the learning rate that minimizes prediction error while ensuring model stability.

```python
def get_models():
    models = dict()
    # create 9 evenly spaced values between 0.2 and 1.0
    learning_rates = np.linspace(0.2, 1.0, 9)
    for learning_rate in learning_rates:
        # Round to 2 decimal places for clean keys
        lr_rounded = round(learning_rate, 2)
        key = f"{lr_rounded:.2f}"
        models[key] = GradientBoostingRegressor(learning_rate=lr_rounded, random_state=1, los
    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, n
    return scores

# get the models to evaluate
models = get_models()

# evaluate the models and store results
results, names = list(), list()
mean_scores = []  # Track mean scores separately

for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X_train_final, y_train)
    # store the results
    results.append(scores)
    names.append(name)
    # Calculate and store mean score
    mean_score = np.mean(scores)
    mean_scores.append(mean_score)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, mean_score, np.std(scores)))

# plot model performance for comparison
plt.figure(figsize=(10, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error', fontsize=15)
plt.xlabel('Learning rate', fontsize=15)
plt.title('Model Performance by Learning Rate', fontsize=16)
plt.grid(True, linestyle='--', alpha=0.7)

# Find the best model using the saved mean scores
best_index = np.argmin(mean_scores)
best_lr = names[best_index]
best_score = mean_scores[best_index]

# Highlight the best model on the plot
plt.axvline(x=best_index+1, color='red', linestyle='--', alpha=0.7)
plt.text(best_index+1.2, min(mean_scores)*0.95,
         f'Best: {best_lr} (RMSE: {best_score:.1f})',
         color='red', fontweight='bold')

plt.show()

# Print the best model information
print(f"\nBest model: {best_lr} with RMSE: {best_score:.3f}")
```

```
>0.20 4193.7 (301.2)
>0.30 3740.3 (306.3)
>0.40 3630.0 (212.2)
>0.50 3529.6 (181.5)
>0.60 3650.2 (169.0)
>0.70 3644.7 (142.5)
>0.80 3908.9 (260.6)
>0.90 3968.7 (201.1)
>1.00 4208.3 (368.5)
```



```
Best model: 0.50 with RMSE: 3529.563
```

9.5.1.4 Learning Rate and Number of Trees Are Closely Linked

The **learning rate** and **number of trees** (`n_estimators`) are tightly coupled hyperparameters in gradient boosting. Their balance plays a key role in model performance and overfitting control.

- A **lower learning rate** slows the learning process, requiring **more trees** to achieve strong performance.
- A **higher learning rate** speeds up training but may cause the model to **overfit** if not regularized properly.

A high learning rate with too few trees can lead to poor generalization, while a very low learning rate with too many trees may improve accuracy but increase training time significantly.

Best practice: Use a low to moderate learning rate (e.g., `0.01`–`0.1`) combined with **early stopping** to find the optimal number of trees.

9.5.1.5 Effect of Stochastic Gradient Boosting on Cross-Validation Error

Stochastic Gradient Boosting enhances generalization by introducing randomness into the model-building process. Two key hyperparameters that control this are `subsample` and `max_features`, and they operate on **different dimensions** of the data:

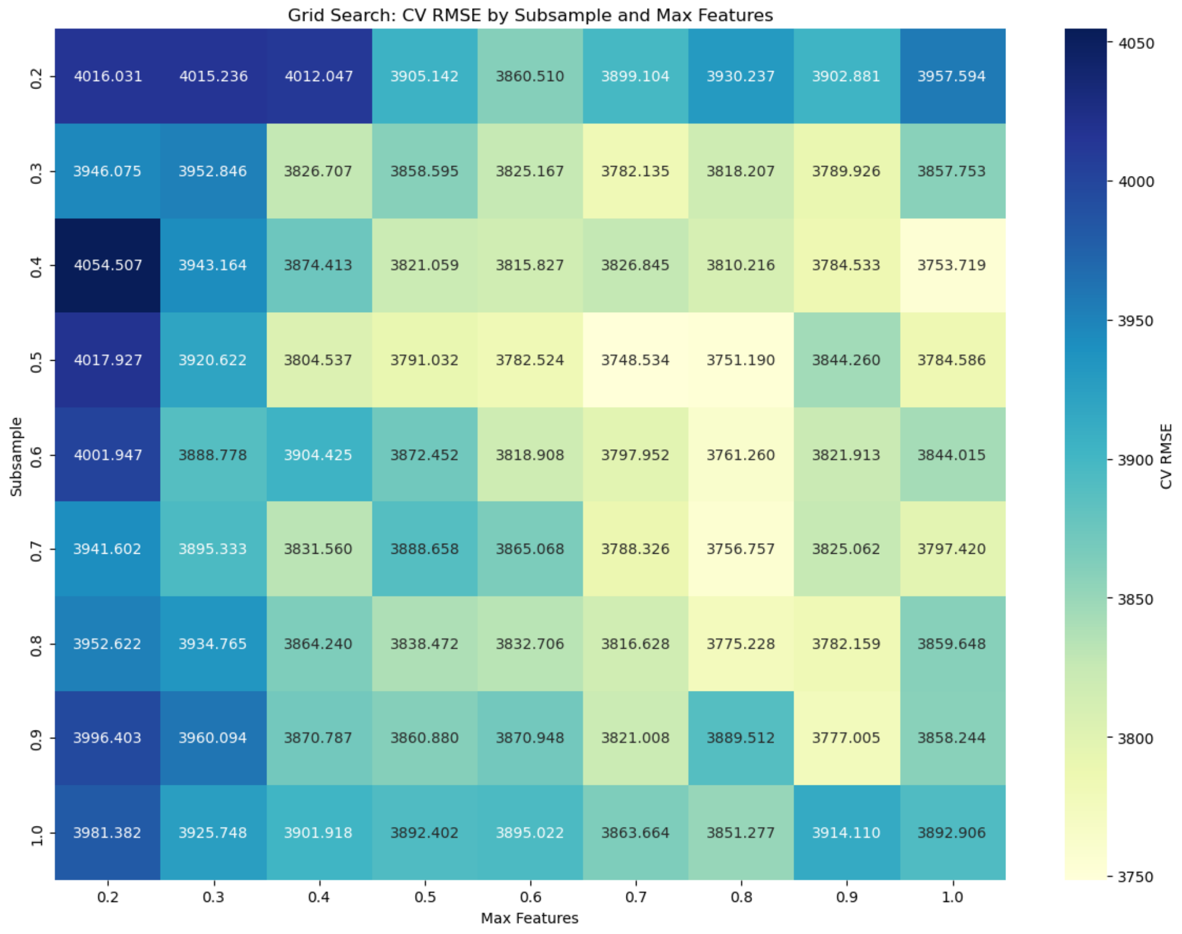
Parameter	Applies To	Purpose
`subsample`	Rows (data points)	Randomly samples a fraction of the training data for each tree
`max_features`	Columns (features)	Randomly samples a fraction of the features for each tree or split

By tuning these parameters, we can reduce overfitting and increase model robustness. However, setting them too low may lead to underfitting due to insufficient information per tree.

```
from sklearn.metrics import make_scorer, mean_squared_error

# Define model
model = GradientBoostingRegressor(n_estimators=100, max_depth=4, learning_rate=0.1, random_s

# Define param grid
param_grid = {
    'subsample': np.linspace(0.2, 1.0, 9),
    'max_features': np.linspace(0.2, 1.0, 9)
}

# RMSE scoring
scorer = make_scorer(mean_squared_error, greater_is_better=False)

# Grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid,
                    scoring=scorer, cv=5, n_jobs=-1, verbose=1)
grid.fit(X_train_final, y_train)

# Create DataFrame from results
results_df = pd.DataFrame(grid.cv_results_)
results_df['mean_rmse'] = np.sqrt(-results_df['mean_test_score'])
```

```
Fitting 5 folds for each of 81 candidates, totalling 405 fits
```

```
# Round subsample and max_features to 2 decimal places for display
results_df['subsample'] = results_df['param_subsample'].astype(float).round(2)
results_df['max_features'] = results_df['param_max_features'].astype(float).round(2)

# Then pivot using the rounded values
heatmap_data = results_df.pivot(index='subsample', columns='max_features', values='mean_rmse

# Plot heatmap
plt.figure(figsize=(12, 9))
sns.heatmap(heatmap_data, annot=True, fmt=".3f", cmap="YlGnBu", cbar_kws={'label': 'CV RMSE'}
plt.title('Grid Search: CV RMSE by Subsample and Max Features')
plt.ylabel('Subsample')
plt.xlabel('Max Features')
plt.tight_layout()
plt.show()

# Find the location (subsample, max_features) of the minimum RMSE
min_rmse = heatmap_data.min().min()
best_location = heatmap_data.stack().idxmin()  # returns a tuple: (subsample, max_features)

print(f"Best RMSE: {min_rmse:.3f} at subsample = {best_location[0]}, max_features = {best_lo
```



```
Best RMSE: 3748.534 at subsample = 0.5, max_features = 0.7
```

9.5.1.6 Effect of Tree Complexity on Cross-Validation Error (Not Tuned Here)

Tree complexity controls how expressive and flexible each individual tree in the gradient boosting ensemble can be. While deeper and more complex trees can capture intricate patterns in the data, they are also more prone to overfitting, especially when combined with many trees.

Key parameters include:

- `max_depth`: Limits the depth of each tree. Shallower trees (e.g., depth 3–6) are preferred for reducing overfitting.
- `min_samples_split`: Specifies the minimum number of samples required to split an internal node. Higher values make the tree more conservative.

- `min_samples_leaf`: Sets the minimum number of samples required to be at a leaf node. This also helps smooth the model and avoid capturing noise.

These parameters influence the bias-variance trade-off by adjusting how expressive each tree can be.

Since we have already discussed and tuned these parameters in earlier lessons (decision trees and random forests), we will **not tune them again here**.

9.5.1.7 Loss Function (`loss`)

In gradient boosting, the loss function determines how the model measures prediction errors and guides the optimization process during training. Here's a breakdown of common loss functions for regression and classification tasks:

- **Regression**:
  - `squared_error`: Penalizes larger errors more heavily; sensitive to outliers. *(Default for regression)*
  - `absolute_error`: Penalizes all errors equally; more robust to outliers.
  - `huber`: Combines squared and absolute error; less sensitive to outliers than `squared_error` and smoother than `absolute_error`.
- **Classification**:
  - `log_loss`: Also known as logistic loss or deviance; commonly used for binary and multiclass classification.
  - `exponential`: Used by AdaBoost; heavily penalizes misclassified points, making it more sensitive to outliers.

Choosing an appropriate loss function ensures the model is optimized for the specific structure and goals of the problem.

9.5.2 Joint Hyperparameter Optimization

Since the optimal values of hyperparameters are often interdependent, they should be tuned **together** rather than in isolation to achieve the best performance.Next we will simultaneously tune multiple core hyperparameters to find the best combination for overall model performance.

9.5.2.1 Using `BayesSearchCV` for Hyperparameter Tuning

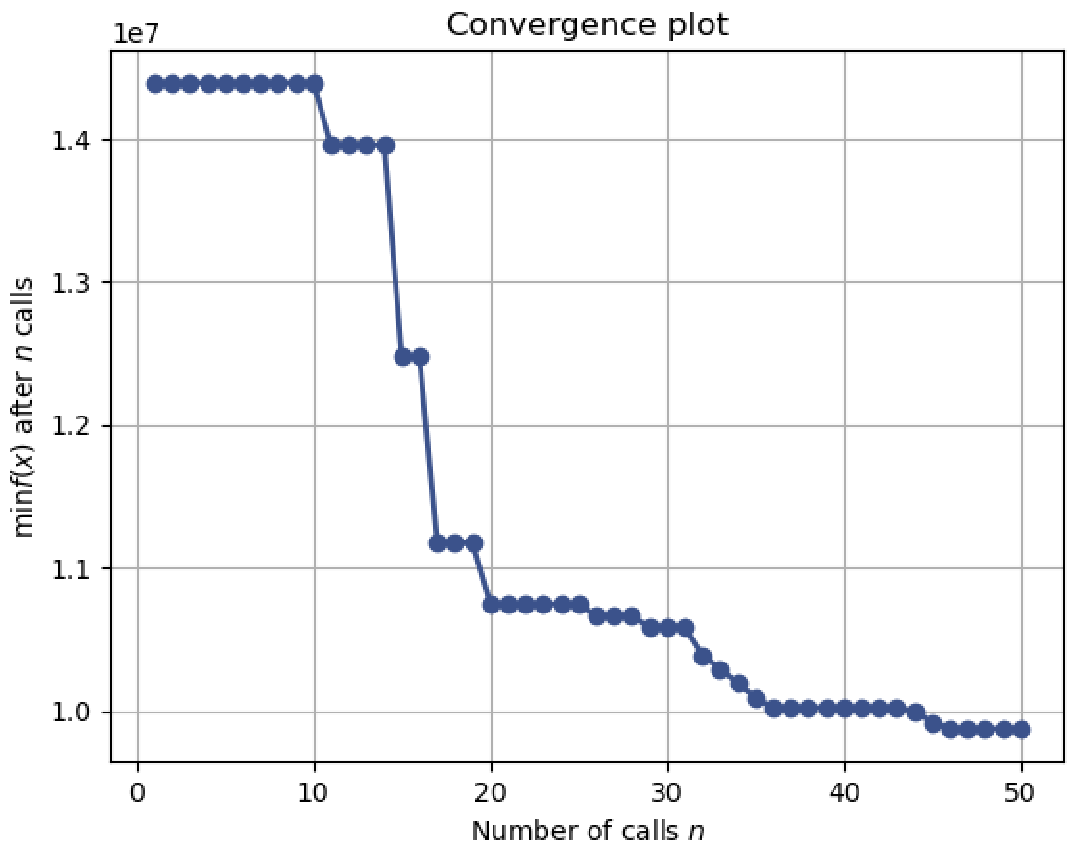
We can use `BayesSearchCV` with early stopping to **simultaneously tune multiple hyper-parameters** in a more efficient and automated way.

```
# time the search
start = time.time()
# Define the search space
search_space = {
    'learning_rate': Real(0.01, 0.8, prior='log-uniform'),  # Prefer lower rates
    'max_depth': Integer(4, 32),          # Shallow trees to prevent overfitting
    'min_samples_split': Integer(2, 100), # Regularize splits
    'min_samples_leaf': Integer(1, 30),  # Regularize leaves
    'subsample': Real(0.1, 1.0),         # Stochastic sampling
    'max_features': Categorical([
        'sqrt', 'log2', None,  # String options
        0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9  # Fractional options (discrete)
    ])  # Feature sampling
}

# Define the model
model_with_early_stopping = GradientBoostingRegressor(
    n_estimators=10000,  # Start with a large number of trees
    validation_fraction=0.1,  # Reserve 10% of training data for validation
    n_iter_no_change=10,      # Stop after 20 rounds of no improvement
    tol=0.001,           # Tolerance for early stopping
    random_state=42
)
# Define the search
bayes_cv  = BayesSearchCV(
    model_with_early_stopping,
    search_space,
    n_iter=50,  # Number of iterations
    scoring='neg_mean_squared_error',
    cv=5,  # Cross-validation folds
    n_jobs=-1,  # Use all available cores
    verbose=1,  # Verbosity level
    random_state=42  # For reproducibility
)
# Fit the model
bayes_cv.fit(X_train_final, y_train)
# Stop the timer
end = time.time()
```

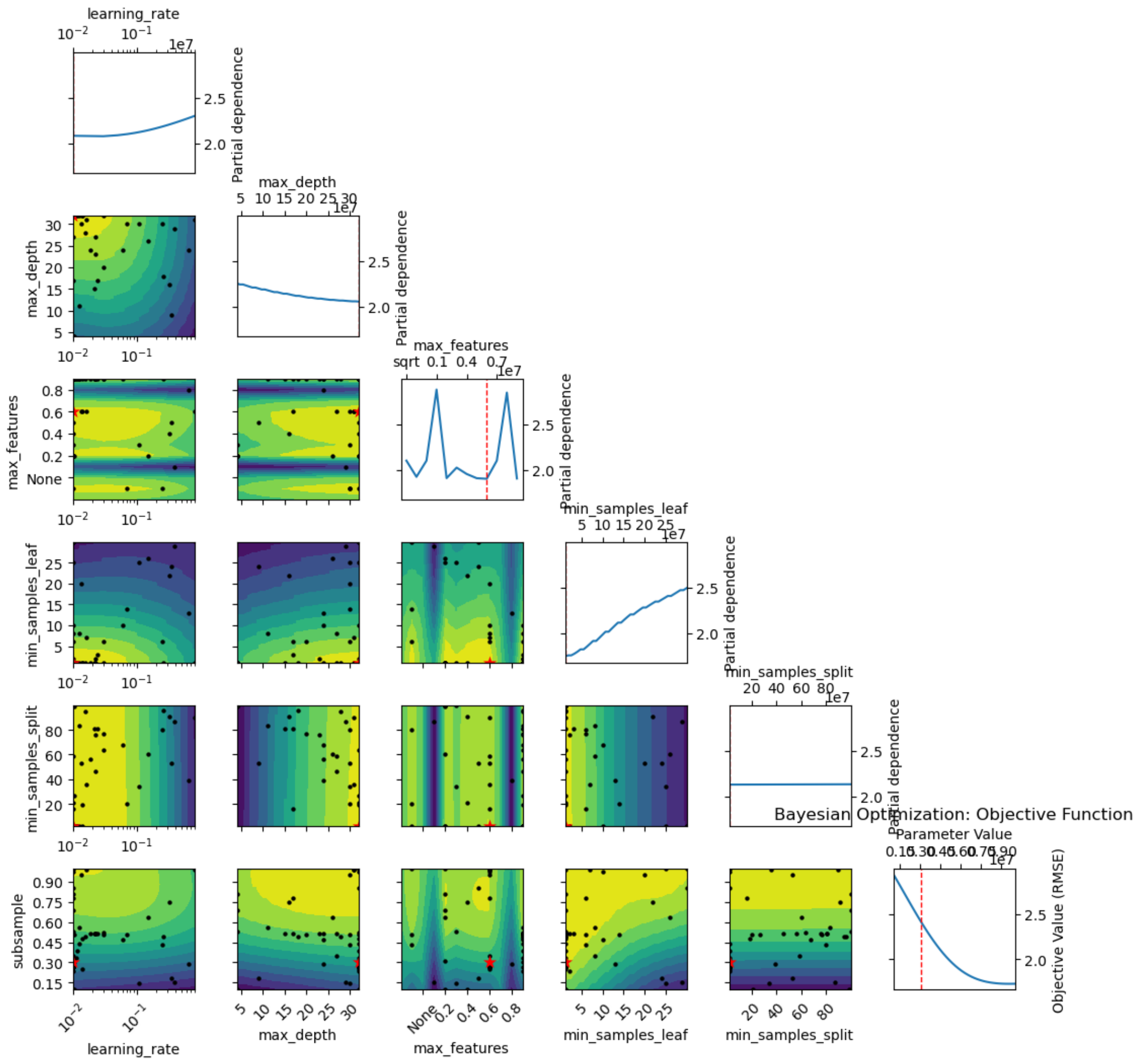
```python
# Calculate elapsed time
elapsed_time = (end - start)/60  # Convert to minutes
# Print elapsed time
print(f"Elapsed time for Bayesian optimization with early stopping: {elapsed_time:.2f} minut

# Extract the best parameters and score
best_params = bayes_cv.best_params_
best_score = np.sqrt(-bayes_cv.best_score_)

print(f"Best Parameters: {best_params}")
print(f"Best CV RMSE: {best_score:.3f}")
```

```
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
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
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
Elapsed time for Bayesian optimization with early stopping: 10.00 minutes
Best Parameters: OrderedDict({'learning_rate': 0.01, 'max_depth': 32, 'max_features': 0.6, '
Best CV RMSE: 3142.483
```

```
# Plot the optimization results
plot_convergence(bayes_cv.optimizer_results_);
```



```
# Plot the objective function
plot_objective(bayes_cv.optimizer_results_[0])
plt.title('Bayesian Optimization: Objective Function')
plt.xlabel('Parameter Value')
plt.ylabel('Objective Value (RMSE)')
plt.show()
```



9.5.2.2 Hyperparameter Optimization with Optuna

```python
def objective(trial):
    # Define hyperparameters to optimize
    params = {
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.8, log=True),
        'max_depth': trial.suggest_int('max_depth', 4, 32),
        'min_samples_split': trial.suggest_int('min_samples_split', 2, 100),
        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 30),
        'subsample': trial.suggest_float('subsample', 0.1, 1.0),
        'max_features': trial.suggest_categorical(
            'max_features',
            ['sqrt', 'log2', None, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
        ),
        'n_iter_no_change': 10,  # Stop if no improvement in 50 rounds
        'validation_fraction': 0.1,  # 10% of training data for validation
        'tol': 0.001,  # Tolerance for early stopping
        'n_estimators': 10000,  # Start with a large number of trees
        'random_state': 42
    }

    # Initialize the model with the parameters, adding early stopping
    model = GradientBoostingRegressor(
        **params
    )
    model = GradientBoostingRegressor(**params)
    # Define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # Perform cross-validation
    scores = cross_val_score(model, X_train_final, y_train, cv=cv, scoring='neg_mean_squared_
    return np.mean(np.sqrt(-scores))
```

```
start = time.time()
# Create a study object
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=70, timeout=600)  # 50 trials or 10 min
# Stop the timer
end = time.time()
# Calculate elapsed time
elapsed_time = (end - start)/60  # Convert to minutes
print(f"Elapsed time for Optuna optimization: {elapsed_time:.2f} minutes")
# Extract the best parameters and score
best_params_optuna = study.best_params
best_score_optuna = study.best_value
print(f"Best Parameters: {best_params_optuna}")
print(f"Best CV RMSE: {best_score_optuna:.3f}")
```

```
[I 2025-05-08 11:07:29,094] A new study created in memory with name: no-name-84a8f2a9-1577-4
[I 2025-05-08 11:07:30,013] Trial 0 finished with value: 6402.677876019744 and parameters: {
[I 2025-05-08 11:07:31,621] Trial 1 finished with value: 5817.716936023959 and parameters: {
```

```
[I 2025-05-08 11:07:33,048] Trial 2 finished with value: 4639.695965157119 and parameters: {
[I 2025-05-08 11:07:33,778] Trial 3 finished with value: 3888.640108615851 and parameters: {
[I 2025-05-08 11:07:34,527] Trial 4 finished with value: 5637.440291328167 and parameters: {
[I 2025-05-08 11:08:15,193] Trial 5 finished with value: 4349.349334283657 and parameters: {
[I 2025-05-08 11:08:16,690] Trial 6 finished with value: 3538.201600150815 and parameters: {
[I 2025-05-08 11:08:40,300] Trial 7 finished with value: 4222.0969339973235 and parameters:
[I 2025-05-08 11:08:40,844] Trial 8 finished with value: 5987.876440565123 and parameters: {
[I 2025-05-08 11:08:41,963] Trial 9 finished with value: 5930.236597113848 and parameters: {
[I 2025-05-08 11:08:43,184] Trial 10 finished with value: 4249.647691193036 and parameters:
[I 2025-05-08 11:08:43,792] Trial 11 finished with value: 3721.412018003831 and parameters:
[I 2025-05-08 11:08:44,200] Trial 12 finished with value: 3992.3681134868325 and parameters:
[I 2025-05-08 11:08:45,596] Trial 13 finished with value: 3745.0231375714984 and parameters:
[I 2025-05-08 11:08:46,415] Trial 14 finished with value: 4496.274561417728 and parameters:
[I 2025-05-08 11:08:50,798] Trial 15 finished with value: 3532.351997763184 and parameters:
[I 2025-05-08 11:08:57,644] Trial 16 finished with value: 3712.683820416855 and parameters:
[I 2025-05-08 11:09:08,351] Trial 17 finished with value: 3878.4225603020363 and parameters:
[I 2025-05-08 11:09:12,426] Trial 18 finished with value: 3258.6536376666345 and parameters:
[I 2025-05-08 11:09:14,871] Trial 19 finished with value: 3299.8057419420575 and parameters:
[I 2025-05-08 11:09:22,909] Trial 20 finished with value: 3558.1548802314146 and parameters:
[I 2025-05-08 11:09:24,627] Trial 21 finished with value: 3344.1360986274194 and parameters:
[I 2025-05-08 11:09:26,856] Trial 22 finished with value: 3346.290872146259 and parameters:
[I 2025-05-08 11:09:30,886] Trial 23 finished with value: 3682.572188125887 and parameters:
[I 2025-05-08 11:09:36,517] Trial 24 finished with value: 3324.1356706329725 and parameters:
[I 2025-05-08 11:09:48,021] Trial 25 finished with value: 3477.877832840085 and parameters:
[I 2025-05-08 11:09:51,103] Trial 26 finished with value: 3543.4521345710564 and parameters:
[I 2025-05-08 11:09:54,116] Trial 27 finished with value: 3234.0013427023514 and parameters:
[I 2025-05-08 11:10:00,202] Trial 28 finished with value: 3508.54835466015 and parameters: {
[I 2025-05-08 11:10:02,298] Trial 29 finished with value: 5004.42107047821 and parameters: {
[I 2025-05-08 11:10:03,666] Trial 30 finished with value: 4324.686970916422 and parameters:
[I 2025-05-08 11:10:11,068] Trial 31 finished with value: 3484.424151386007 and parameters:
[I 2025-05-08 11:10:18,699] Trial 32 finished with value: 3291.106187349627 and parameters:
[I 2025-05-08 11:10:22,168] Trial 33 finished with value: 3325.6927625938033 and parameters:
[I 2025-05-08 11:10:27,550] Trial 34 finished with value: 3243.0601177315375 and parameters:
[I 2025-05-08 11:10:36,064] Trial 35 finished with value: 3258.8499173154833 and parameters:
[I 2025-05-08 11:10:40,944] Trial 36 finished with value: 4733.54257152398 and parameters: {
[I 2025-05-08 11:10:55,316] Trial 37 finished with value: 3330.1658474531714 and parameters:
[I 2025-05-08 11:11:08,984] Trial 38 finished with value: 3657.4081420035955 and parameters:
[I 2025-05-08 11:11:11,528] Trial 39 finished with value: 3246.3324692045935 and parameters:
[I 2025-05-08 11:11:14,072] Trial 40 finished with value: 3223.908916424979 and parameters:
[I 2025-05-08 11:11:16,764] Trial 41 finished with value: 3262.2664150628366 and parameters:
[I 2025-05-08 11:11:18,027] Trial 42 finished with value: 3536.1048831110393 and parameters:
[I 2025-05-08 11:11:20,006] Trial 43 finished with value: 3725.757422297686 and parameters:
[I 2025-05-08 11:11:21,612] Trial 44 finished with value: 3401.5545066433783 and parameters:
```

```
[I 2025-05-08 11:11:25,264] Trial 45 finished with value: 3635.4930797564184 and parameters:
[I 2025-05-08 11:11:26,855] Trial 46 finished with value: 3297.8485341280875 and parameters:
[I 2025-05-08 11:11:28,066] Trial 47 finished with value: 3484.8710748592084 and parameters:
[I 2025-05-08 11:11:29,916] Trial 48 finished with value: 5067.6932650619365 and parameters:
[I 2025-05-08 11:11:33,570] Trial 49 finished with value: 3519.5312534022605 and parameters:
[I 2025-05-08 11:11:38,292] Trial 50 finished with value: 4947.269431523515 and parameters:
[I 2025-05-08 11:11:42,794] Trial 51 finished with value: 3223.184387572671 and parameters:
[I 2025-05-08 11:11:47,351] Trial 52 finished with value: 3215.28175313329 and parameters: {
[I 2025-05-08 11:11:51,995] Trial 53 finished with value: 3251.8248895524334 and parameters:
[I 2025-05-08 11:12:03,513] Trial 54 finished with value: 3397.6799413335893 and parameters:
[I 2025-05-08 11:12:10,304] Trial 55 finished with value: 3260.8279178294406 and parameters:
[I 2025-05-08 11:12:12,294] Trial 56 finished with value: 3192.9791005502093 and parameters:
[I 2025-05-08 11:12:13,927] Trial 57 finished with value: 3269.998258903558 and parameters:
[I 2025-05-08 11:12:17,716] Trial 58 finished with value: 3457.122430183835 and parameters:
[I 2025-05-08 11:12:20,307] Trial 59 finished with value: 3247.949537776105 and parameters:
[I 2025-05-08 11:12:51,502] Trial 60 finished with value: 4050.280737550346 and parameters:
[I 2025-05-08 11:12:56,091] Trial 61 finished with value: 3249.9684847878243 and parameters:
[I 2025-05-08 11:13:01,774] Trial 62 finished with value: 3328.4822044271473 and parameters:
[I 2025-05-08 11:13:12,800] Trial 63 finished with value: 3490.4301182885783 and parameters:
[I 2025-05-08 11:13:15,262] Trial 64 finished with value: 3540.8369062274187 and parameters:
[I 2025-05-08 11:13:20,471] Trial 65 finished with value: 3294.353628295925 and parameters:
[I 2025-05-08 11:13:29,893] Trial 66 finished with value: 3326.4094166662317 and parameters:
[I 2025-05-08 11:13:44,533] Trial 67 finished with value: 3548.591414006273 and parameters:
[I 2025-05-08 11:13:51,439] Trial 68 finished with value: 3228.1005724210245 and parameters:
[I 2025-05-08 11:13:58,247] Trial 69 finished with value: 3264.495066558836 and parameters:
```

```
Elapsed time for Optuna optimization: 6.49 minutes
Best Parameters: {'learning_rate': 0.03446122898299111, 'max_depth': 25, 'min_samples_split'
Best CV RMSE: 3192.979
```

```
optuna.visualization.plot_optimization_history(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
optuna.visualization.plot_param_importances(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
optuna.visualization.plot_parallel_coordinate(study).show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

9.6 Independent Study

In this notebook, we used the car dataset for a guided regression task to illustrate the core hyperparameters in gradient boosting and how to tune them to balance bias and variance.

For your practice, please work with the **diabetes dataset** and complete the following:

- Fit a baseline gradient boosting classifier.
- Tune key hyperparameters: `learning_rate`, `n_estimators`, `max_depth`, and `subsample`.
- Use **early stopping** to determine the optimal number of trees.
- Compare training and test `roc_auc` before and after tuning.
- Visualize the learning curve (training vs test error across iterations).
- Summarize what combination of hyperparameters yielded the best performance and how they impacted bias and variance.

Feel free to use `GridSearchCV`, `BayesSearchCV`, or other tuning as you prefer.

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

```
print(train.shape, test.shape)
train.head()
```

```
(614, 9) (154, 9)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	2	88	74	19	53	29.0	0.229	22
1	2	129	84	0	0	28.0	0.284	27
2	0	102	78	40	90	34.5	0.238	24
3	0	123	72	0	0	36.3	0.258	52
4	1	144	82	46	180	46.1	0.335	46

```
# check the distribution of the target variable
train['Outcome'].value_counts(normalize=True)
```

```
Outcome
0    0.662866
1    0.337134
Name: proportion, dtype: float64
```

```
# define the features and target variable
X_train = train.drop(columns=['Outcome'])
y_train = train['Outcome']
X_test = test.drop(columns=['Outcome'])
y_test = test['Outcome']
```

9.7 Foundational Paper

The foundational paper introducing "vanilla" Gradient Boosting is:

Greedy Function Approximation: A Gradient Boosting Machine
Author: Jerome H. Friedman
Published in: *The Annals of Statistics*, 2001, Vol. 29, No. 5, pp. 1189–1232
DOI: 10.1214/aos/1013203451

10 XGBoost

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

XGBoost (Extreme Gradient Boosting) is a scalable and efficient implementation of gradient boosting developed by Tianqi Chen and Carlos Guestrin in 2016. It has become one of the most popular machine learning algorithms for structured/tabular data, widely used in Kaggle competitions and production environments.

Compared to vanilla Gradient Boosting, XGBoost includes additional system-level and algorithmic optimizations such as:

- Regularization (to reduce overfitting)
- Tree pruning
- Parallelized tree construction
- Missing value handling
- Out-of-core computation for large datasets

10.1 XGBoost Intuition

XGBoost extends vanilla gradient boosting with:

- **Regularization**: Penalizes complex models via L1/L2 terms in the loss function.
- **Second-order optimization**: Uses both gradients and hessians for faster convergence and more accurate splits.
- **Split constraints**: Prevents splits with insufficient gain (via `gamma`) during tree growth, avoiding the need for post-pruning.

10.2 How XGBoost Works (Regression Example)

XGBoost minimizes the following **regularized objective** at each boosting round t:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Where: - l is a differentiable convex loss function (e.g., squared error) - f_t is the prediction function at iteration t (a tree) - $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum w_j^2$ is the regularization term (penalizes the number of leaves T and leaf weights w_j)

To simplify optimization, XGBoost applies a **second-order Taylor approximation** of the loss function:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$

Where: - $g_i = \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i}$ is the first-order derivative (gradient) - $h_i = \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$ is the second-order derivative (hessian)

XGBoost then chooses the **tree structure and leaf values** that minimize this approximate objective.

10.3 Using XGBoost

Although **XGBoost is not part of Scikit-learn**, it provides a **Scikit-learn-compatible API** through the `xgboost.sklearn` module. This allows you to use XGBoost models seamlessly with Scikit-learn tools such as `Pipeline`, `GridSearchCV`, and `cross_val_score`.

The main classes are:

- `XGBRegressor`: for regression tasks
- `XGBClassifier`: for classification tasks

To install the package:

```
pip install xgboost
```

> **Note:** XGBoost is a separate library, not part of Scikit-learn, but it provides a **Scikit-learn-compatible API** via `XGBClassifier` and `XGBRegressor`.
> This makes it easy to integrate XGBoost models into Scikit-learn workflows such as `Pipeline`, `GridSearchCV`, and `cross_val_score`.

10.4 Core Hyperparameter Categories

10.4.1 Model Complexity

- `n_estimators`: Number of boosting rounds
- `max_depth`: Maximum depth of a tree
- `min_child_weight`: Minimum sum of instance weight needed in a child

10.4.2 Learning and Regularization

- `learning_rate` (`eta`): Shrinkage rate to scale each tree's contribution
- `subsample`: Fraction of rows used per boosting round
- `colsample_bytree`: Fraction of features used per tree
- `colsample_bylevel`, `colsample_bynode`: Further control over feature subsampling

10.4.3 Regularization

- `gamma`: Minimum loss reduction required to make a further partition
- `reg_alpha`: L1 regularization term on weights (Lasso)
- `reg_lambda`: L2 regularization term on weights (Ridge)

10.4.4 Optimization Control

- `objective`: Loss function (e.g., `'reg:squarederror'`, `'binary:logistic'`)
- `tree_method`: Tree construction algorithm (`'auto'`, `'hist'`, `'gpu_hist'`)
- `early_stopping_rounds`: Stop if validation score doesn't improve after N rounds

However, there are other hyperparameters that can be tuned as well. Check out the list of all hyperparameters in the XGBoost documentation.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import root_mean_squared_error, r2_score, accuracy_score, precision_sco
from xgboost import XGBRegressor, XGBClassifier
import seaborn as sns

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
# 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

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


# Identify categorical and numerical columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(exclude=['object']).columns.tolist()

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

Let's define some helper functions before building any models.

```
# Create preprocessing for numerical and categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numerical_cols),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols)
    ]
)

# Function to evaluate model
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    rmse = root_mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"Root Mean Squared Error: {rmse:.2f}")
    print(f"R² Score: {r2:.4f}")

    return rmse,  r2

# Function to plot feature importance
def plot_feature_importance(model, preprocessor, X):
    if hasattr(model, 'feature_importances_'):
        # Get feature names after one-hot encoding
        cat_features = preprocessor.named_transformers_['cat'].get_feature_names_out(categor
        all_features = np.append(numerical_cols, cat_features)

        # Get feature importances
        importances = model.feature_importances_

        # Sort feature importances in descending order
        indices = np.argsort(importances)[::-1]

        # Create a DataFrame for easier visualization
        importance_df = pd.DataFrame({
            'Feature': all_features[indices][:20],  # Top 20 features
            'Importance': importances[indices][:20]
        })
        importance_df = importance_df.sort_values(by='Importance', ascending=False)
        return importance_df
```

10.4.5 Baseline Model

```python
# ===== 1. Baseline Model =====
print("\n===== Baseline XGBoost Model =====")
baseline_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(random_state=42))
])

baseline_pipeline.fit(X_train, y_train)
print("\nBaseline Model Evaluation:")
baseline_metrics = evaluate_model(baseline_pipeline, X_test, y_test)

# Plot feature importance for baseline model
baseline_importance = plot_feature_importance(baseline_pipeline.named_steps['regressor'],
                                              baseline_pipeline.named_steps['preprocessor'],
                                              X)
```

```
===== Baseline XGBoost Model =====

Baseline Model Evaluation:
Root Mean Squared Error: 3354.16
R² Score: 0.9617
```

10.4.6 Early Stopping in XGBoost

Early stopping is a technique that stops training when the model's performance on a validation set stops improving, helping to prevent overfitting and reduce training time.

10.4.6.1 How It Works

At each boosting round, XGBoost tracks a performance metric (e.g., RMSE or log loss) on a **validation set**. If the metric doesn't improve for a specified number of rounds (`early_stopping_rounds`), training is halted.

- Saves computation by avoiding unnecessary boosting rounds.
- Returns the model from the best iteration (with the lowest validation error).

10.4.6.2 Requirements

- You must provide an `eval_set` containing a validation set.
- The evaluation metric must be one that XGBoost can track (`eval_metric` is optional but recommended).

```
# ===== 2. Early Stopping =====
print("\n===== XGBoost with Early Stopping =====")
# Create validation set for early stopping
X_train_es, X_val, y_train_es, y_val = train_test_split(X_train, y_train, test_size=0.1, ranc

# Preprocess the validation set
preprocessor_fit = preprocessor.fit(X_train_es)
X_train_es_transformed = preprocessor_fit.transform(X_train_es)
X_val_transformed = preprocessor_fit.transform(X_val)


# Train with early stopping
early_stop_model = XGBRegressor(
    random_state=42,
    n_estimators=1000,
    early_stopping_rounds=20
)
early_stop_model.fit(
    X_train_es_transformed, y_train_es,
    eval_set=[(X_val_transformed, y_val)],
    verbose=False
)
```

```

===== XGBoost with Early Stopping =====


XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=20,
             enable_categorical=False, eval_metric=None, feature_types=None,
             feature_weights=None, gamma=None, grow_policy=None,
             importance_type=None, interaction_constraints=None,
             learning_rate=None, max_bin=None, max_cat_threshold=None,
             max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
             max_leaves=None, min_child_weight=None, missing=nan,
             monotone_constraints=None, multi_strategy=None, n_estimators=1000,
             n_jobs=None, num_parallel_tree=None, ...)
```

```
# Update the pipeline with the best model
early_stop_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(
        random_state=42,
        n_estimators=early_stop_model.best_iteration,  # Use the best number of iterations
    ))
])

early_stop_pipeline.fit(X_train, y_train)
print("\nEarly Stopping Model Evaluation:")
early_stop_metrics = evaluate_model(early_stop_pipeline, X_test, y_test)
print(f"Best number of iterations: {early_stop_model.best_iteration}")
```

```
Early Stopping Model Evaluation:
Root Mean Squared Error: 3332.03
R² Score: 0.9622
Best number of iterations: 193
```

10.4.7 `gamma` in XGBoost

Definition:
`gamma` (also called `min_split_loss`) specifies the **minimum loss reduction required** to make a further partition (split) on a leaf node of the tree.

How it works:

- During tree construction, XGBoost evaluates whether splitting a node reduces the overall training loss.
- If the **reduction in loss is less than `gamma`**, the split is **discarded**, and the node becomes a leaf.
- Higher values of `gamma` make the algorithm more **conservative**, leading to **simpler trees**.

Formula:
At each split, XGBoost calculates the gain (reduction in regularized loss):

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right) - \gamma$$

Where: - G_L, H_L: gradient and hessian sums for the **left child** - G_R, H_R: gradient and hessian sums for the **right child** - λ: L2 regularization term - γ: minimum loss reduction required to make a split

Effect of gamma:

gamma Value	Behavior	Risk
0 (default)	Most splits are allowed	Overfitting possible
Moderate	Small-gain splits are blocked	More robust trees
High	Very few splits allowed	Underfitting possible

Use case: - Tune gamma to **prune noisy or unnecessary splits**. - Helpful when the model is **overfitting**, especially on small datasets.

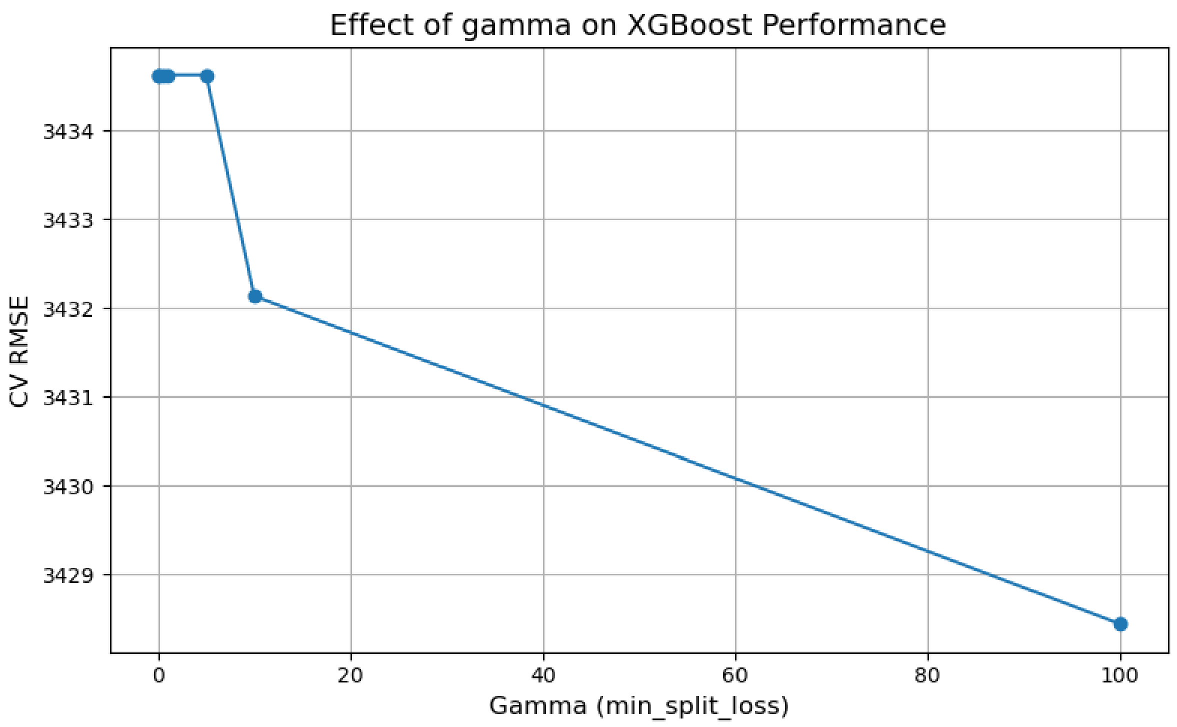
Example:

```
XGBRegressor(gamma=1.0)
```

```
# ===== 3. Regularization Experiments: varying gamma =====

print("\n===== XGBoost with Regularization: Varying Gamma =====")
# Define the parameter grid for gamma
param_grid = {
    'regressor__gamma': [0, 0.1, 0.5, 1, 5, 10, 100],
}
# Create a new pipeline for the regularization experiment
regularization_gamma_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(random_state=42, n_estimators=early_stop_model.best_iteration
])

# Perform grid search with cross-validation
grid_search_gamma = GridSearchCV(
    regularization_gamma_pipeline,
    param_grid,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

grid_search_gamma.fit(X_train, y_train)
print("\nBest Parameters from Regularization Tuning (  - gamma)::")
print(grid_search_gamma.best_params_)
print("Best Cross-Validation RMSE: {:.2f}".format(np.sqrt(-grid_search_gamma.best_score_)))

# Evaluate the best model from grid search
best_gamma_model = grid_search_gamma.best_estimator_
print("\nTest Set Evaluation for Best Model from Gamma Regularization Tuning:")
regularization_metrics = evaluate_model(best_gamma_model, X_test, y_test)
```

```
===== XGBoost with Regularization: Varying Gamma =====

Best Parameters from Regularization Tuning (  - gamma)::
{'regressor__gamma': 100}
Best Cross-Validation RMSE: 3428.44

Test Set Evaluation for Best Model from Gamma Regularization Tuning:
Root Mean Squared Error: 3332.02
R² Score: 0.9622
```

```
# Extract gamma values and corresponding mean CV RMSE
gamma_values = grid_search_gamma.cv_results_['param_regressor__gamma'].data

# Convert mean test scores to RMSE, rounding to 2 decimal places
mean_rmse_scores = np.sqrt(-grid_search_gamma.cv_results_['mean_test_score'])
# Round the RMSE scores to 2 decimal places
mean_rmse_scores = np.round(mean_rmse_scores, 4)

# Plot gamma vs RMSE with plain y-axis tick labels
plt.figure(figsize=(8, 5))
plt.plot(gamma_values, mean_rmse_scores, marker='o')
plt.xlabel('Gamma (min_split_loss)', fontsize=12)
plt.ylabel('CV RMSE', fontsize=12)
plt.title('Effect of gamma on XGBoost Performance', fontsize=14)
plt.grid(True)

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



Effect of `gamma` on XGBoost Performance

As `gamma` increases, XGBoost becomes more selective about making splits.

- **Low gamma (0–5)**: Trees grow freely → higher RMSE due to possible overfitting.
- **Moderate to high gamma (10–100)**: Blocks weak splits → simpler trees with better validation performance.

In this dataset, higher `gamma` values improved generalization by preventing unnecessary splits.

10.4.8 `reg_lambda` and `reg_alpha` in XGBoost

XGBoost includes **regularization** to help prevent overfitting by penalizing complex trees.

- `reg_lambda` (L2 regularization):
  - Penalizes large leaf weights using a squared penalty.
  - Encourages smaller, smoother weight values (like Ridge regression).
  - Helps when many features contribute weakly.
- `reg_alpha` (L1 regularization):
  - Penalizes absolute values of leaf weights.

  - Can shrink some weights to zero, effectively performing feature selection (like Lasso).
  - Useful when you expect only a few strong features.

Objective function with regularization:

$$\mathcal{L} = \text{Loss} + \gamma T + \frac{1}{2}\lambda \sum_j w_j^2 + \alpha \sum_j |w_j|$$

Where: - Loss: training loss (e.g., squared error or log loss) - T: number of leaves in the tree - w_j: weight of the j-th leaf - λ: L2 regularization (Ridge penalty) - α: L1 regularization (Lasso penalty) - γ: cost for adding a new leaf (controls tree growth)

Understanding them via Ridge, Lasso, and ElasticNet you learned in STAT303-2

Just like Ridge/Lasso regularization helps linear models generalize better, `reg_lambda` and `reg_alpha` help XGBoost prevent overfitting by controlling how complex the trees become through leaf weight penalties.

- `reg_alpha` $= 0 \rightarrow$ **No L1 penalty**, behaves like Ridge (only L2 used)
- `reg_lambda` $= 0 \rightarrow$ **No L2 penalty**, behaves like Lasso (only L1 used)
- `reg_alpha > 0` and `reg_lambda > 0` $\rightarrow$ behaves like **ElasticNet**

This analogy helps understand how XGBoost controls model complexity:

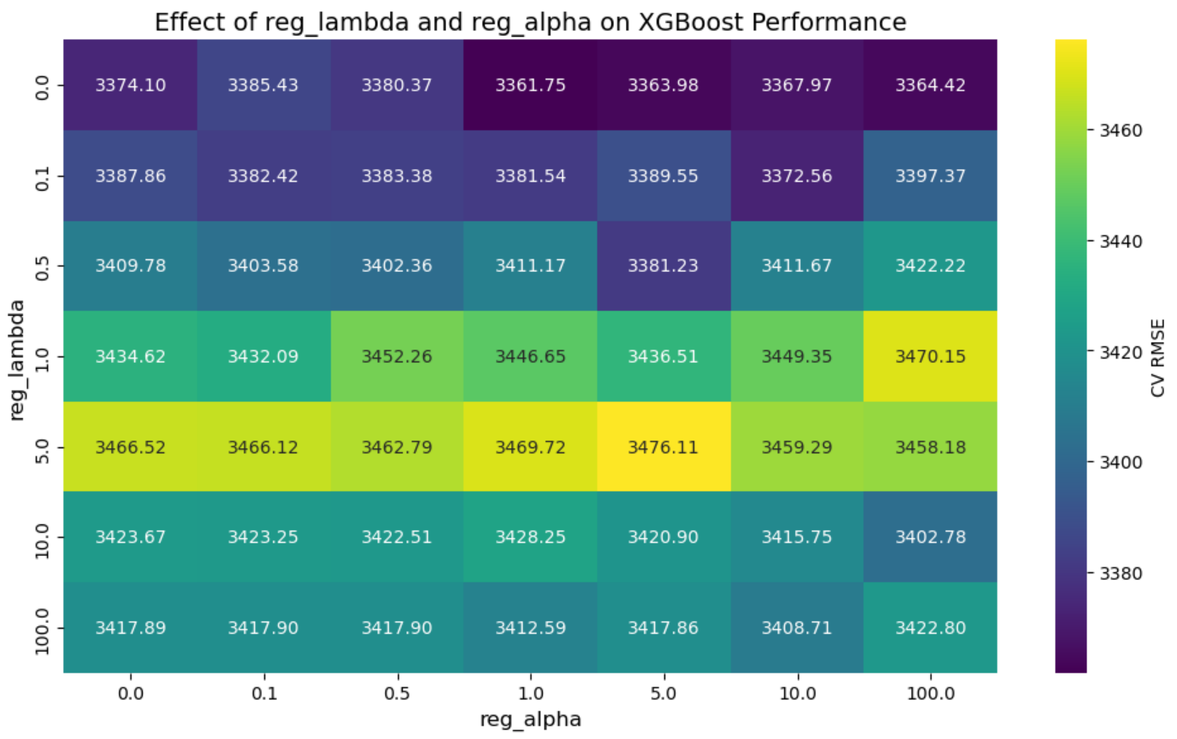
Setting	Behavior
`reg_alpha=0`, `reg_lambda>0`	Like **Ridge** → smooth leaf weights, all included
`reg_alpha>0`, `reg_lambda=0`	Like **Lasso** → some leaf weights may shrink to zero
Both > 0	Like **ElasticNet** → balance shrinkage and sparsity
Both $= 0$ (default)	No regularization → may overfit on small/noisy data

```python
# ===== 3. Regularization Experiments: tuning reg_lambda and reg_alpha =====
print("\n===== Exploring Regularization Parameters: reg_lambda and reg_alpha =====")

# Define the parameter grid for reg_lambda and reg_alpha
param_grid_reg = {
    'regressor__reg_lambda': [0, 0.1, 0.5, 1, 5, 10, 100],
    'regressor__reg_alpha': [0, 0.1, 0.5, 1, 5, 10, 100],
}

# Create a new pipeline for the regularization experiment
regularization_lambda_alpha_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(random_state=42, n_estimators=early_stop_model.best_iteration
])

# Perform grid search with cross-validation
grid_search_lambda_alpha_reg = GridSearchCV(
    regularization_lambda_alpha_pipeline,
    param_grid_reg,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)
grid_search_lambda_alpha_reg.fit(X_train, y_train)
print("\nBest Parameters from Lambda and Alpha Regularization Tuning:")
print(grid_search_lambda_alpha_reg.best_params_)
print("Best Cross-Validation RMSE: {:.2f}".format(np.sqrt(-grid_search_lambda_alpha_reg.best

# Evaluate the best model from grid search
best_lambda_alpha_model = grid_search_lambda_alpha_reg.best_estimator_
print("\nTest Set Evaluation for Best Regularization Model (  and  ):")
regularization_metrics_reg = evaluate_model(best_lambda_alpha_model, X_test, y_test)
```

```
===== Exploring Regularization Parameters: reg_lambda and reg_alpha =====

Best Parameters from Lambda and Alpha Regularization Tuning:
{'regressor__reg_alpha': 1, 'regressor__reg_lambda': 0}
Best Cross-Validation RMSE: 3361.75

Test Set Evaluation for Best Regularization Model (  and  ):
Root Mean Squared Error: 3551.09
R² Score: 0.9570
```

```
# Extract reg_lambda and reg_alpha values and corresponding mean CV RMSE
reg_lambda_values = grid_search_lambda_alpha_reg.cv_results_['param_regressor__reg_lambda'].
reg_alpha_values = grid_search_lambda_alpha_reg.cv_results_['param_regressor__reg_alpha'].da

# Convert mean test scores to RMSE, rounding to 2 decimal places
mean_rmse_scores_reg = np.sqrt(-grid_search_lambda_alpha_reg.cv_results_['mean_test_score'])
# Round the RMSE scores to 2 decimal places
mean_rmse_scores_reg = np.round(mean_rmse_scores_reg, 4)

# Create a DataFrame for easier plotting
```

```
reg_lambda_alpha_results_df = pd.DataFrame({
    'reg_lambda': reg_lambda_values,
    'reg_alpha': reg_alpha_values,
    'mean_rmse': mean_rmse_scores_reg
})

# Pivot the DataFrame for heatmap
regreg_lambda_alpha_results_df_pivot_df = reg_lambda_alpha_results_df.pivot(index='reg_lambda

# Plotting the heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(regreg_lambda_alpha_results_df_pivot_df, annot=True, fmt=".2f", cmap='viridis', 
plt.title('Effect of reg_lambda and reg_alpha on XGBoost Performance', fontsize=14)
plt.xlabel('reg_alpha', fontsize=12)
plt.ylabel('reg_lambda', fontsize=12)
plt.tight_layout();
```



reg_lambda \ reg_alpha	0.0	0.1	0.5	1.0	5.0	10.0	100.0
0.0	3374.10	3385.43	3380.37	3361.75	3363.98	3367.97	3364.42
0.1	3387.86	3382.42	3383.38	3381.54	3389.55	3372.56	3397.37
0.5	3409.78	3403.58	3402.36	3411.17	3381.23	3411.67	3422.22
1.0	3434.62	3432.09	3452.26	3446.65	3436.51	3449.35	3470.15
5.0	3466.52	3466.12	3462.79	3469.72	3476.11	3459.29	3458.18
10.0	3423.67	3423.25	3422.51	3428.25	3420.90	3415.75	3402.78
100.0	3417.89	3417.90	3417.90	3412.59	3417.86	3408.71	3422.80

10.4.9 Exploring Regularization Hyperparameters Simultaneously

In addition to `gamma`, `reg_lambda`, and `reg_alpha`, the parameters `max_depth` and `min_child_weight` also control the complexity of XGBoost models. These parameters behave similarly to how they work in other tree-based models.

Rather than tuning them in isolation, it's important to recognize that these parameters **interact** with one another. In the next step, we will tune them **simultaneously** to better capture their combined effect on model performance.

```
# ===== 3. Regularization Experiments: Simultaneous Exploration  =====
print("\n===== Exploring Regularization Parameters Simultaneously =====")
# Define regularization parameters to test
reg_params = [
    {'regressor__max_depth': 3, 'regressor__min_child_weight': 1, 'regressor__gamma': 0,
     'regressor__reg_alpha': 0, 'regressor__reg_lambda': 1},
    {'regressor__max_depth': 3, 'regressor__min_child_weight': 1, 'regressor__gamma': 0,
     'regressor__reg_alpha': 1, 'regressor__reg_lambda': 1},
    {'regressor__max_depth': 5, 'regressor__min_child_weight': 3, 'regressor__gamma': 0.1,
     'regressor__reg_alpha': 0, 'regressor__reg_lambda': 1},
    {'regressor__max_depth': 5, 'regressor__min_child_weight': 3, 'regressor__gamma': 0.1,
     'regressor__reg_alpha': 1, 'regressor__reg_lambda': 5},
    {'regressor__max_depth': 7, 'regressor__min_child_weight': 1, 'regressor__gamma': 0.2,
     'regressor__reg_alpha': 5, 'regressor__reg_lambda': 10}
]

# Store results for comparison
reg_results = []

for i, params in enumerate(reg_params):
    print(f"\nRegularization Test {i+1}:")
    print(params)

    # Create pipeline with these parameters
    reg_pipeline = Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', XGBRegressor(
            random_state=42,
            n_estimators=early_stop_model.best_iteration,
            **{k.replace('regressor__', ''): v for k, v in params.items()}
        ))
    ])
    # Train and evaluate
    reg_pipeline.fit(X_train, y_train)
    print("\nModel Evaluation:")
    metrics = evaluate_model(reg_pipeline, X_test, y_test)
    rmse, r2 = metrics

    # Store results
    reg_results.append({
        'test': i+1,
        'params': params,
        'rmse': rmse,
        'r2': r2
    })

# Find best regularization parameters
reg_df = pd.DataFrame(reg_results)
best_reg_idx = reg_df['rmse'].idxmin()
best_reg_params = reg_df.loc[best_reg_idx, 'params']
print("\nBest Regularization Parameters:")
print(best_reg_params)
print("Best RMSE: {:.2f}".format(reg_df['rmse'].min()))
```

```
===== Exploring Regularization Parameters Simultaneously =====

Regularization Test 1:
{'regressor__max_depth': 3, 'regressor__min_child_weight': 1, 'regressor__gamma': 0, 'regress

Model Evaluation:
Root Mean Squared Error: 3774.48
R² Score: 0.9514

Regularization Test 2:
{'regressor__max_depth': 3, 'regressor__min_child_weight': 1, 'regressor__gamma': 0, 'regress

Model Evaluation:
Root Mean Squared Error: 3774.48
R² Score: 0.9514

Regularization Test 3:
{'regressor__max_depth': 5, 'regressor__min_child_weight': 3, 'regressor__gamma': 0.1, 'regre
```

```
Model Evaluation:
Root Mean Squared Error: 3334.18
R² Score: 0.9621

Regularization Test 4:
{'regressor__max_depth': 5, 'regressor__min_child_weight': 3, 'regressor__gamma': 0.1, 'regre

Model Evaluation:
Root Mean Squared Error: 3298.88
R² Score: 0.9629

Regularization Test 5:
{'regressor__max_depth': 7, 'regressor__min_child_weight': 1, 'regressor__gamma': 0.2, 'regre

Model Evaluation:
Root Mean Squared Error: 3222.51
R² Score: 0.9646

Best Regularization Parameters:
{'regressor__max_depth': 7, 'regressor__min_child_weight': 1, 'regressor__gamma': 0.2, 'regre
Best RMSE: 3222.51
```

10.4.10 Comprehensive Hyperparameter Tuning

In this step, we expand our search to include a broader set of influential hyperparameters that govern both **model complexity** and **regularization strength** in XGBoost. These include:

- **`learning_rate`**: Controls the contribution of each tree in the ensemble.
- **`max_depth`** and **`min_child_weight`**: Control tree complexity and can help prevent overfitting.
- **`gamma`**: Adds regularization by requiring a minimum loss reduction for a split.
- **`subsample`** and **`colsample_bytree`**: Introduce stochasticity to reduce overfitting by sampling rows and features, respectively.
- **`reg_alpha`** (L1 regularization) and **`reg_lambda`** (L2 regularization): Add penalties to leaf weights to shrink overly complex trees.

Rather than optimizing these parameters independently, we will **tune them together** using a grid search to capture the complex interactions between them. This comprehensive search aims to identify a well-balanced model that generalizes well to unseen data.

This comprehensive tuning process helps us identify the most effective combination of hyperparameters for maximizing predictive performance while minimizing overfitting.

10.4.10.1 Why `GridSearchCV` Is Not a Practical Option

While `GridSearchCV` is a straightforward and exhaustive approach, it can be **extremely time-consuming**, especially when tuning many hyperparameters over multiple values. In our case, the parameter grid includes:

- 3 values for `learning_rate`
- 3 values for `max_depth`
- 3 values for `min_child_weight`
- 3 values for `gamma`
- 3 values each for `subsample` and `colsample_bytree`
- 3 values for `reg_alpha`
- 3 values for `reg_lambda`

This results in a **total of 3 = 6,561 combinations**. With 3-fold cross-validation, this would involve training and evaluating **over 19,000 models**, making it **computationally expensive and inefficient**.

The code is included below if you're curious to try it out — just **uncomment the `.fit()` line** to experience how long it takes.

```
# ===== 4. Comprehensive Hyperparameter Tuning =====
print("\n===== Comprehensive Hyperparameter Tuning Using GridSearchCV=====")
# Define hyperparameter grid
param_grid = {
    'regressor__learning_rate': [0.01, 0.05, 0.1],
    'regressor__max_depth': [3, 5, 7],
    'regressor__min_child_weight': [1, 3, 5],
    'regressor__gamma': [0, 0.1, 0.2],
    'regressor__subsample': [0.8, 0.9, 1.0],
    'regressor__colsample_bytree': [0.8, 0.9, 1.0],
    'regressor__reg_alpha': [0, 1, 5],
    'regressor__reg_lambda': [1, 5, 10]
}

# Create a pipeline for grid search
tune_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(
        random_state=42,
        n_estimators=early_stop_model.best_iteration
    ))
])
```

```python
# Set up grid search with cross-validation
grid_search = GridSearchCV(
    tune_pipeline,
    param_grid,
    cv=3,
    scoring='neg_root_mean_squared_error',
    n_jobs=-1,
    verbose=1
)

# uncomment the line below to run the grid search (it may take a long time)
#grid_search.fit(X_train, y_train)
```

```
===== Comprehensive Hyperparameter Tuning =====
```

10.4.10.2 Smarter Tuning with Optuna or BayesSearchCV

Instead of exhaustively evaluating every combination like `GridSearchCV`, we can use smarter search strategies like:

- **Optuna**: A powerful hyperparameter optimization framework that uses **Tree-structured Parzen Estimators (TPE)** to efficiently explore the search space. It dynamically chooses the next set of hyperparameters to try based on past performance.
- **BayesSearchCV** (from `scikit-optimize`): Implements **Bayesian optimization**, which builds a probabilistic model of the objective function and selects the most promising hyperparameters to try next.

These methods are:

- **More efficient**: They converge to good solutions with far fewer iterations.
- **Flexible**: They support conditional hyperparameter tuning.
- **Scalable**: Much better suited for high-dimensional or expensive-to-evaluate models.

In summary, we commented out the grid search due to its high cost and instead favor more **intelligent, efficient** hyperparameter search methods like `Optuna` or `BayesSearchCV` for practical use.

10.4.10.2.1 `BayesSearchCV` (from `skopt`)

You define the search space using a dictionary where:

- For pipelines and scikit-learn integration, `BayesSearchCV` is simpler
- Keys are hyperparameter names (matching pipeline step names like `'regressor__max_depth'`)
- Values are distributions or discrete ranges from `skopt.space`

> **Key Tip**: Use `Real(..., prior='log-uniform')` for parameters like `learning_rate`, which benefit from exploring small values on a **logarithmic scale**.
> This helps the search algorithm better identify optimal values in ranges where performance is sensitive to small changes (e.g., between 0.01 and 0.1).

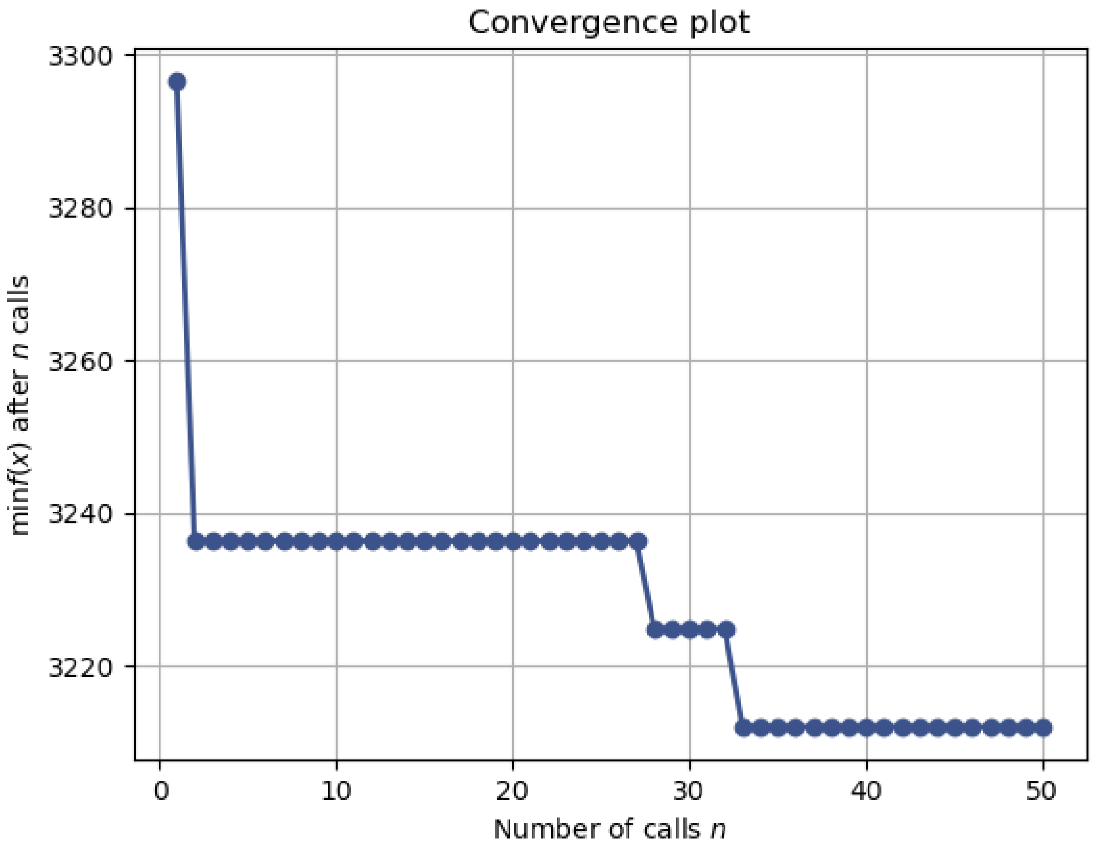
```
# define the search space for Bayesian optimization

search_space = {
    'regressor__learning_rate': Real(0.01, 0.5, prior='uniform'),
    'regressor__max_depth': Integer(3, 7),
    'regressor__min_child_weight': Integer(1, 5),
    'regressor__gamma': Real(0, 0.2),
    'regressor__subsample': Real(0.5, 1.0),
    'regressor__colsample_bytree': Real(0.5, 1.0),
    'regressor__reg_alpha': Real(0, 5),
    'regressor__reg_lambda': Real(1, 10)
}

# Create a pipeline for Bayesian optimization
bayes_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(
        random_state=42,
        n_estimators=early_stop_model.best_iteration
    ))
])
# Set up Bayesian optimization with cross-validation
bayes_search = BayesSearchCV(
    bayes_pipeline,
    search_space,
    cv=3,
    n_iter=50,  # Number of iterations for Bayesian optimization
    scoring='neg_root_mean_squared_error',
    n_jobs=-1,
    verbose=0
```

```python
)
# Perform Bayesian optimization
bayes_search.fit(X_train, y_train)

# Print the best parameters and score
print("\nBest Parameters from Bayesian Optimization:")
print(bayes_search.best_params_)
print("Best Cross-Validation RMSE: {:.2f}".format(-bayes_search.best_score_))
# Evaluate the best model from Bayesian optimization
best_bayes_model = bayes_search.best_estimator_
print("\nBayesian Optimization Model Evaluation:")
bayes_metrics = evaluate_model(best_bayes_model, X_test, y_test)
```

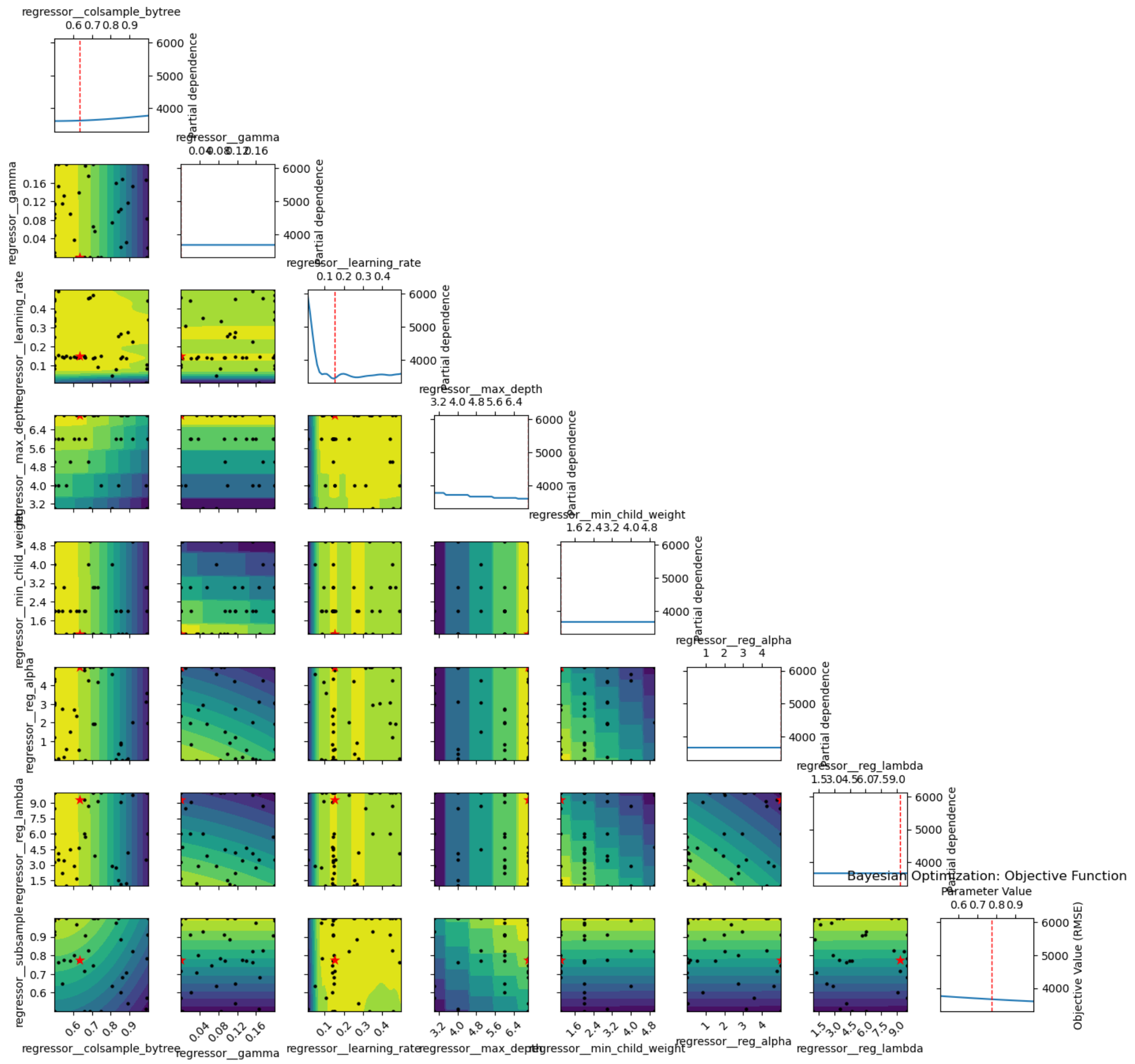
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Fitting 3 folds for each of 1 candidates, totalling 3 fits

Best Parameters from Bayesian Optimization:
OrderedDict({'regressor__colsample_bytree': 0.6350099974437949, 'regressor__gamma': 0.0, 'reg
Best Cross-Validation RMSE: 3212.04

Bayesian Optimization Model Evaluation:
Root Mean Squared Error: 3083.10
R² Score: 0.9676
```

Let's visualize the search results

```
# plot convergence
plot_convergence(bayes_search.optimizer_results_);
```



```
# Plot the objective function
plot_objective(bayes_search.optimizer_results_[0])
plt.title('Bayesian Optimization: Objective Function')
plt.xlabel('Parameter Value')
plt.ylabel('Objective Value (RMSE)')
plt.show()
```

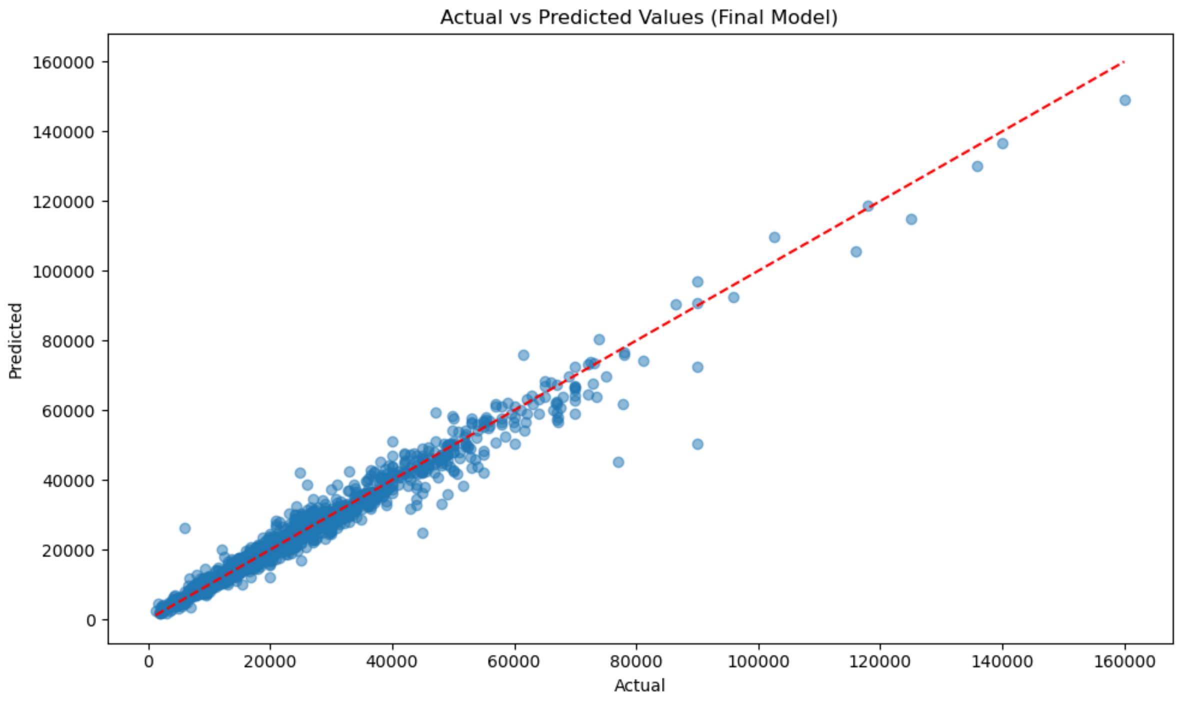


```python
# Create the final model with the best hyperparameters
print("\n===== Final Model with Best Hyperparameters =====")
final_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(
        random_state=42,
        n_estimators=early_stop_model.best_iteration,
        **{k.replace('regressor__', ''): v for k, v in bayes_search.best_params_.items()}
    ))
])
```

```python
# Train the final model
final_pipeline.fit(X_train, y_train)

# Evaluate final model
print("\nFinal Model Evaluation:")
final_metrics = evaluate_model(final_pipeline, X_test, y_test)
```

```
===== Final Model with Best Hyperparameters =====

Final Model Evaluation:
Root Mean Squared Error: 3083.10
R² Score: 0.9676
```

```python
# Display actual vs predicted values for the final model
y_pred = final_pipeline.predict(X_test)
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted Values (Final Model)')
plt.tight_layout()
plt.show()
```



```
# output the feature importance for the final model
final_importance = plot_feature_importance(final_pipeline.named_steps['regressor'],
                                           final_pipeline.named_steps['preprocessor'],
                                           X)
final_importance
```

	Feature	Importance
0	model_ I800	0.109385
1	engineSize	0.085000
2	transmission_Manual	0.080872
3	brand_hyundi	0.045237
4	brand_vw	0.041919
5	brand_ford	0.038523
6	model_ Mustang	0.031646
7	model_ i8	0.030787
8	brand_bmw	0.029788
9	year	0.025983
10	brand_merc	0.024978
11	model_ R8	0.024132
12	model_ S Class	0.024024

	Feature	Importance
13	brand_audi	0.019605
14	model_ X7	0.019070
15	model_ V Class	0.014943
16	model_ X-CLASS	0.013910
17	brand_toyota	0.013511
18	mpg	0.012854
19	model_ X4	0.011403

10.4.10.2.2 Tuning with Optuna

With Optuna, you define the search space inside an objective function using trial suggestions:

```python
import optuna
from sklearn.model_selection import cross_val_score
# Define the objective function for Optuna
def objective(trial):
    # Suggest hyperparameters
    params = {
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.5),
        'max_depth': trial.suggest_int('max_depth', 3, 7),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 5),
        'gamma': trial.suggest_float('gamma', 0, 0.2),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 0, 5),
        'reg_lambda': trial.suggest_float('reg_lambda', 1, 10),
        'random_state': 42,
        'n_estimators': 1000
    }

    # Define the model
    model = XGBRegressor(**params)

    # Optionally: wrap in pipeline if preprocessing is needed
    pipeline = Pipeline([
        ('preprocessor', preprocessor),  # assumed to be defined earlier
        ('regressor', model)
    ])

    # Evaluate with cross-validation
    score = cross_val_score(pipeline, X_train, y_train, cv=3, scoring='neg_root_mean_squared
    return score  # Maximize negative RMSE (i.e., minimize RMSE)

# Run the Optuna study
study = optuna.create_study(direction='maximize')  # maximizing negative RMSE
study.optimize(objective, n_trials=50, timeout=600)

# Display the best result
print("Best trial:")
print(f"  RMSE (CV): {-study.best_value:.4f}")
print("  Best hyperparameters:")
for key, value in study.best_params.items():
    print(f"    {key}: {value}")
```

```
[I 2025-05-14 04:47:06,217] A new study created in memory with name: no-name-59242cf6-161a-4
[I 2025-05-14 04:47:07,258] Trial 0 finished with value: -3162.7913411458335 and parameters:
[I 2025-05-14 04:47:09,975] Trial 1 finished with value: -3368.8380533854165 and parameters:
[I 2025-05-14 04:47:10,925] Trial 2 finished with value: -3432.02392578125 and parameters: {
[I 2025-05-14 04:47:12,191] Trial 3 finished with value: -3344.6248372395835 and parameters:
[I 2025-05-14 04:47:13,975] Trial 4 finished with value: -3270.3804524739585 and parameters:
[I 2025-05-14 04:47:16,075] Trial 5 finished with value: -3520.0126953125 and parameters: {'
[I 2025-05-14 04:47:18,141] Trial 6 finished with value: -3230.2576497395835 and parameters:
[I 2025-05-14 04:47:20,158] Trial 7 finished with value: -3265.4407552083335 and parameters:
[I 2025-05-14 04:47:22,225] Trial 8 finished with value: -3521.05517578125 and parameters: {
[I 2025-05-14 04:47:23,242] Trial 9 finished with value: -3271.8055013020835 and parameters:
[I 2025-05-14 04:47:25,226] Trial 10 finished with value: -3261.6067708333335 and parameters
[I 2025-05-14 04:47:27,859] Trial 11 finished with value: -3377.5233561197915 and parameters
[I 2025-05-14 04:47:30,092] Trial 12 finished with value: -3359.8211263020835 and parameters
[I 2025-05-14 04:47:32,225] Trial 13 finished with value: -3389.99609375 and parameters: {'l
[I 2025-05-14 04:47:33,783] Trial 14 finished with value: -3303.03955078125 and parameters:
[I 2025-05-14 04:47:36,609] Trial 15 finished with value: -3350.712158203125 and parameters:
[I 2025-05-14 04:47:39,729] Trial 16 finished with value: -3751.3375651041665 and parameters
[I 2025-05-14 04:47:41,442] Trial 17 finished with value: -3220.991943359375 and parameters:
[I 2025-05-14 04:47:42,826] Trial 18 finished with value: -3196.406494140625 and parameters:
[I 2025-05-14 04:47:43,876] Trial 19 finished with value: -3185.7190755208335 and parameters
[I 2025-05-14 04:47:45,326] Trial 20 finished with value: -3203.70068359375 and parameters:
[I 2025-05-14 04:47:46,993] Trial 21 finished with value: -3228.9916178385415 and parameters
[I 2025-05-14 04:47:48,459] Trial 22 finished with value: -3219.378662109375 and parameters:
[I 2025-05-14 04:47:49,809] Trial 23 finished with value: -3211.786865234375 and parameters:
[I 2025-05-14 04:47:51,676] Trial 24 finished with value: -3334.4794921875 and parameters: {
[I 2025-05-14 04:47:53,576] Trial 25 finished with value: -3267.5618489583335 and parameters
[I 2025-05-14 04:47:54,926] Trial 26 finished with value: -3202.6060384114585 and parameters
```

```
[I 2025-05-14 04:47:56,627] Trial 27 finished with value: -3199.9505208333335 and parameters
[I 2025-05-14 04:47:58,059] Trial 28 finished with value: -3233.1486002604165 and parameters
[I 2025-05-14 04:47:59,893] Trial 29 finished with value: -3205.3538411458335 and parameters
[I 2025-05-14 04:48:01,494] Trial 30 finished with value: -3383.5137532552085 and parameters
[I 2025-05-14 04:48:03,243] Trial 31 finished with value: -3167.3258463541665 and parameters
[I 2025-05-14 04:48:05,445] Trial 32 finished with value: -3281.510986328125 and parameters:
[I 2025-05-14 04:48:07,111] Trial 33 finished with value: -3204.1426595052085 and parameters
[I 2025-05-14 04:48:08,476] Trial 34 finished with value: -3220.5049641927085 and parameters
[I 2025-05-14 04:48:10,219] Trial 35 finished with value: -3141.7112630208335 and parameters
[I 2025-05-14 04:48:12,727] Trial 36 finished with value: -3424.205810546875 and parameters:
[I 2025-05-14 04:48:14,696] Trial 37 finished with value: -3118.7006022135415 and parameters
[I 2025-05-14 04:48:16,481] Trial 38 finished with value: -3122.8831380208335 and parameters
[I 2025-05-14 04:48:18,515] Trial 39 finished with value: -3139.6028645833335 and parameters
[I 2025-05-14 04:48:21,326] Trial 40 finished with value: -3165.2281901041665 and parameters
[I 2025-05-14 04:48:22,927] Trial 41 finished with value: -3153.988525390625 and parameters:
[I 2025-05-14 04:48:25,194] Trial 42 finished with value: -3138.40869140625 and parameters:
[I 2025-05-14 04:48:27,181] Trial 43 finished with value: -3634.5953776041665 and parameters
[I 2025-05-14 04:48:28,967] Trial 44 finished with value: -3160.775146484375 and parameters:
[I 2025-05-14 04:48:31,730] Trial 45 finished with value: -3168.1395670572915 and parameters
[I 2025-05-14 04:48:34,529] Trial 46 finished with value: -3193.5437825520835 and parameters
[I 2025-05-14 04:48:37,245] Trial 47 finished with value: -3228.6897786458335 and parameters
[I 2025-05-14 04:48:40,511] Trial 48 finished with value: -3211.089111328125 and parameters:
[I 2025-05-14 04:48:42,711] Trial 49 finished with value: -3200.5746256510415 and parameters

Best trial:
  RMSE (CV): 3118.7006
  Best hyperparameters:
    learning_rate: 0.050933621089207015
    max_depth: 5
    min_child_weight: 2
    gamma: 0.17859904779145075
    subsample: 0.687046200984366
    colsample_bytree: 0.5234986611495844
    reg_alpha: 2.814493606788762
    reg_lambda: 4.8395816535714555
```

Let's visualize the result

```python
import optuna.visualization as vis

fig1 = vis.plot_optimization_history(study)
fig1.show()
```

```
fig2 = vis.plot_param_importances(study)
fig2.show()

fig3 = vis.plot_slice(study)
fig3.show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

10.4.10.3 After Training: Analyze and Refine

Once the tuning is complete, **don't forget to visualize the search results** to understand how different hyperparameters affected performance. This helps you:

- Identify which parameters had the most impact.
- Spot trends (e.g., performance plateaus or sharp drop-offs).
- Detect boundary effects (e.g., best values lie at the edge of the current search space).

> **Key Tip**: If the best values are near the boundary of your current search space, consider **fine-tuning the search space** and re-running the optimization.

You can visualize results using:

- `Optuna`'s built-in plots like `plot_optimization_history()` and `plot_param_importances()`.
- `BayesSearchCV`'s `cv_results_` attribute to create custom plots using `pandas` or `seaborn`.

Effective tuning is often **iterative** — let the data guide you!

10.5 XGBoost for Imbalanced Classification

10.5.1 Common Strategies Across Libraries

Imbalanced classification arises when one class is significantly underrepresented—common in applications like **fraud detection**, **rare disease diagnosis**, and **anomaly detection**.

- **Use Better Evaluation Metrics**
  Avoid relying on accuracy. Instead, use metrics that reflect class imbalance, such as:
  - **F1-score**
  - **AUC-PR** (Area Under the Precision-Recall Curve)
  - **Matthews Correlation Coefficient (MCC)**
- **Threshold Tuning**
  Adjust the **decision threshold** to balance between precision and recall based on your use case.
- **Stratified Sampling**
  When splitting the dataset (for training/validation or cross-validation), use **stratified sampling** to maintain the class distribution in each fold.

10.5.2 Handling Class Imbalance with `scale_pos_weight` in XGBoost

While XGBoost (and other gradient boosting libraries) can perform well on imbalanced datasets, models can become biased toward the **majority class** if no corrective strategies are used. One of the most effective built-in solutions in XGBoost is the `scale_pos_weight` parameter.

10.5.2.1 What Does `scale_pos_weight` Do?

The `scale_pos_weight` parameter adjusts the **relative importance** of positive class examples (`label = 1`) by scaling their gradients and Hessians during training.

- A higher value places **more penalty on misclassifying positive samples**
- This encourages the model to **focus more on the minority class**, helping improve recall and F1-score

10.5.2.2 When to Use It

Use `scale_pos_weight > 1` when:

- The dataset is **heavily imbalanced**
- You care more about the **positive class performance** (e.g., improving **recall**, **precision**, or **F1-score**)

10.5.3 How to Set It

A commonly used heuristic:

$$\text{scale\_pos\_weight} = \frac{\text{Number of negative samples}}{\text{Number of positive samples}}$$

This provides a **balanced gradient contribution** during training and serves as a good **starting point**. You can further fine-tune this value via cross-validation for optimal performance.

Note: While `scale_pos_weight` adjusts learning behavior internally, you should still monitor metrics like **AUC-PR**, **F1-score**, or **recall** to ensure it's improving your model's performance on the minority class.

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

```
print(diabetes_train.shape, diabetes_test.shape)
diabetes_train.head()
```

```
(614, 9) (154, 9)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	2	88	74	19	53	29.0	0.229	22
1	2	129	84	0	0	28.0	0.284	27
2	0	102	78	40	90	34.5	0.238	24
3	0	123	72	0	0	36.3	0.258	52
4	1	144	82	46	180	46.1	0.335	46

```
# check the outcome of the distribution in the training set and test set
print(diabetes_train['Outcome'].value_counts(normalize=True))
print(diabetes_test['Outcome'].value_counts(normalize=True))
```

```
Outcome
0    0.662866
1    0.337134
Name: proportion, dtype: float64
Outcome
0    0.603896
1    0.396104
Name: proportion, dtype: float64
```

```python
# Data Preprocessing
X_diabetes = diabetes_train.drop(columns=['Outcome'])
y_diabetes = diabetes_train['Outcome']
X_diabetes_test = diabetes_test.drop(columns=['Outcome'])
y_diabetes_test = diabetes_test['Outcome']

# define categorical and numerical columns
categorical_cols_diabetes = X_diabetes.select_dtypes(include=['object']).columns.tolist()
numerical_cols_diabetes = X_diabetes.select_dtypes(exclude=['object']).columns.tolist()

# define the preprocessor, passing the numerical and categorical columns
preprocessor_diabetes = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numerical_cols_diabetes),  # no scaling
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols_diabetes)
    ]
)

# Create a pipeline for the diabetes dataset
diabetes_pipeline = Pipeline([
    ('preprocessor', preprocessor_diabetes),
    ('regressor', XGBClassifier(random_state=42))
])

# Train the pipeline on the diabetes dataset
diabetes_pipeline.fit(X_diabetes, y_diabetes)
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num', 'passthrough',
                                                  ['Pregnancies', 'Glucose',
                                                   'BloodPressure',
                                                   'SkinThickness', 'Insulin',
                                                   'BMI',
                                                   'DiabetesPedigreeFunction',
                                                   'Age']),
                                                 ('cat',
                                                  OneHotEncoder(handle_unknown='ignore'),
                                                  [])])),
                ('regressor',
                 XGBClassifier(base_score=None, booster=None, callbacks=None,
                               colsample_bylevel=None, colsample_...
                               feature_types=None, feature_weights=None,
                               gamma=None, grow_policy=None,
                               importance_type=None,
                               interaction_constraints=None, learning_rate=None,
                               max_bin=None, max_cat_threshold=None,
                               max_cat_to_onehot=None, max_delta_step=None,
                               max_depth=None, max_leaves=None,
                               min_child_weight=None, missing=nan,
                               monotone_constraints=None, multi_strategy=None,
                               n_estimators=None, n_jobs=None,
                               num_parallel_tree=None, ...))])
```

```
# make predictions on the test set

print("\n===== Test Performance with Default Setting =====")
y_diabetes_pred = diabetes_pipeline.predict(X_diabetes_test)
# evaluate the model in terms of accuracy, precision, recall, and f1-score
accuracy = np.mean(y_diabetes_pred == y_diabetes_test)
print(f"Accuracy on Diabetes Test Set: {accuracy:.2f}")
precision = precision_score(y_diabetes_test, y_diabetes_pred)
print(f"Precision on Diabetes Test Set: {precision:.2f}")
recall = recall_score(y_diabetes_test, y_diabetes_pred)
print(f"Recall on Diabetes Test Set: {recall:.2f}")
f1 = f1_score(y_diabetes_test, y_diabetes_pred)
print(f"F1 Score on Diabetes Test Set: {f1:.2f}")
```

```
===== Test Performance with Default Setting =====
Accuracy on Diabetes Test Set: 0.73
Precision on Diabetes Test Set: 0.68
Recall on Diabetes Test Set: 0.59
F1 Score on Diabetes Test Set: 0.63
```

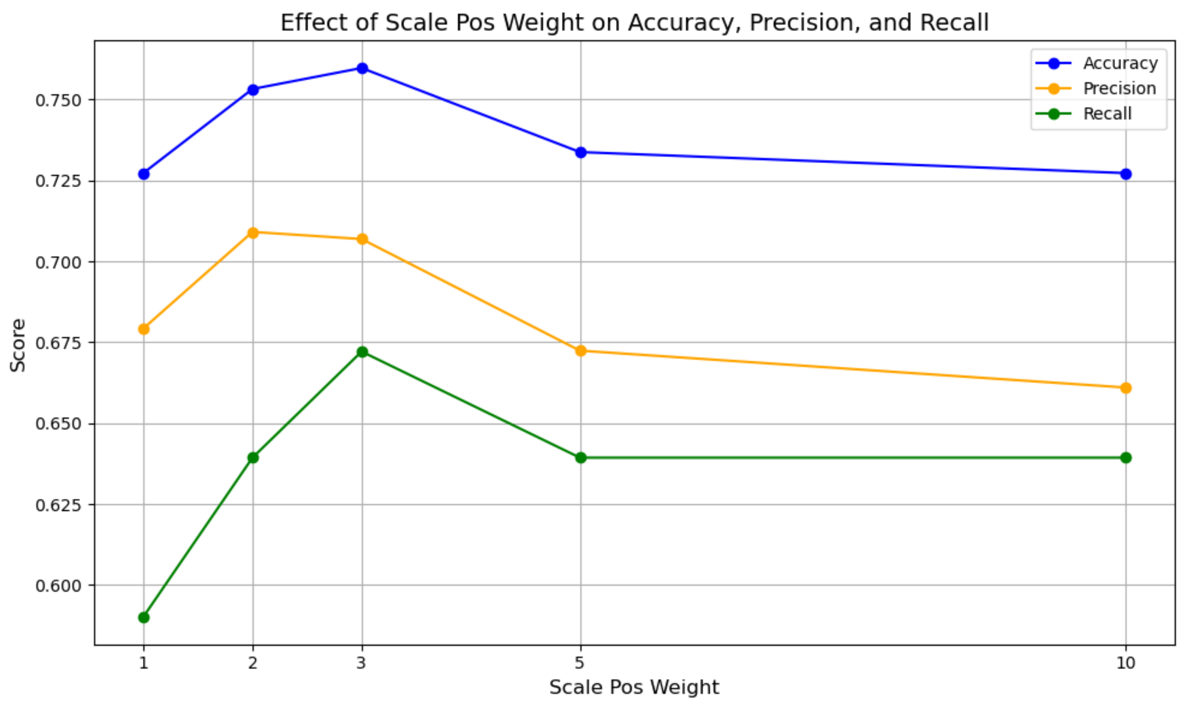
10.5.4 Using `scale_pos_weight`

```
neg = (diabetes_train['Outcome'] == 0).sum()
pos = (diabetes_train['Outcome'] == 1).sum()
baseline_ratio = neg / pos

# set a small grid for scale_pos_weight
scale_pos_weight = [1, 2, 3, 5, 10]
```

```
# Create a new pipeline for the diabetes dataset
diabetes_pipeline_scale = Pipeline([
    ('preprocessor', preprocessor_diabetes),
    ('regressor', XGBClassifier(random_state=42))
])

# Store results for comparison
diabetes_results = []
for i, weight in enumerate(scale_pos_weight):
    print(f"\nScale Pos Weight Test {i+1}:")
    print(f"scale_pos_weight = {weight}")

    # Update the pipeline with the scale_pos_weight parameter
    diabetes_pipeline_scale.set_params(regressor__scale_pos_weight=weight)

    # Train and evaluate
    diabetes_pipeline_scale.fit(X_diabetes, y_diabetes)
    y_diabetes_pred = diabetes_pipeline_scale.predict(X_diabetes_test)

    # Evaluate the model
    accuracy = np.mean(y_diabetes_pred == y_diabetes_test)
    precision = precision_score(y_diabetes_test, y_diabetes_pred)
    recall = recall_score(y_diabetes_test, y_diabetes_pred)
    f1 = f1_score(y_diabetes_test, y_diabetes_pred)

    # Store results
    diabetes_results.append({
        'test': i+1,
        'scale_pos_weight': weight,
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1
    })
# Create a DataFrame for the results
diabetes_results_df = pd.DataFrame(diabetes_results)

# Find the best scale_pos_weight based on F1 score
best_f1_idx = diabetes_results_df['f1'].idxmax()
best_scale_pos_weight = diabetes_results_df.loc[best_f1_idx, 'scale_pos_weight']
print("\n===== Test Performance with Best Scale Pos Weight: =====")
print(best_scale_pos_weight)
print("Best F1 Score: {:.2f}".format(diabetes_results_df['f1'].max()))
# find the accuracy, precision, and recall for the best scale_pos_weight
best_accuracy = diabetes_results_df.loc[best_f1_idx, 'accuracy']
best_precision = diabetes_results_df.loc[best_f1_idx, 'precision']
best_recall = diabetes_results_df.loc[best_f1_idx, 'recall']
print(f"Best Accuracy: {best_accuracy:.2f}")
print(f"Best Precision: {best_precision:.2f}")
print(f"Best Recall: {best_recall:.2f}")
```

```
Scale Pos Weight Test 1:
scale_pos_weight = 1

Scale Pos Weight Test 2:
scale_pos_weight = 2

Scale Pos Weight Test 3:
scale_pos_weight = 3

Scale Pos Weight Test 4:
scale_pos_weight = 5

Scale Pos Weight Test 5:
scale_pos_weight = 10

Best Scale Pos Weight:
3
Best F1 Score: 0.69
Best Accuracy: 0.76
Best Precision: 0.71
Best Recall: 0.67
```

```
# Plot the results for accuracy, precision, and recall with different colors
plt.figure(figsize=(10, 6))
plt.plot(diabetes_results_df['scale_pos_weight'], diabetes_results_df['accuracy'], marker='o
plt.plot(diabetes_results_df['scale_pos_weight'], diabetes_results_df['precision'], marker='
plt.plot(diabetes_results_df['scale_pos_weight'], diabetes_results_df['recall'], marker='o',
plt.xlabel('Scale Pos Weight', fontsize=12)
plt.ylabel('Score', fontsize=12)
plt.title('Effect of Scale Pos Weight on Accuracy, Precision, and Recall', fontsize=14)
plt.grid(True)
plt.xticks(scale_pos_weight)
plt.legend()
plt.tight_layout()
plt.show()
```



Tuning `scale_pos_weight` changes the **model's internal learning dynamics**, not just the decision threshold. This allows the model to adjust how it learns from imbalanced data.

In this case, as `scale_pos_weight` increases from 1 to 3:

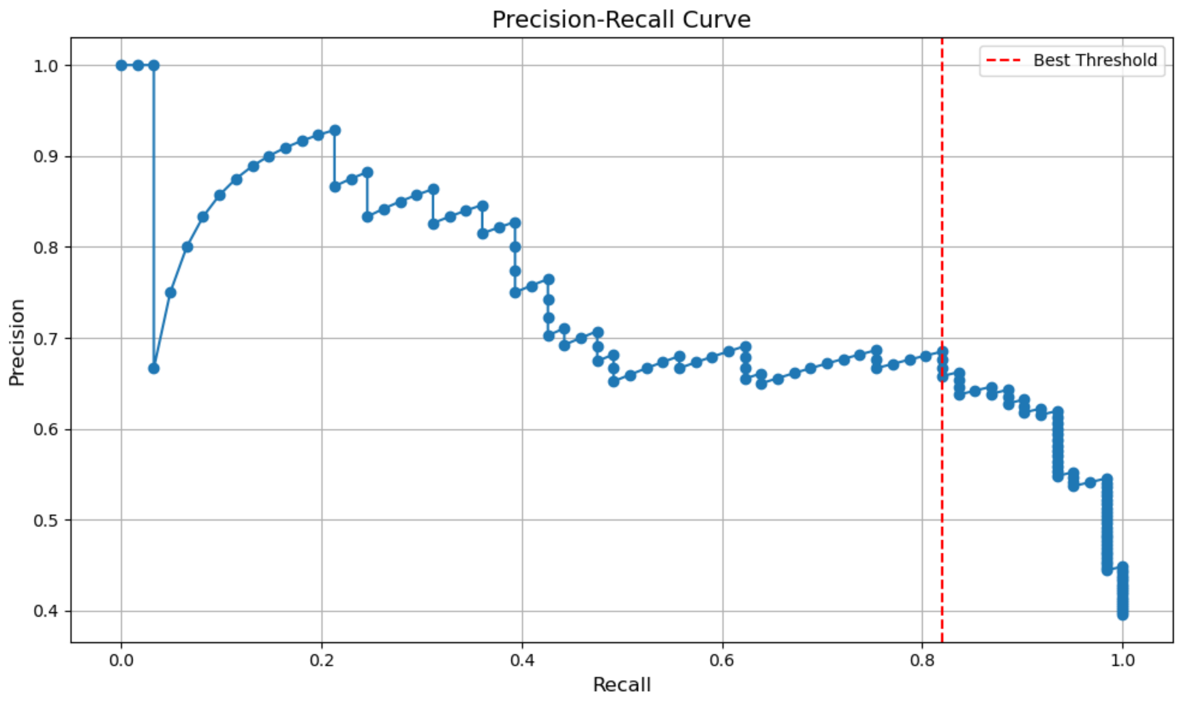
- XGBoost starts giving **more importance to the minority (positive) class**.
- The model becomes better at identifying **true positives** → **Recall increases**
- Simultaneously, it avoids more **false positives** → **Precision increases**

This indicates the model was previously **underperforming on the positive class**, and that moderate rebalancing (e.g., `scale_pos_weight = 3`) helped improve **both recall and precision** — something that can happen when the model is initially biased toward the majority class.

10.5.5 Threshold adjustment

```
# get prediction probabilities
y_diabetes_pred_proba = diabetes_pipeline.predict_proba(X_diabetes_test)[:, 1]

# plot the precision-recall curve
precision, recall, thresholds = precision_recall_curve(y_diabetes_test, y_diabetes_pred_proba

# final threshold
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-9) # avoid division by zero
best_f1_threshold = thresholds[np.argmax(f1_scores)]
best_threshold = np.round(best_f1_threshold, 2)
print(f"Best Threshold for F1 Score: {best_threshold:.2f}")

# adjust the threshold for the predictions
y_diabetes_pred_adjusted = (y_diabetes_pred_proba >= best_f1_threshold).astype(int)
# evaluate the model with the adjusted threshold
accuracy_adjusted = np.mean(y_diabetes_pred_adjusted == y_diabetes_test)
print(f"Accuracy with Adjusted Threshold: {accuracy_adjusted:.2f}")
precision_adjusted = precision_score(y_diabetes_test, y_diabetes_pred_adjusted)
print(f"Precision with Adjusted Threshold: {precision_adjusted:.2f}")
recall_adjusted = recall_score(y_diabetes_test, y_diabetes_pred_adjusted)
print(f"Recall with Adjusted Threshold: {recall_adjusted:.2f}")
# plot the precision-recall curve
plt.figure(figsize=(10, 6))
plt.plot(recall, precision, marker='o')
plt.xlabel('Recall', fontsize=12)
plt.ylabel('Precision', fontsize=12)
plt.title('Precision-Recall Curve', fontsize=14)
plt.grid(True)
plt.axvline(x=recall[np.argmax(f1_scores)], color='red', linestyle='--', label='Best Thresho
plt.legend()
plt.tight_layout()
plt.show()
```

```
Best Threshold for F1 Score: 0.19
Accuracy with Adjusted Threshold: 0.78
Precision with Adjusted Threshold: 0.68
Recall with Adjusted Threshold: 0.82
```



10.5.6 Alternative Method: Custom Instance Weights (`sample_weight`)

You can assign **custom weights to individual training samples** using the `sample_weight` parameter in `fit()` (scikit-learn API) or `weight` in `DMatrix` (native API). This gives you **fine-grained control** over how much each sample contributes to the loss and gradient during training.

Example (scikit-learn API):

```python
from xgboost import XGBClassifier
import numpy as np

# Assign higher weights to positive class
weights = np.where(y_train == 1, 5, 1)

model = XGBClassifier()
model.fit(X_train, y_train, sample_weight=weights)
```

When to Use This:

- When you want more flexibility than scale_pos_weight allows

- When the imbalance is complex (e.g., multi-class or cost-sensitive)
- When you want to incorporate domain knowledge into weight assignments

10.5.7 `scale_pos_weight` vs. `sample_weight`

Feature	`scale_pos_weight`	`sample_weight`
Applies to	Entire positive class (`label = 1`)	Individual samples
How it works	Multiplies gradients and Hessians of the positive class during training	Directly scales the loss function per instance
Use case	Binary classification with class imbalance	Any situation needing custom weighting (e.g., multi-class, domain-driven)
Flexibility	One global weight value	Full per-sample control
Where to set	`scale_pos_weight` parameter in `XGBClassifier` or `DMatrix`	`sample_weight` in `.fit()` or `weight=` in `DMatrix`

10.6 Resources for Learning XGBoost

The foundational paper is:

XGBoost: A Scalable Tree Boosting System
Authors: Tianqi Chen and Carlos Guestrin
Conference: KDD 2016
Link to paper (PDF)
DOI: 10.1145/2939672.2939785

XGBoost is a relatively recent algorithm (2016), and thus not yet included in many standard textbooks. Below are helpful learning resources:

- Documentation
- Slides by Tianqi Chen
- Reference Paper
- Video by Tianqi Chen (author)
- StatQuest Video Explanation

11 LightGBM and CatBoost

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

Gradient boosting is one of the most powerful techniques for **structured/tabular data**, often serving as the **go-to choice for tabular data tasks in machine learning competitions**.

In the previous chapter, we explored **XGBoost** in detail—covering its **optimization objective**, **regularization techniques**, **split finding algorithms**, and its role as a cornerstone in modern **tabular modeling**.

While **XGBoost** is highly effective, other libraries have introduced innovations to address challenges like scalability and categorical feature handling. In this chapter, we focus on two **advanced gradient boosting libraries** and their **key innovations**:

- **LightGBM**: Developed by Microsoft, **LightGBM** is optimized for **speed and scalability**. It introduces **Gradient-based One-Side Sampling (GOSS)**, which prioritizes instances with larger gradients for faster training, and **Exclusive Feature Bundling (EFB)**, which reduces memory usage by grouping mutually exclusive features. **LightGBM** also supports **categorical features** efficiently, making it ideal for **large datasets and high-dimensional features**.
- **CatBoost**: Created by Yandex, **CatBoost** excels in its **optimized support for categorical features** through advanced target-based encoding. It uses **ordered boosting** to prevent prediction shift and reduce overfitting, often performing well with **minimal tuning** on datasets rich in **categorical variables**.

11.1 What They Share with XGBoost

While **LightGBM** and **CatBoost** introduce unique innovations, they build on the same foundational principles as **XGBoost**:

- They use a **similar objective function structure** (loss plus regularization) to balance model fit and complexity.
- They apply a **second-order Taylor approximation** for efficient optimization of the loss function.

- They support **histogram-based split-finding algorithms** to speed up training, with **LightGBM** particularly optimized for this approach.
- They support **parallel tree building**, significantly accelerating training compared to traditional gradient boosting.
- They provide **native support for categorical encoding**, reducing the need for pre-processing (e.g., one-hot encoding), with **XGBoost** introducing this feature starting in version 1.5.0.

11.2 LightGBM

11.2.1 What is LightGBM?

LightGBM (Light Gradient Boosting Machine) is a high-performance gradient boosting framework developed by Microsoft in 2017. Designed for **speed and scalability**, it is typically faster than **XGBoost** due to its optimized algorithms, with accuracy that is generally comparable but may require tuning. LightGBM excels in:

- **Handling large-scale datasets** with many rows and features
- **Achieving high speed and memory efficiency** through innovations like **Gradient-based One-Side Sampling (GOSS)**, **Exclusive Feature Bundling (EFB)**, and **histogram-based splitting**

Like **XGBoost** and **CatBoost**, it supports **native categorical encoding** and **parallel tree building**, enhancing its efficiency for tabular data tasks. See the LightGBM paper for details on its algorithmic innovations and performance benchmarks.

11.2.2 What Makes LightGBM Lighting Fast?

LightGBM often outperforms XGBoost in **training speed** and **memory efficiency**, thanks to several key innovations:

11.2.2.1 Leaf-Wise Tree Growth

- LightGBM splits the **leaf with the largest potential loss reduction**, unlike XG-Boost's **level-wise** approach.
- This leads to **lower loss per tree**, making learning more efficient — though it may **overfit** without proper regularization.
- Main controls:
  - `num_leaves`: primary control for tree complexity
  - `max_depth`: optional constraint to prevent overfitting

11.2.2.2 GOSS (Gradient-based One-Side Sampling)

- GOSS improves speed by:
  - **Retaining all instances with large gradients** (i.e., high error)
  - **Randomly sampling those with small gradients**
- This reduces the dataset size while maintaining accurate split decisions.

In gradient boosting, the tree is fit to the **negative gradient** of the loss:

$$r_m = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$$

Observations with larger gradients have more influence on reducing the loss — GOSS prioritizes those. This approach reduces the number of data points processed per iteration, speeding up training while preserving important information.

11.2.2.3 Exclusive Feature Bundling (EFB)

- **EFB** reduces memory usage and accelerates training by bundling **mutually exclusive** features (i.e., features that rarely have non-zero values simultaneously) in **high-dimensional sparse feature spaces**.
- This is particularly effective for datasets with **many categorical variables** or **one-hot encoded features**, avoiding the memory overhead of one-hot encoding and complementing LightGBM's **native categorical support**.

Example:
The table below shows two mutually exclusive features, `feature1` and `feature2`, bundled into a single `feature_bundle` by assigning distinct value ranges (e.g., 1–4 for `feature1`, 5–6 for `feature2`):

`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

- **Hyperparameter for EFB**:

  - `enable_bundle`: Enabled by default to activate automatic bundling.
  - `max_conflict_rate`: Controls the maximum conflict rate for bundling (default: 0.0, no conflicts allowed); adjust (e.g., 0.1) to allow minor overlaps.

This approach reduces the number of features processed per iteration, speeding up training while preserving important information.

Combined with **GOSS**, **EFB** makes **LightGBM** especially well-suited for **large-scale, sparse, tabular datasets**, offering **speed and scalability** while maintaining comparable accuracy with proper tuning.

11.2.3 Using LightGBM

Although **LightGBM is not part of Scikit-learn**, it provides a **Scikit-learn-compatible API** through the `lightgbm.sklearn` module. This allows you to use LightGBM models seamlessly with Scikit-learn tools such as `Pipeline`, `GridSearchCV`, and `cross_val_score`.

The main classes are:

- `LGBMRegressor`: for regression tasks

- `LGBMClassifier`: for classification tasks

To install the package:

```
pip install lightgbm
```

> **Note:** LightGBM is a separate library, not part of Scikit-learn, but it provides a **Scikit-learn-compatible API** via `LGBMClassifier` and `LGBMRegressor`.
> This makes it easy to integrate LightGBM models into Scikit-learn workflows such as `Pipeline`, `GridSearchCV`, and `cross_val_score`.

11.2.3.1 Core LightGBM Hyperparameters

Core Tree Structure:

- `num_leaves`: Maximum number of leaves (terminal nodes) per tree.
- `min_data_in_leaf`: Minimum number of data points required in a leaf.
- `max_depth`: Maximum depth of a tree (used to control overfitting).

Learning Control and Regularization:

- `learning_rate ( )`: Shrinks the contribution of each tree.
- `n_estimators`: Number of boosting rounds.

- `lambda_l1` / `lambda_l2`: L1 and L2 regularization on leaf weights.
- `min_gain_to_split`: Minimum loss reduction required to make a further split (structure regularization).

Data Handling:

- `feature_fraction`: Fraction of features randomly sampled for each tree (a.k.a. `colsample_bytree` in XGBoost).
- `bagging_fraction`: Fraction of data randomly sampled for each iteration.
- `bagging_freq`: Frequency (in iterations) to perform bagging.
- `categorical_feature`: Specifies which features are categorical (enables native handling).

Speed vs. Accuracy Trade-offs:

- `max_bin`: Number of bins used to bucket continuous features.
- `data_sample_strategy` : `bagging` or `goss`
- `top_rate` *(goss only)*: Fraction of instances with the largest gradients to keep.
- `other_rate` *(goss only)*: Fraction of small-gradient instances to randomly sample. - `enable_bundle`: set this to true to spped up the training for sparse datasets

Optimization Control:

- `boosting`: Type of boosting algorithm (`gbdt`, `dart`, `rf`, etc.).
- `early_stopping_rounds`: Stops training if the validation score doesn't improve over a set number of rounds.

Imbalanced Data

- `scale_pos_weight`: Manually sets the weight for the positive class in binary classification.
- `is_unbalance`: Automatically adjusts class weights based on the training data distribution.

  These two options are **mutually exclusive** — use **only one**. If both are set, `scale_pos_weight` takes priority.

For full details and advanced options, see the LightGBM Parameters Guide.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import root_mean_squared_error, r2_score, accuracy_score, precision_sco
from xgboost import XGBRegressor, XGBClassifier
import lightgbm as lgb
import seaborn as sns

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

We'll continue to use the same datasets that we have been using throughout the course.

```python
# 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

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

# 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()

# convert the categorical columns to category type
for col in categorical_feature:
    X[col] = X[col].astype('category')


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

11.2.3.2 Building a Baseline Model Using LightGBM's Native Categorical Feature Support

LightGBM provides **built-in support for handling categorical features**, eliminating the need for manual encoding (like one-hot or ordinal encoding). By directly passing categorical column names or indices to the model, LightGBM can internally apply efficient encoding and optimized split finding for categorical variables.

In this section, we'll use this native capability to **quickly build a baseline model**, taking advantage of LightGBM's efficiency with structured data that includes categorical columns.

This baseline model serves as a **starting point** for comparison against more advanced tuning

```
%%time
# ===== 1. Baseline Model =====
print("\n===== Baseline LightGBM Model =====")
# Initialize the LightGBM regressor
model = lgb.LGBMRegressor(random_state=42)

# Train the model with categorical features specified
model.fit(
    X_train,
    y_train,
    categorical_feature=categorical_feature
)

# Predict on the test set
y_pred = model.predict(X_test)

# Calculate evaluation metrics
rmse = root_mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Output results
print(f"Test RMSE: {rmse:.4f}")
print(f"Test R²: {r2:.4f}")
```

```
===== Baseline LightGBM Model =====
Test RMSE: 3680.8999
Test R²: 0.9538
CPU times: total: 875 ms
Wall time: 82.6 ms
```

11.2.3.3 Enabling GOSS and EFB in LightGBM

11.2.3.3.1 GOSS is not enabled by default.

The default boosting type is `gbdt` (traditional Gradient Boosting Decision Tree), which uses all data instances for each iteration without sampling based on gradients.

To use GOSS, you must explicitly set the `boosting_type` parameter to `goss` in the model configuration. When you do this, LightGBM uses GOSS with default values for its specific hyperparameters:

- `top_rate`: 0.2 (keeps 20% of instances with large gradients)
- `other_rate`: 0.1 (randomly samples 10% of instances with small gradients)

11.2.3.3.2 EFB is enabled by default

```
enable_bundle = True
```

This optimization reduces dimensionality by bundling mutually exclusive sparse features, such as those resulting from one-hot encoding.

Note: In our car dataset, the data size is small and there are only a few categorical features, so these optimizations may not have a noticeable impact. However, for large-scale datasets with many categorical features, enabling GOSS and EFB is highly recommended to improve training efficiency and reduce memory usage.

```
%%time
# ===== 2. LightGBM with GOSS Sampling =====
print("\n===== LightGBM with GOSS Sampling =====")

# Initialize the LightGBM regressor with GOSS
model_goss = lgb.LGBMRegressor(
    boosting_type='goss',
    random_state=42
)

# Train the model with categorical features specified
model_goss.fit(
    X_train,
    y_train,
    categorical_feature=categorical_feature
)
```

```
# Predict on the test set
y_pred_goss = model_goss.predict(X_test)

# Calculate evaluation metrics
rmse_goss = root_mean_squared_error(y_test, y_pred_goss)
r2_goss = r2_score(y_test, y_pred_goss)

# Output results
print(f"Test RMSE (GOSS): {rmse_goss:.4f}")
print(f"Test R² (GOSS): {r2_goss:.4f}")
```

```
===== LightGBM with GOSS Sampling =====
Test RMSE (GOSS): 3510.7726
Test R² (GOSS): 0.9580
CPU times: total: 766 ms
Wall time: 79.6 ms
```

11.2.3.4 Tuning `top_rate` and `other_rate` in GOSS

Even with this small dataset, we observed a **shorter execution time** and a **slight improvement in performance** using GOSS.

The default settings are reasonable for many datasets. To leverage **GOSS** more effectively, optimize performance by tuning **top_rate** (e.g., 0.1, 0.2, 0.3, 0.4) and **other_rate** (e.g., 0.05, 0.1, 0.15, 0.2) using cross-validation, especially for large or noisy datasets.

> **Note:** When using `boosting_type='goss'`, LightGBM requires that
> `top_rate + other_rate   1.0`
> This constraint ensures that the combined sample used for training does not exceed the size of the full dataset.

```
# tuning the top_rate and other_rate parameters
# Initialize the LightGBM regressor with GOSS
model_goss_tune = lgb.LGBMRegressor(
    boosting_type='goss',
    random_state=42
)
# Define the parameter grid for tuning
param_grid = {
    'top_rate': Real(0.1, 0.6, prior='uniform'),
    'other_rate': Real(0.1, 0.4, prior='uniform'),
}
# Initialize the BayesSearchCV object
opt = BayesSearchCV(
    model_goss_tune,
    param_grid,
    n_iter=10,
    cv=3,
    n_jobs=-1,
    random_state=42
)
# Fit the model
opt.fit(
    X_train,
    y_train,
    categorical_feature=categorical_feature
)
# the best parameters
print("Best parameters found: ", opt.best_params_)

# Predict on the test set
y_pred_opt = opt.predict(X_test)

# Calculate evaluation metrics
rmse_opt = root_mean_squared_error(y_test, y_pred_opt)
r2_opt = r2_score(y_test, y_pred_opt)
# Output results
print(f"Test RMSE (GOSS with tuning): {rmse_opt:.4f}")
print(f"Test R² (GOSS with tuning): {r2_opt:.4f}")
```

```
Best parameters found:  OrderedDict({'other_rate': 0.33986603248215197, 'top_rate': 0.319014
Test RMSE (GOSS with tuning): 3458.7664
Test R² (GOSS with tuning): 0.9592
```

11.2.3.5 Optimizing LightGBM with `BayesSearchCV`

`BayesSearchCV` from `scikit-optimize` provides an efficient way to tune hyperparameters. Here's how to set this up:

```
%%time
# ===== 2. Hyperparameter Tuning with Bayesian Optimization =====
# Define the parameter space for Bayesian optimization
```

```
param_space = {
    'num_leaves': Integer(20, 100),
    'max_depth': Integer(5, 50),
    'min_data_in_leaf': Integer(1, 100),
    'learning_rate': Real(0.01, 0.5, prior='uniform'),
    'n_estimators': Integer(50, 500),
    'top_rate': Real(0.1, 0.6, prior='uniform'),
    'other_rate': Real(0.1, 0.4, prior='uniform'),
}
# Create the Bayesian search object
bayes_search = BayesSearchCV(
    # using verbose=-1 to suppress warnings
    # using n_jobs=-1 to use all available cores
    # using random_state=42 for reproducibility
    estimator=lgb.LGBMRegressor( categorical_feature=categorical_feature, random_state=42, bo
    # Define the parameter space for Bayesian optimization
    search_spaces=param_space,
    n_iter=50,
    scoring='neg_root_mean_squared_error',
    cv=3,
    n_jobs=-1,
    random_state=42
)
# Fit the Bayesian search object to the training data
bayes_search.fit(X_train, y_train)
# Get the best parameters and score
best_params = bayes_search.best_params_
best_score = bayes_search.best_score_
print(f"Best Parameters: {best_params}")
print(f"Best Score: {best_score}")
# Get the best model
best_model = bayes_search.best_estimator_
# Make predictions on the test set
y_pred_bayes = best_model.predict(X_test)
# Calculate RMSE and R2 score for the best model
rmse_bayes = root_mean_squared_error(y_test, y_pred_bayes)
r2_bayes = r2_score(y_test, y_pred_bayes)
print(f"RMSE (Bayesian Optimized): {rmse_bayes}")
print(f"R2 Score (Bayesian Optimized): {r2_bayes}")
```

```
Best Parameters: OrderedDict({'learning_rate': 0.31777940485083805, 'max_depth': 5, 'min_data
Best Score: -3361.8218393725633
```

```
RMSE (Bayesian Optimized): 3071.418344800289
R2 Score (Bayesian Optimized): 0.9678447743461689
CPU times: total: 49.4 s
Wall time: 1min 35s
```

LightGBM achieved performance comparable to **XGBoost**. By leveraging **GOSS** (Gradient-based One-Side Sampling) for gradient sampling and **EFB** (Exclusive Feature Bundling) for feature reduction, it improved training speed slightly on large, sparse datasets. These optimizations, along with **native categorical feature support**, can also help reduce cross-validation tuning time by simplifying the feature space and accelerating learning.

11.3 CatBoost

11.3.1 What is CatBoost?

CatBoost (short for *Categorical Boosting*) is a high-performance gradient boosting framework developed by **Yandex**. While several modern boosting frameworks support native categorical features, CatBoost uses **optimized encoding strategies**—such as **ordered target statistics**—that often lead to better performance with less risk of overfitting on **categorical-heavy** data.

In addition to its categorical handling, CatBoost includes features like **ordered boosting** and strong **regularization**, which help reduce overfitting. It typically requires **less hyperparameter tuning** than XGBoost or LightGBM, making it more **user-friendly**, especially on datasets with many categorical variables.

11.3.2 What Makes CatBoost Unique?

CatBoost introduces several **key innovations** that set it apart from other gradient boosting frameworks:

11.3.2.1 Symmetric (Oblivious) Trees

CatBoost builds **symmetric (oblivious) decision trees**, where the same feature and split threshold are used at each level of the tree across all nodes. This structure results in:

- **Robust to noise**
- **Improved regularization**
- **Faster inference times**

11.3.2.2 Advanced Categorical Feature Handling

CatBoost can **natively process categorical features** using an approach based on **ordered target statistics**, which:

- Avoids target leakage during training
- Typically outperforms traditional encodings like one-hot or label encoding

11.3.2.3 Ordered Boosting (vs. Standard Boosting)

Traditional gradient boosting algorithms often suffer from **prediction shift**, a form of overfitting that occurs when the model uses the same data to compute residuals and to fit new trees.

CatBoost addresses this with **ordered boosting**, a permutation-driven strategy that builds each tree on one subset of data and computes residuals on another (unseen) subset.

Recall that gradient boosting fits trees on the gradient of the loss function:

$$r_m = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$$

In classic boosting, this gradient is calculated using the same training observations that were used to fit the model, which leads to target leakage.

In contrast, CatBoost:

- Shuffles the data at each iteration
- Computes residuals for an observation **only from prior observations** in the permutation
- Ensures that **each gradient estimate is based on unseen data**

This significantly improves the model's **generalizability** and reduces overfitting, especially on **small or noisy datasets**.

11.3.2.4 Handling of Text and Embedding Features

CatBoost can process text features directly by converting them into numerical representations (e.g., using bag-of-words or embeddings) within the model, reducing the need for external preprocessing. It also supports integration with pre-trained embeddings, which is useful for natural language processing (NLP) tasks.

Together, these innovations make CatBoost a strong candidate for modeling **high-dimensional, categorical, and imbalanced tabular data**, even with minimal feature engineering or hyperparameter tuning.

11.3.2.5 Ease of Use and Defaults

CatBoost's default hyperparameters are well-tuned for a wide range of problems, reducing the need for extensive tuning. For example, its learning rate, depth, and regularization parameters often yield strong performance out of the box. In the paper, the authors also showed that CatBoost outperforms 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.

11.3.3 Using CatBoost

CatBoost provides a **scikit-learn-compatible API** through `CatBoostClassifier` and `CatBoostRegressor`, which makes it easy to integrate into pipelines and use with tools like `GridSearchCV`, `cross_val_score`, and `train_test_split`.

11.3.4 Installation

To install CatBoost, run:

```
pip install catboost
```

> GPU users: CatBoost automatically detects and uses GPU if available. You can explicitly enable it with `task_type='GPU'`.

11.3.5 CatBoost for Regression

Let us check the performance of `CatBoostRegressor()` without tuning, i.e., with default hyperparameter settings on our car dataset

The parameter `cat_features` will be used to specify the indices of the categorical predictors for target encoding.

```python
# build a catboostregressor model
from catboost import CatBoostRegressor
# Initialize the CatBoost regressor
model_cat = CatBoostRegressor(
    cat_features=categorical_feature,
    random_seed=42,
    verbose=0
)

# Train the model
model_cat.fit(X_train, y_train)
# Predict on the test set
y_pred_cat = model_cat.predict(X_test)
# Calculate evaluation metrics
rmse_cat = root_mean_squared_error(y_test, y_pred_cat)
r2_cat = r2_score(y_test, y_pred_cat)
# Output results
print(f"Test RMSE (CatBoost): {rmse_cat:.4f}")
print(f"Test R² (CatBoost): {r2_cat:.4f}")
```

```
Test RMSE (CatBoost): 3307.2604
Test R² (CatBoost): 0.9627
```

Even with default hyperparameter settings, CatBoost has outperformed both XGBoost and LightGBM in terms of test RMSE and R-squared.

11.3.6 Tuning `CatBoostRegressor`

You can tune the hyperparameters of `CatBoostRegressor` using **Optuna** or other tuning strategies, just as you would for `XGBoost` or `LightGBM`. However, CatBoost has a **distinct set of hyperparameters**, reflecting its unique design choices.

11.3.6.1 Hyperparameters not used in CatBoost:

- `reg_alpha`: CatBoost does **not** support L1 regularization on leaf weights; it uses only **L2 regularization** (`l2_leaf_reg`).
- `colsample_bytree`: CatBoost **does not** use this parameter; it uses `rsm` and handles feature selection differently.

These parameters are common in XGBoost and LightGBM but are **not part of CatBoost's configuration**.

11.3.6.2 Unique Hyperparameters in CatBoost

CatBoost introduces several hyperparameters related to **categorical feature handling** and **ordered boosting**:

- `one_hot_max_size`: Threshold for switching between **one-hot encoding** and **target encoding** for categorical features.
- `boosting_type='Ordered'`: Ordered boosting is **enabled by default** in CatBoost to reduce overfitting and prevent prediction shift.
- `bootstrap_type='Bayesian'`: Default bootstrap method. Works well with ordered boosting.
- `bagging_temperature`: Works with `bootstrap_type='Bayesian'`.
  Controls how sharply bootstrap weights are distributed:
  - **Low values** (e.g., `0`): more uniform sampling (close to deterministic).
  - **High values** (e.g., `1`, `5`, `10`): more aggressive sampling—some rows are weighted more heavily.
- `random_strength`: Adds randomness to the **split selection score**, especially useful for regularizing categorical splits.
- `rsm`: Random selection rate for column sampling.

These CatBoost-specific hyperparameters are important when fine-tuning with Cross-Validation, particularly for datasets with many **categorical features** or at risk of **overfitting**.

```
import optuna
from optuna import create_study
from catboost import CatBoostRegressor, Pool

# create a validation set for early stopping
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.1, rando

#convert to Catboost pool
train_pool = Pool(X_train, y_train, cat_features=categorical_feature)
valid_pool = Pool(X_valid, y_valid, cat_features=categorical_feature)

# Define the objective function for Optuna
def objective(trial):
    # Define the hyperparameters to tune
    params = {
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3),
        'depth': trial.suggest_int('depth', 4, 10),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1e-8, 10.0, log=True),
        'min_data_in_leaf': trial.suggest_int('min_data_in_leaf', 1, 30),
        'bagging_temperature': trial.suggest_float('bagging_temperature', 0.0, 1.0),
        'random_strength': trial.suggest_float('random_strength', 1e-8, 10.0, log=True),

        # Fixed parameters
        'iterations': 3000,  # Set to a high number, early stopping will determine the actua
        'verbose': False,
        'random_seed': 42
    }

    # Create and train the model with early stopping
    model = CatBoostRegressor(**params)

    # Use early stopping to prevent overfitting
    model.fit(
        train_pool,
        eval_set=valid_pool,
        early_stopping_rounds=20,  # Stop if no improvement for 50 rounds
        verbose=False,
        n_jobs=-1
    )

    # Evaluate on validation set
    y_pred = model.predict(valid_pool)
    val_rmse = root_mean_squared_error(y_valid, y_pred)

    # Return negative RMSE (for maximization)
    return -val_rmse

# Create and run the study
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)
```

```
[I 2025-05-16 06:53:24,620] A new study created in memory with name: no-name-2b89c789-86c1-4
[I 2025-05-16 06:53:36,438] Trial 0 finished with value: -2885.7311361568736 and parameters:
[I 2025-05-16 06:54:07,518] Trial 1 finished with value: -2620.171198484414 and parameters:
[I 2025-05-16 06:55:01,902] Trial 2 finished with value: -2696.8294539080816 and parameters:
[I 2025-05-16 06:55:22,721] Trial 3 finished with value: -2724.528794214105 and parameters:
[I 2025-05-16 06:55:36,587] Trial 4 finished with value: -2878.820243617394 and parameters:
[I 2025-05-16 06:55:49,775] Trial 5 finished with value: -2668.619984949151 and parameters:
```

```
[I 2025-05-16 06:55:57,119] Trial 6 finished with value: -2912.272947098299 and parameters: 
[I 2025-05-16 06:56:13,622] Trial 7 finished with value: -2695.2648598625924 and parameters:
[I 2025-05-16 06:56:26,903] Trial 8 finished with value: -2753.084945654914 and parameters: 
[I 2025-05-16 06:56:34,769] Trial 9 finished with value: -2921.4504143784643 and parameters:
[I 2025-05-16 06:56:45,103] Trial 10 finished with value: -2797.656433395867 and parameters:
[I 2025-05-16 06:57:00,983] Trial 11 finished with value: -2987.804912052353 and parameters:
[I 2025-05-16 06:58:30,211] Trial 12 finished with value: -2656.269812196844 and parameters:
[I 2025-05-16 06:59:27,791] Trial 13 finished with value: -2744.3027101379294 and parameters
[I 2025-05-16 06:59:53,265] Trial 14 finished with value: -2697.6837955182573 and parameters
[I 2025-05-16 07:00:21,452] Trial 15 finished with value: -2766.897166940931 and parameters:
[I 2025-05-16 07:01:00,089] Trial 16 finished with value: -2731.9161533728666 and parameters
[I 2025-05-16 07:01:13,872] Trial 17 finished with value: -2950.387762628911 and parameters:
[I 2025-05-16 07:01:24,805] Trial 18 finished with value: -2799.2851339120025 and parameters
[I 2025-05-16 07:01:36,339] Trial 19 finished with value: -2772.5541221320145 and parameters
```

```python
# Get best parameters and train final model with early stopping
best_params = study.best_params
print("Best parameters:", best_params)

# Get the best trial
best_trial = study.best_trial
print("Best trial:", best_trial)
```

```
Best parameters: {'learning_rate': 0.05369228812155998, 'depth': 7, 'l2_leaf_reg': 2.5645654
Best trial: FrozenTrial(number=1, state=1, values=[-2620.171198484414], datetime_start=datet
```

```python
# Use column indices instead of names
cat_feature_indices = [X_train.columns.get_loc(col) for col in categorical_feature]

# Add iterations parameter back for final model
best_params['iterations'] = 3000  # High number, early stopping will be used

# create a train+validation set for final model
train_val_pool = Pool(
    np.vstack((X_train, X_valid)),
    np.concatenate((y_train, y_valid)),
    cat_features=cat_feature_indices
)

# Create a test pool
test_pool = Pool(X_test, y_test, cat_features=categorical_feature)
```

```python
# Train final model on combined train+validation data
final_model = CatBoostRegressor(**best_params)
final_model.fit(
    train_val_pool,
    eval_set=test_pool,
    early_stopping_rounds=50,
    verbose=False
)

# Get actual number of trees used after early stopping
actual_iterations = final_model.tree_count_
print(f"Actual number of trees used: {actual_iterations}")

# Evaluate on test set
y_pred_test = final_model.predict(X_test)
test_rmse = root_mean_squared_error(y_test, y_pred_test)
test_r2 = r2_score(y_test, y_pred_test)
print(f"Test RMSE: {test_rmse:.4f}")
print(f"Test R²: {test_r2:.4f}")
```

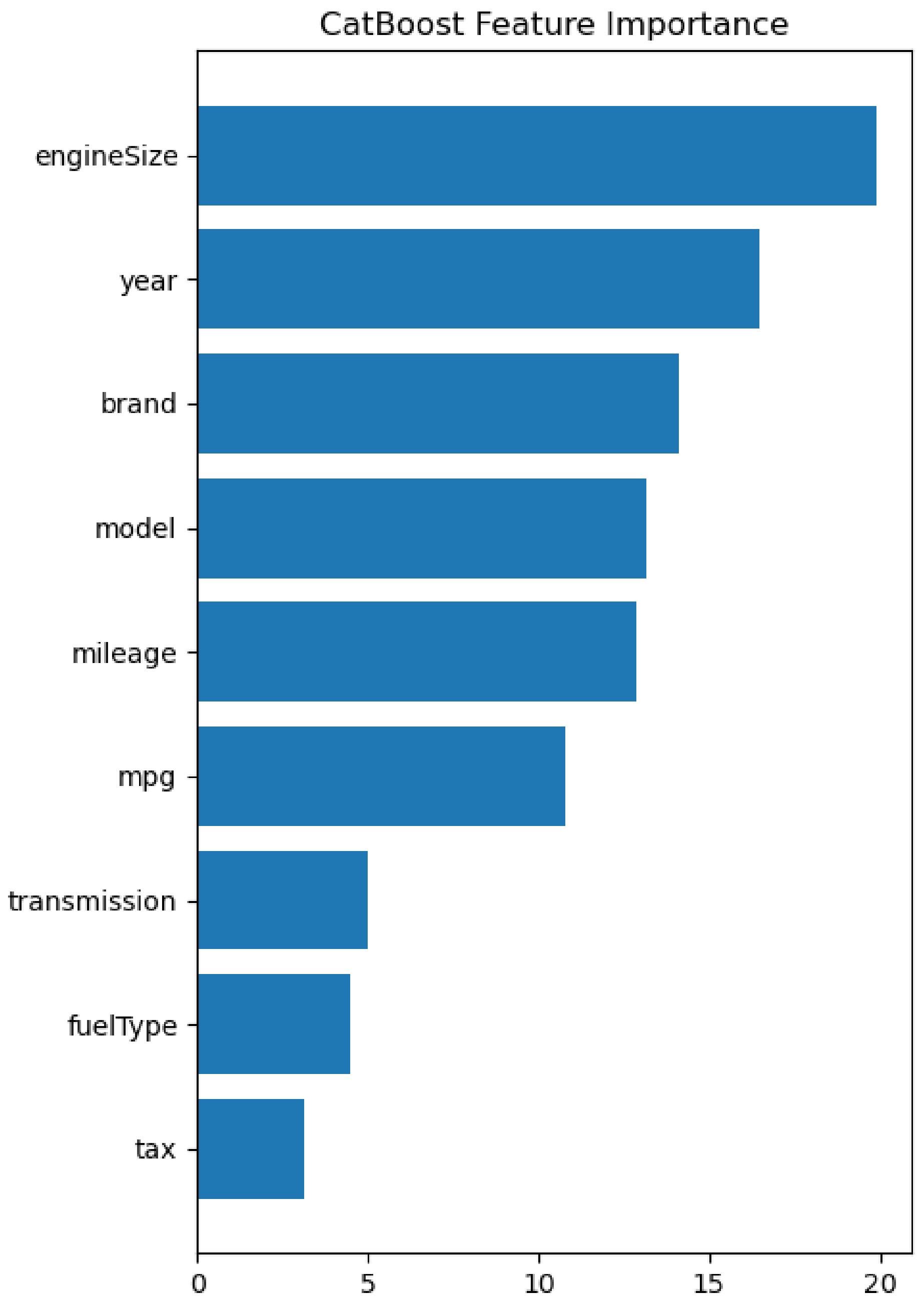
```
Actual number of trees used: 1021
Test RMSE: 3147.8477
Test R²: 0.9662
```

```python
fig1 = optuna.visualization.plot_optimization_history(study)
fig1.show()

fig2 = optuna.visualization.plot_param_importances(study)
fig2.show()

# Plot feature importance from the final model
# Get feature importance values
feature_importance = final_model.get_feature_importance()

# Get sorted indices
sorted_idx = np.argsort(feature_importance)

# Plot
plt.figure(figsize=(5, 7))
plt.barh(range(len(sorted_idx)), feature_importance[sorted_idx])
```

```python
# Use feature names if X is a DataFrame
feature_names = X.columns if hasattr(X, 'columns') else np.array(range(X.shape[1]))
plt.yticks(range(len(sorted_idx)), feature_names[sorted_idx])

plt.title('CatBoost Feature Importance')
plt.tight_layout()
plt.show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```



Check the documentation for hyperparameter tuning.

With proper hyperparameter tuning, CatBoost can achieve better performance than its default settings. However, even without tuning, CatBoost's default configuration often outperforms the default settings of XGBoost and LightGBM, particularly on datasets with categorical features.

11.3.7 When to Use CatBoost Over XGBoost

- When your dataset is “categorical-heavy**
- **CatBoost** tends to perform well **out of the box** with minimal hyperparameter tuning, making it more user-friendly for quick experimentation or deployment

- CatBoost’s **GPU implementation** is optimized for handling categorical data efficiently, and can **outperform XGBoost** on datasets dominated by categorical variables
  > While both libraries support GPU acceleration, CatBoost’s architecture is particularly well-suited for categorical-heavy tasks

11.4 Handling Imbalanced Classification: XGBoost vs. LightGBM vs. CatBoost

Imbalanced classification occurs when one class significantly outnumbers the other (e.g., fraud detection, disease diagnosis). Each boosting library offers tools to address this issue:

XGBoost:

- **Parameter**: `scale_pos_weight`
  - Formula:

$$\texttt{scale\_pos\_weight} = \frac{\text{Number of negative samples}}{\text{Number of positive samples}}$$

  - Increases the gradient of the positive class during training.
- **Additional Strategies**:
  - Use custom `eval_metric` (e.g., `"auc"`, `"aucpr"`, or `"logloss"`)
  - Apply early stopping on validation AUC

LightGBM:

- **Parameter**: `scale_pos_weight` (same as in XGBoost)
- **Alternative**: `is_unbalance = TRUE`
  - Automatically adjusts class weights based on distribution
- **Other Tips**:
  - Use `metric = "auc"` or `"binary_logloss"` for better guidance during training
  - Resampling techniques also compatible

CatBoost:

- **Parameter**: `class_weights`
  - Accepts a numeric vector (e.g., `class_weights = c(1, 5)` for [negative, positive])
  - Directly modifies the loss function to emphasize minority class
- **Advantages**:
  - More flexible than `scale_pos_weight`
  - Works well with default settings
- **Other Tips**:
  - Use `loss_function = "Logloss"` and `eval_metric = "AUC"` for binary classification

Below is the summary table:

Library	Imbalance Handling Parameter	Default Support	Recommended Metric
XGBoost	`scale_pos_weight`	No	`auc`, `aucpr`
LightGBM	`scale_pos_weight`, `is_unbalance`	Yes (with flag)	`auc`, `binary_logloss`
CatBoost	`class_weights`	Yes	`Logloss`, `AUC`

11.5 Summary: XGBoost vs. LightGBM vs. CatBoost

Gradient boosting is a powerful ensemble technique, and XGBoost, LightGBM, and CatBoost are three of its most widely used implementations. Each has unique strengths and is well-suited to different use cases.

XGBoost:

- **Strengths**: Robust, well-documented, strong performance on structured/tabular data
- **Split Finding**: Level-wise tree growth
- **Regularization**: Explicit L1 and L2 regularization
- **Flexibility**: Highly customizable with many hyperparameters
- **Best for**: General-purpose tabular data, especially when you have time to tune parameters

LightGBM:

- **Strengths**: Fast training, low memory usage, excellent scalability
- **Split Finding**: Leaf-wise tree growth with depth control
- **Binning**: Uses histogram-based algorithm with `max_bin` to speed up training
- **Best for**: Large-scale datasets, high-dimensional features, and when training speed matters

CatBoost:

- **Strengths**: Handles categorical features natively, works well with minimal tuning
- **Boosting Innovation**: Uses *ordered boosting* to prevent prediction shift
- **Categorical Encoding**: Uses target-based encoding internally
- **Best for**: Datasets with many categorical variables or limited time for tuning

Final Thoughts

All three libraries are powerful and battle-tested. Here's a rough guideline:

- **Use XGBoost** if you want control, flexibility, and a well-documented standard
- **Use LightGBM** when training speed and large data scalability are your top priorities
- **Use CatBoost** when working with many categorical features or seeking strong baseline results with minimal tuning

11.6 References

- LightGBM Paper (Original NIPS 2017)
- LightGBM Official Website
- CatBoost Paper (arXiv)
- CatBoost Official Website

12 Smarter Hyperparameter Tuning

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

Tree-based models, like decision trees, random forests, and boosting algorithms (e.g., XG-Boost, LightGBM, CatBoost), benefit significantly from hyperparameter optimization. Tools like `GridSearchCV`, `RandomizedSearchCV`, and `cross_val_score` in scikit-learn enable robust tuning via cross-validation, similar to linear or logistic regression models. Below, we explore these tools and smarter alternatives for complex models.

12.1 Cross-Validation Basics

- `cross_val_score`: Evaluates a model's performance for a fixed set of hyperparameters using cross-validation. Requires manual loops to search over hyperparameter combinations, making it labor-intensive for extensive tuning.
- `GridSearchCV` and `RandomizedSearchCV`: Automate hyperparameter search with built-in cross-validation, internally handling loops to evaluate combinations efficiently.

12.2 `GridSearchCV`: Exhaustive but Limited

`GridSearchCV` performs an **exhaustive search** over a predefined grid of hyperparameter values.

- **Pros**:
  - Guarantees finding the best combination within the specified grid.
  - Ideal for small, well-defined search spaces (e.g., tuning `max_depth=[3, 5, 7]` and `n_estimators=[100, 200]`).
- **Cons**:
  - Requires **discrete lists** for each hyperparameter, unable to sample from continuous distributions.

    – Grid size grows **exponentially** with more hyperparameters, making it **computationally infeasible** for complex models like boosting trees (e.g., XGBoost with 5+ hyperparameters).
    – Time-consuming, especially for large datasets or slow models.

Use Case: Best for simple models or when you have a small, targeted set of hyperparameter values.

12.3 `RandomizedSearchCV`: Efficient but Random

`RandomizedSearchCV` samples a fixed number of hyperparameter combinations (`n_iter`) randomly from user-defined distributions.

- **Pros**:
    - **More efficient** than `GridSearchCV` for large search spaces, as it evaluates fewer combinations.
    - Supports **continuous distributions** (e.g., `scipy.stats.loguniform` for learning rate), offering greater flexibility.

      ```
      from scipy.stats import uniform
      'learning_rate': uniform(loc=0.01, scale=0.3),  # Range: [0.01, 0.31)
      ```

    - **Faster**, as it avoids exhaustive enumeration.
- **Cons**:
    - Random sampling may **miss optimal regions**, especially in high-dimensional spaces or with limited iterations.
    - Performance depends on the choice of `n_iter` and distribution quality.

Use Case: Preferred for **initial exploration** or when **computational resources are limited** but the search space is large.

12.4 Smarter Tuning for Complex Models

For boosting models, neural networks, and other high-dimensional algorithms, hyperparameter tuning becomes especially challenging. Boosting models—such as **Gradient Boosting**, **XGBoost**, and **LightGBM**—are powerful but come with a large number of hyperparameters, including learning rate, max depth, number of estimators, and regularization terms. Many of these parameters exist in a **continuous space**, rather than a simple set of discrete options.

This complexity makes **GridSearchCV** computationally expensive and often impractical. Even **RandomizedSearchCV**, while more efficient, can be suboptimal in high-dimensional or continuous search spaces due to its purely random sampling strategy.

To overcome these limitations, more advanced hyperparameter optimization techniques have been developed. These methods efficiently explore the hyperparameter space and often achieve better performance with fewer evaluations.

Below, we introduce two such approaches.

12.4.1 `BayesSearchCV` (Bayesian Optimization)

- **How it works**: Uses a probabilistic surrogate model (e.g., Gaussian Process) to predict promising hyperparameter combinations based on past evaluations.
- **Advantages**:
  - Converges faster than random search by focusing on high-performing regions.
  - It works natively with scikit-learn pipelines, simplifying tuning for workflows with preprocessing
  - GP (Gassian Process) - based optimization is effective for smooth, continuous hyperparameters (e.g., learning rate, regularization), converging faster in low-dimensional spaces (Tuning 2-5 continuous parameters).
  - Ideal for produnction environment restricted to scikit-learn extensions
- **Library**: `scikit-optimize` or `bayes_opt`.
- **Use Case**: Ideal for **small to medium** search spaces with **continuous parameters** on **small to medium-sized** datasets.

We will now return to the car dataset and perform hyperparameter optimization on the XGBoost model using BayesSearchCV.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import root_mean_squared_error, r2_score, accuracy_score, precision_sco
from xgboost import XGBRegressor, XGBClassifier
import seaborn as sns
import plotly
```

```
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
```

12.4.1.1 Data Preprocessing

```
# 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']

# 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()

# convert the categorical columns to category type
for col in categorical_feature:
    X[col] = X[col].astype('category')


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

12.4.1.2 Baseline Performance

```
# build a baseline xgboost model
xgb = XGBRegressor(objective='reg:squarederror', enable_categorical=True, random_state=42)
# fit the model
xgb.fit(X_train, y_train, )
# make predictions
y_pred = xgb.predict(X_test)

print ('Baseline Model: ')
# calculate the RMSE
rmse = root_mean_squared_error(y_test, y_pred)
print(f'RMSE: {rmse}')
# calculate the R2 score
r2 = r2_score(y_test, y_pred)
print(f'R2: {r2}')
```

```
Baseline Model:
RMSE: 3299.648193359375
R2: 0.9628884792327881
```

12.4.1.3 `BayesSearchCV` inital tuning

Let's use `BayesSearchCV` to boost its performance

```
%%time
# define the search space for Bayesian optimization
search_space = {
    'n_estimators': Integer(50, 500),
    'max_depth': Integer(5, 30),
    'learning_rate': Real(0.01, 0.3, prior='uniform'),
    'subsample': Real(0.5, 1.0, prior='uniform'),
    'colsample_bytree': Real(0.5, 1.0, prior='uniform'),
    'gamma': Real(0, 5, prior='uniform'),
}

# define the model
xgb = XGBRegressor(objective='reg:squarederror', enable_categorical=True, random_state=42)
# define the Bayesian optimization search
bayes_search = BayesSearchCV(
    xgb,
    search_space,
    n_iter=50,
    scoring='neg_root_mean_squared_error',
    cv=3,
    n_jobs=-1,
    random_state=42,
    verbose=0
)
# fit the model
bayes_search.fit(X_train, y_train)
```

```
CPU times: total: 46.6 s
Wall time: 3min 24s
```

```
BayesSearchCV(cv=3,
              estimator=XGBRegressor(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=True, eval_metric=None,
                                     feature_types=None, feature_weights=None,
                                     gamma=None, grow_policy=None,
                                     importance_type=None,
                                     interaction_constraints=None...
                              'gamma': Real(low=0, high=5, prior='uniform', transform='normali
                              'learning_rate': Real(low=0.01, high=0.3, prior='uniform', trans
                              'max_depth': Integer(low=5, high=30, prior='uniform', transform=
                              'n_estimators': Integer(low=50, high=500, prior='uniform', trans
                              'subsample': Real(low=0.5, high=1.0, prior='uniform', transform=
```

```
print('Bayesian Optimization Results: ')
# get the best parameters
best_params_bayes = bayes_search.best_params_
print('Best Parameters: ')
print(best_params_bayes)
# get the best score
best_score_bayes = bayes_search.best_score_
print('Best CV Score: ', best_score_bayes)
# make predictions
```

```
y_pred_bayes = bayes_search.predict(X_test)
# calculate the RMSE
rmse_bayes = root_mean_squared_error(y_test, y_pred_bayes)
print('Test RMSE: ', rmse_bayes)
# calculate the R2 score
r2_bayes = r2_score(y_test, y_pred_bayes)
print('Test R2: ', r2_bayes)
```

```
Bayesian Optimization Results:
Best Parameters:
OrderedDict({'colsample_bytree': 0.7800962787418171, 'gamma': 5.0, 'learning_rate': 0.032536
Best CV Score:  -3061.9464518229165
Test RMSE:  3236.476318359375
Test R2:  0.9642958641052246
```

12.4.1.4 Visualizing `BayesSearchCV` Results

Visualizing `BayesSearchCV` outcomes helps interpret the hyperparameter optimization process for tree-based models like **XGBoost**. Visualization can provide critical insights into convergence, parameter impact, and model behavior, helping you understand and refine model behavior.

Key Benefits of Visualization:

- **Check Convergence**:
  Plot best scores vs. iterations to determine if the search has stabilized or requires more trials.
- **Identify Key Parameters**:
  Use scatter plots to reveal which hyperparameters (e.g., `learning_rate`) significantly influence performance.
- **Explore Interactions**:
  Use heatmaps or contour plots to examine relationships between hyperparameters (e.g., `learning_rate` vs. `max_depth`).
- **Diagnose Issues**:
  Spot poor search regions or insufficient iterations through patterns in the plots.

Practical Tips

- **Plots**:
  Use convergence, scatter, heatmap, or parallel coordinate plots with libraries like **Matplotlib** or **Seaborn**.