Midterm Examination

Name: Zankhana Mehta

NUID: 002320268

Mail: mehta.zan@northeastern.edu

PROBLEM 1

1. Generalization is the ability of a tranined machine learning model to perform well on new or unseen data. It is the models capacity to capture underlying patterns and trends in the data.

Relationship between training and generalization: Training involves teaching a model using a dataset, where it adjusts its parameters to reduce errors on that data. Generalization is the model's ability to apply what it has learned to new, unseen data. If a model performs well on training data but poorly on new data, it has likely overfitted—meaning it memorized the training data instead of learning the underlying patterns. The goal is to strike a balance: the model should fit the training data well and also perform accurately on new data. The challenge is to avoid overfitting (too complex) or underfitting (too simple).

Relationship between regularization and generalization: Regularization helps improve generalization by adding a penalty for complexity, preventing the model from overfitting the training data. It works by balancing the bias-variance tradeoff. It reduces the model's variance by limiting its flexibility, which can enhance generalization.

2. The loss function used for logistic regression is the log loss or binary cross-entropy loss, which measures the difference between the true labels (y) and the predicted probabilities (p). For a single training example, the loss is:

$$\text{Loss} = -[y \log(p) + (1 - y) \log(1 - p)]$$

For a dataset with (N) samples, the average log loss) is:

$$J(\theta) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

Relationship Between Logistic Regression and Maximum Likelihood Estimation (MLE)

Logistic regression is closely related to Maximum Likelihood Estimation (MLE), a statistical method for estimating the parameters (weights) of a model. MLE seeks to find the parameter values that maximize the likelihood of observing the given data.

Here's how logistic regression connects to MLE:

Likelihood Function: For binary classification, the likelihood of the data is the product of the probabilities of each sample, given the predicted probabilities (p_i). This is:

$$L(\theta) = \prod_{i=1}^{N} p_i^{y_i} (1 - p_i)^{1-y_i}$$

Log-Likelihood: To simplify computations, we often work with the log of the likelihood function. The log-likelihood for logistic regression is:

$$\ell(\theta) = \sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

Maximizing Log-Likelihood: The goal of MLE is to find the parameters beta that maximize the log-likelihood. Since maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood, the cost function of logistic regression (binary cross-entropy loss) is simply the negative log-likelihood.

Thus, minimizing the logistic regression loss function is equivalent to maximizing the likelihood of the parameters given the data, which is the essence of maximum likelihood estimation.

3. a) Subset selection methods like best subset selection or forward selection do not necessarily require feature standardization. These methods focus on selecting the most important features based on their contribution to the model. However, when features have significantly different scales or units, the methods might favor features with larger values, as these features may have larger coefficients and appear more influential in the model. In such cases, standardizing the features helps ensure that each feature is treated equally, preventing those with larger scales from dominating the selection process. Therefore, while standardization isn't a must, it can be beneficial when features have varying scales to make the comparison of their contributions fairer.

b) For regularized methods like LASSO (L1 regularization) and Ridge (L2 regularization),

standardizing features is crucial.

LASSO penalizes the absolute values of coefficients. If features are on different scales, larger-valued features may be penalized less, potentially staying in the model, while smaller-valued features may be excluded. Standardizing ensures equal penalization across features, improving feature selection.

Ridge penalizes the sum of squared coefficients. If features have different scales, the regularization will disproportionately affect larger features, making it harder for smaller ones to compete. Standardization ensures the penalty is applied consistently across all features.

Standardization ensures fair and effective regularization and feature selection.

4. Rudge regression cannot typically perform variable selection. It applies L2 regularization, which shrinks the coefficients of less important variables but does not force any coefficients to be exactly zero. As a result, even features that do not contribute much to the model may still retain small but non-zero coefficients. It reduces the impact of less relevant features by shrinking their coefficients but keeps them in the model. We can manually conduct an analysis to see the variables with less coefficient values and then remove them further, this could be one way to perform variable selection using ridge regression.

Whereas, Lasso applies L1 regularization, which encourages sparsity in the coefficients. By adding a penalty on the absolute values of the coefficients, LASSO forces some coefficients to become exactly zero, effectively eliminating some variables from the model. This results in a simpler, more interpretable model that includes only the most important features. Therefore, LASSO performs variable selection by driving less relevant feature coefficients to zero, thus removing them from the model.

The regularization parameter λ controls the strength of the penalty applied to the model's coefficients.

For Ridge Regression:

Bias: As λ increases, the coeficients are more heavily penalized, causing them to shrink towards zero. This increases the bias of the model because the coefficients are forced to be smaller than they would be in an unregularized model.

Variance: As λ increases, the variance decreases because the model becomes simpler (fewer large coefficients), and it overfits the training data less.

Number of selected variables: Ridge regression does not eliminate any variables, so the number of selected variables remains the same regardless of the value of λ. All variables remain in the model, but with smaller coefficients as λ increases.

For LASSO:

Bias: As λ increases, LASSO imposes a stronger penalty, forcing more coefficients to shrink to zero. This increases the bias because the model becomes overly simplistic, potentially excluding important variables.

Variance: As λ increases, variance decreases because the model becomes simpler and less likely to overfit the training data.

Number of selected variables: The number of selected variables decreases as λ increases. With a very large λ, LASSO can shrink many coefficients to zero, effectively eliminating them from the model. This makes LASSO a powerful tool for variable selection.

5. (a) Logistic Regression:Logistic Regression has a linear decision boundary. Logistic regression models the probability of class membership using a linear function of the input features. Therefore, the decision boundary between classes is a straight line (or hyperplane in higher dimensions).

   (b) Linear Discriminant Analysis (LDA):This too has a linear decision boundary. LDA assumes that the features follow a Gaussian distribution with the same covariance matrix for each class. The decision boundary is linear because LDA finds a linear combination of features that best separates the classes.

   (c) Quadratic Discriminant Analysis :QDA has a Quadratic (curved) decision boundary. It assumes that each class has its own covariance matrix, which allows the decision boundaries to be quadratic. This means the boundaries can bend and curve to better fit the data, making QDA more flexible than LDA in capturing nonlinear separations between classes.

   (d) k-Nearest Neighbors (k-NN): KNN has a Non-linear and Piecewise decision boundary. The decision boundary of k-NN is highly flexible and can take any shape, depending on the data and the value of k. The boundary is created by the local proximity of data points and changes as the number of neighbors considered (k) changes. With small k, the boundary can be highly irregular, while with larger k, it tends to smooth out.

   (e) Random Forests: They too have a Non-linear and Piecewise. Random forests are

an ensemble of decision trees, where each tree provides a classification. Since decision trees split the data based on feature values at specific thresholds, the decision boundaries of random forests are highly non-linear and complex.

6. LASSO (Least Absolute Shrinkage and Selection Operator) is often referred to as shrinkage because it applies a penalty to the model coefficients, which shrinks the coefficients of less important features towards zero. This shrinkage results in some coefficients becoming exactly zero, effectively removing the corresponding features from the model. This is why LASSO performs both regularization and variable selection.

Let's say we have a model with two variables: x1: Strong correlation with the target variable. and x2: Weak correlation with the target variable.

Scenario:

Without LASSO the The least squares regression would find non-zero coefficients for both variables, even if the coefficient for x2 is very small.

Wheras, with LASSO as λ increases, the LASSO penalty becomes more important. The coefficient for x2 (the weakly correlated variable) will shrink faster towards zero, as the penalty for its small value is greater than for the strongly correlated variable.

7. In LOO-CV, for a dataset with n samples, the model is trained n times. Each time, one sample is left out as the test set, and the remaining n–1 samples are used for training. After training on n–1 samples, the model is tested on the left-out sample, and this process is repeated for each sample in the dataset.

In K-fold cross-validation, the dataset is split into k subsets (or folds). The model is trained on k–1 of these folds and tested on the remaining fold. This process is repeated k times, each time with a different fold as the test set, and the results are averaged.

Key Differences:

Data Splitting: LOOCV uses each sample as a test set once, while K-fold divides the dataset into k groups and uses a different group for testing in each iteration.

Computational Efficiency: LOOCV can be computationally expensive for large datasets as it trains the model n times, whereas K-fold requires only k iterations.

Bias vs. Variance: LOOCV generally has lower bias but higher variance, while K-fold strikes a better balance between bias and variance. Cross-validation helps avoid

overfitting by providing a more reliable estimate of how the model will perform on unseen data. It does this by training and evaluating the model on different subsets of the data, allowing the model to be tested on data it hasn't seen during training. This gives you an indication of whether the model is overfitting to the training data or if it can generalize well to new, unseen data.

Here are some common strategies to prevent overfitting:

Regularization: L1 (LASSO) that shrinks coefficients toward zero, potentially setting some to zero, leading to a sparser model. L2 (Ridge) that shrinks coefficients towards zero, but doesn't set them to zero. Good for reducing variance.

Cross-Validation: Use cross-validation to select the best model parameters (e.g., regularization parameter, tree depth) based on performance on unseen data.

Early Stopping: For iterative models (like neural networks), stop training when performance on a validation set starts to decline.

Feature Selection: Remove less informative features to simplify the model.

8. Performance Metrics for Classification

a. Accuracy:The proportion of correctly classified instances out of all instances. Accuracy= True Positives + True Negatives/Total Instances

b. Pricision(Positive prediction value): The proportion of true positives (correctly predicted positive instances) out of all predicted positives. Precision =True Positives/True Positives + False Positives

c. Recall (Sensitivity, True Positive Rate): The proportion of true positives out of all actual positives. Recall =True Positives/True Positives + False Negatives

d. F1 score: The harmonic mean of precision and recall. It balances precision and recall, especially when you have imbalanced classes. F1-Score=2×[(Precision×Recall)/(Precision + Recall)]

e. AUC-ROC (Area Under the Receiver Operating Characteristic Curve):Measures the ability of the classifier to distinguish between the positive and negative classes, with values ranging from 0 to 1. ROC Curve plots the true positive rate (recall) against the false positive rate. AUC represents the likelihood of the classifier ranking a randomly chosen positive instance higher than a randomly chosen negative instance.

f. Confusion Matrix: A table showing the actual vs predicted classifications, including:

True Positives (TP) True Negatives (TN) False Positives (FP) False Negatives (FN)

In the case of imbalanced classes, accuracy may not be informative, as a model could simply predict the majority class and still appear to perform well. In such cases, the following metrics are more reliable:

Precision and Recall are still valuable, but their interpretation needs to be adjusted for the context of the imbalance.

F1-Score: A good balance between precision and recall, especially useful in imbalanced scenarios.

Specificity: The proportion of correctly predicted negative instances out of all actual negative instances. This is important when correctly identifying negative instances is crucial, often the case in imbalanced data.

AUC: Still a valuable metric as it considers the performance across all classification thresholds and is less sensitive to class imbalance than accuracy.

These metrics are better suited for imbalanced data because they focus on the performance of the model in predicting both positive and negative instances correctly, not just optimizing for overall accuracy. This provides a better understanding of how the model is performing on both the majority and minority classes.

9. Main differences are

Ensemble Type:

Random Forests: Builds many trees independently and combines their predictions.

Gradient Boosting: Builds trees sequentially, each correcting errors from the previous one.

Speed:

Random Forests: Faster, as trees are built in parallel.

Gradient Boosting: Slower, as trees are built one after another.

Overfitting:

Random Forests: Less prone to overfitting.

Gradient Boosting: More prone to overfitting but can be controlled with tuning.

Performance:

Random Forests: Easier to use, good for general tasks.

Gradient Boosting: More accurate, but needs careful tuning.

If there is enough time and computational resources to fine-tune the model, gradient boosting often performs better in terms of predictive power. If you need a quick, robust model with lower risk of overfitting and less need for tuning, random forests may be a better choice.

10. In transfer learning, "feature extraction" uses a pre-trained model to extract features from new data, essentially treating the model as a fixed feature extractor, while "full fine-tuning" further trains the entire pre-trained model on the new task data, allowing it to adjust its internal representations to better suit the new task

Key Differences:

Model Adjustment:In feature extraction, the pre-trained model's weights are frozen, meaning no further training happens on the new data; in full fine-tuning, the entire model is trained on the new data, allowing all its parameters to adjust.

Dataset Similarity:Feature extraction is often preferred when the new dataset is significantly different from the pre-trained data, as it leverages the general features learned by the pre-trained model without trying to overfit to the new task. Conversely, full fine-tuning is more suitable when the new dataset is similar to the pre-trained data, allowing the model to fine-tune its features for the specific task.

Computational Cost:Feature extraction generally requires less computational power because only the new classifier layers are trained on top of the pre-trained features. Full fine-tuning, on the other hand, requires more computational resources as the entire model is being updated

PROBLEM 2

Part a

In [32]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
import itertools
from sklearn.metrics import mean_squared_error
```

In [5]:
```
train_data = pd.read_csv('SeoulBikeData_6pm_train.csv')
```

In [7]:
```
train_data.head()
```

Out[7]:

	Temperature	Humidity	Wind_speed	Visibility	Dew_point_temperature	Solar_Radi
0	0.6	66	1.4	2000	-5.0	
1	6.0	84	1.9	327	3.4	
2	1.7	90	1.0	66	0.2	
3	2.0	61	3.0	2000	-4.7	
4	-7.7	36	1.0	2000	-20.2	

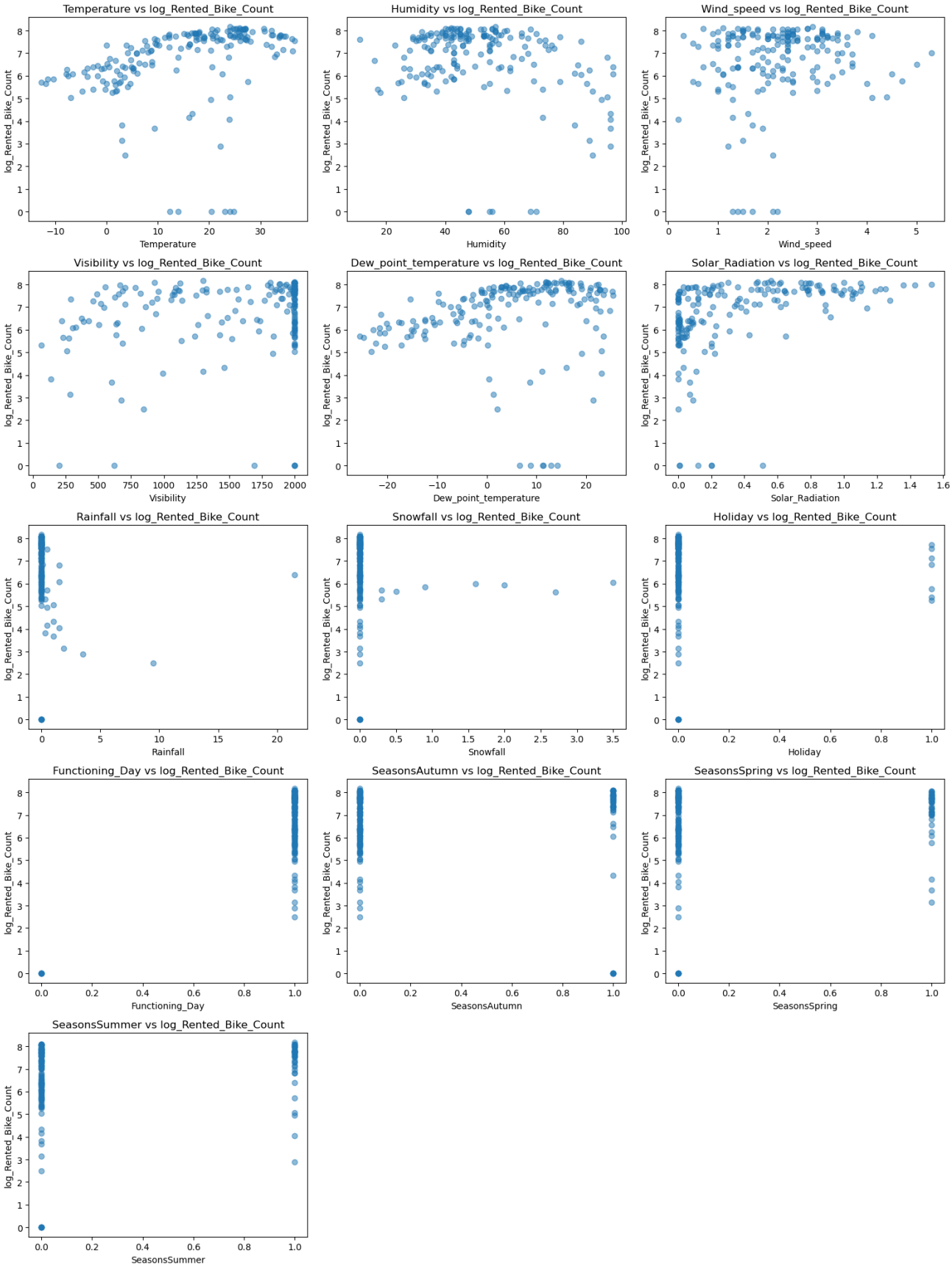
In [21]:
```
train_data.shape
```

Out[21]:
```
(182, 14)
```

In [13]:
```
train_data.columns
```

Out[13]:
```
Index(['Temperature', 'Humidity', 'Wind_speed', 'Visibility',
       'Dew_point_temperature', 'Solar_Radiation', 'Rainfall', 'Snowfall',
       'Holiday', 'Functioning_Day', 'SeasonsAutumn', 'SeasonsSpring',
       'SeasonsSummer', 'log_Rented_Bike_Count'],
      dtype='object')
```

In [15]:
```
input_features = [col for col in train_data.columns if col != 'log_Rented_Bi

# Plot scatter plots for all features against log_Rented_Bike_Count
plt.figure(figsize=(15, 20))
for i, feature in enumerate(input_features, 1):
    plt.subplot(5, 3, i)  # Adjust grid size as needed based on number of fe
    plt.scatter(train_data[feature], train_data['log_Rented_Bike_Count'], al
    plt.title(f'{feature} vs log_Rented_Bike_Count')
    plt.xlabel(feature)
    plt.ylabel('log_Rented_Bike_Count')

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



In [171…
```
correlation_matrix = train_data.corr()
```

```
correlation_matrix
```

Out[171…

	Temperature	Humidity	Wind_speed	Visibility	Dew_point_
Temperature	1.000000	0.142379	-0.077484	0.125385	
Humidity	0.142379	1.000000	-0.267810	-0.553167	
Wind_speed	-0.077484	-0.267810	1.000000	0.057451	
Visibility	0.125385	-0.553167	0.057451	1.000000	
Dew_point_temperature	0.915753	0.516120	-0.170754	-0.101025	
Solar_Radiation	0.671849	-0.249834	0.242855	0.271218	
Rainfall	0.018686	0.324536	-0.144280	-0.231394	
Snowfall	-0.233948	0.122178	-0.084123	-0.086948	
Holiday	-0.019104	-0.162794	0.033501	0.112813	
Functioning_Day	-0.074350	-0.060322	0.101217	0.029299	
SeasonsAutumn	0.093034	0.007803	-0.312750	0.116580	
SeasonsSpring	0.040868	-0.085690	0.291295	-0.137617	
SeasonsSummer	0.641430	0.244406	-0.026393	0.090206	
log_Rented_Bike_Count	0.317119	-0.276239	0.128217	0.230683	

The 3 features that seem useful and their sign of association are

Humidity: negatively correlated, Solar radiation: positively correlated and Functioning_Day: positively correlated

Part b

In [26]:
```
X = train_data.drop(columns=['log_Rented_Bike_Count'])
y = train_data['log_Rented_Bike_Count']

X = sm.add_constant(X)# Adding intercept b0
model = sm.OLS(y, X).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
=====
Dep. Variable:     log_Rented_Bike_Count   R-squared:
0.862
Model:                               OLS   Adj. R-squared:
```

```
0.852
Method:                      Least Squares   F-statistic:
81.01
Date:                   Fri, 08 Nov 2024   Prob (F-statistic):           3.2
0e-65
Time:                           13:28:30   Log-Likelihood:                -1
69.94
No. Observations:                    182   AIC:
367.9
Df Residuals:                        168   BIC:
412.7
Df Model:                             13
Covariance Type:               nonrobust
==============================================================================
=============
                             coef    std err          t      P>|t|      [0.02
5      0.975]
------------------------------------------------------------------------------
-------------
const                      4.9318      1.055      4.674      0.000       2.84
9       7.015
Temperature               -0.1689      0.037     -4.594      0.000      -0.24
1      -0.096
Humidity                  -0.0869      0.012     -7.392      0.000      -0.11
0      -0.064
Wind_speed                 0.0357      0.058      0.617      0.538      -0.07
9       0.150
Visibility                -0.0001      0.000     -1.004      0.317      -0.00
0       0.000
Dew_point_temperature      0.2301      0.040      5.701      0.000       0.15
0       0.310
Solar_Radiation            0.2248      0.239      0.942      0.348      -0.24
6       0.696
Rainfall                  -0.0395      0.030     -1.332      0.185      -0.09
8       0.019
Snowfall                   0.1535      0.134      1.146      0.253      -0.11
1       0.418
Holiday                   -0.6899      0.254     -2.713      0.007      -1.19
2      -0.188
Functioning_Day            7.6250      0.283     26.905      0.000       7.06
6       8.185
SeasonsAutumn              0.9703      0.208      4.662      0.000       0.55
9       1.381
SeasonsSpring              0.5515      0.195      2.831      0.005       0.16
7       0.936
SeasonsSummer              0.2854      0.287      0.996      0.321      -0.28
0       0.851
==============================================================================
==
Omnibus:                       42.952   Durbin-Watson:                   1.7
```

```
50
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              106.7
82
Skew:                          -1.006   Prob(JB):                     6.49e-
24
Kurtosis:                       6.167   Cond. No.                     3.64e+
04
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
[2] The condition number is large, 3.64e+04. This might indicate that there
are
strong multicollinearity or other numerical problems.
```

Variables with low p-values (mostly below 0.05) are statistically significant predictors of log_Rented_Bike_Count. From the above summary the varibales that fall in this range are Temperature, Humidity, Dew_point_temperature, Holiday, Functioning_Day, SeasonsAutumn, SeasonsSpring.

The coefficient of temperature is counter intuitive from my findings in a. The correlation matrix shows a positive correlation of 0.317 between Temperature and log_Rented_Bike_Count, suggesting that as temperature increases, the log of rented bike count also tends to increase. However, the OLS summary shows a negative coefficient of -0.1689 for Temperature, indicating that in the context of the regression model, an increase in temperature is associated with a decrease in the log of rented bike count, all else being equal. Correlation is a pairwise measure—it looks at the relationship between Temperature and log_Rented_Bike_Count in isolation, without considering the other variables in the dataset. Regression Coefficient for Temperature reflects its effect on the target variable (log_Rented_Bike_Count) while controlling for other variables. This means that the negative sign in the regression might be due to how Temperature interacts with other predictors in the model

Part c

In [34]:
```
test_data = pd.read_csv('SeoulBikeData_6pm_test.csv')

X_train = train_data.drop(columns=['log_Rented_Bike_Count'])
y_train = train_data['log_Rented_Bike_Count']
X_test = test_data.drop(columns=['log_Rented_Bike_Count'])
y_test = test_data['log_Rented_Bike_Count']

results = []
```

In [36]:
```
for combo in itertools.combinations(X_train.columns, 3):
    X_train_subset = X_train[list(combo)]
    X_test_subset = X_test[list(combo)]
    X_train_subset = sm.add_constant(X_train_subset)
    X_test_subset = sm.add_constant(X_test_subset)

    model = sm.OLS(y_train, X_train_subset).fit()

    y_train_pred = model.predict(X_train_subset)
    y_test_pred = model.predict(X_test_subset)
    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_test_pred)

    results.append({
        'predictors': combo,
        'train_mse': train_mse,
        'test_mse': test_mse
    })

# Convert results to a DataFrame and find the best model based on test MSE
results_df = pd.DataFrame(results)
best_model = results_df.loc[results_df['test_mse'].idxmin()]

# Display the best model and its MSE values
print("Best 3-predictor model:")
print("Predictors:", best_model['predictors'])
print("Training MSE:", best_model['train_mse'])
print("Test MSE:", best_model['test_mse'])
```

```
Best 3-predictor model:
Predictors: ('Humidity', 'Dew_point_temperature', 'Functioning_Day')
Training MSE: 0.5653637352428881
Test MSE: 0.44996751473174373
```

Part d

In [207…
```
best_train_mse = float('inf')
best_test_mse = float('inf')

for predictor in X_train.columns:
    X_train_subset = sm.add_constant(X_train[[predictor]])
    X_test_subset = sm.add_constant(X_test[[predictor]])

    model = sm.OLS(y_train, X_train_subset).fit()
    train_mse=mean_squared_error(y_train, model.predict(X_train_subset))
    test_mse = mean_squared_error(y_test, model.predict(X_test_subset))

    if (test_mse < best_test_mse) & (train_mse < best_train_mse):
        best_test_mse = test_mse
```

```
        best_train_mse=train_mse
        best_single_predictor = predictor

print("Best single predictor:", best_single_predictor)
print("Single predictor train MSE:", best_train_mse)
print("Single predictor test MSE:", best_test_mse)

remaining_predictors = [p for p in X_train.columns if p != best_single_predi

for predictor in remaining_predictors:
    current_predictors = [best_single_predictor, predictor]
    X_train_subset = sm.add_constant(X_train[current_predictors])
    X_test_subset = sm.add_constant(X_test[current_predictors])

    model = sm.OLS(y_train, X_train_subset).fit()
    train_mse=mean_squared_error(y_train, model.predict(X_train_subset))
    test_mse = mean_squared_error(y_test, model.predict(X_test_subset))

    if (test_mse < best_test_mse) & (train_mse < best_train_mse):
        best_test_mse = test_mse
        best_train_mse=train_mse
        best_two_predictors = current_predictors

print("Best two predictors:", best_two_predictors)
print("Two predictor train MSE:", best_train_mse)
print("Two predictor test MSE:", best_test_mse)
remaining_predictors = [p for p in X_train.columns if p not in best_two_prec

for predictor in remaining_predictors:
    current_predictors = best_two_predictors + [predictor]
    X_train_subset = sm.add_constant(X_train[current_predictors])
    X_test_subset = sm.add_constant(X_test[current_predictors])

    model = sm.OLS(y_train, X_train_subset).fit()
    train_mse = mean_squared_error(y_train, model.predict(X_train_subset))
    test_mse = mean_squared_error(y_test, model.predict(X_test_subset))

    if (test_mse < best_test_mse) & (train_mse < best_train_mse):
        best_test_mse = test_mse
        best_train_mse=train_mse
        best_three_predictors = current_predictors

print("Best three predictors:", best_three_predictors)
print("Three predictor train MSE:", best_train_mse)
print("Three predictor test MSE:", best_test_mse)
```

```
Best single predictor: Functioning_Day
Single predictor train MSE: 1.2214187076814151
Single predictor test MSE: 1.017557579033773
Best two predictors: ['Functioning_Day', 'Temperature']
Two predictor train MSE: 0.8369727240383807
Two predictor test MSE: 0.6170820825304877
Best three predictors: ['Functioning_Day', 'Temperature', 'Humidity']
Three predictor train MSE: 0.611419497795392
Three predictor test MSE: 0.48187069670985916
```

Part e

The training MSE for the best subset selection method is slightly lower than the training MSE for the forward stepwise selection method. Best subset selection searches through all possible combinations of predictors to find the combination with the lowest training MSE, making it more likely to identify the model that best fits the training data. Forward stepwise selection, however, builds the model sequentially by adding one predictor at a time. This method may overlook the best subset if the optimal predictors aren't added early in the sequence, resulting in a higher training MSE.

The test MSE is slightly lower for the best subset selection model than for the forward stepwise selection model, indicating better generalization to new data. This could mean that the model selected through best subset selection has a more suitable combination of predictors for predicting on unseen data. The relatively small difference between the two test MSEs suggests that both methods have selected fairly effective models. However, best subset selection may have a slight edge in generalization because it considers all possible predictor combinations, leading to a model that is potentially more optimal overall.

PROBLEM 3

Part a

In [261…

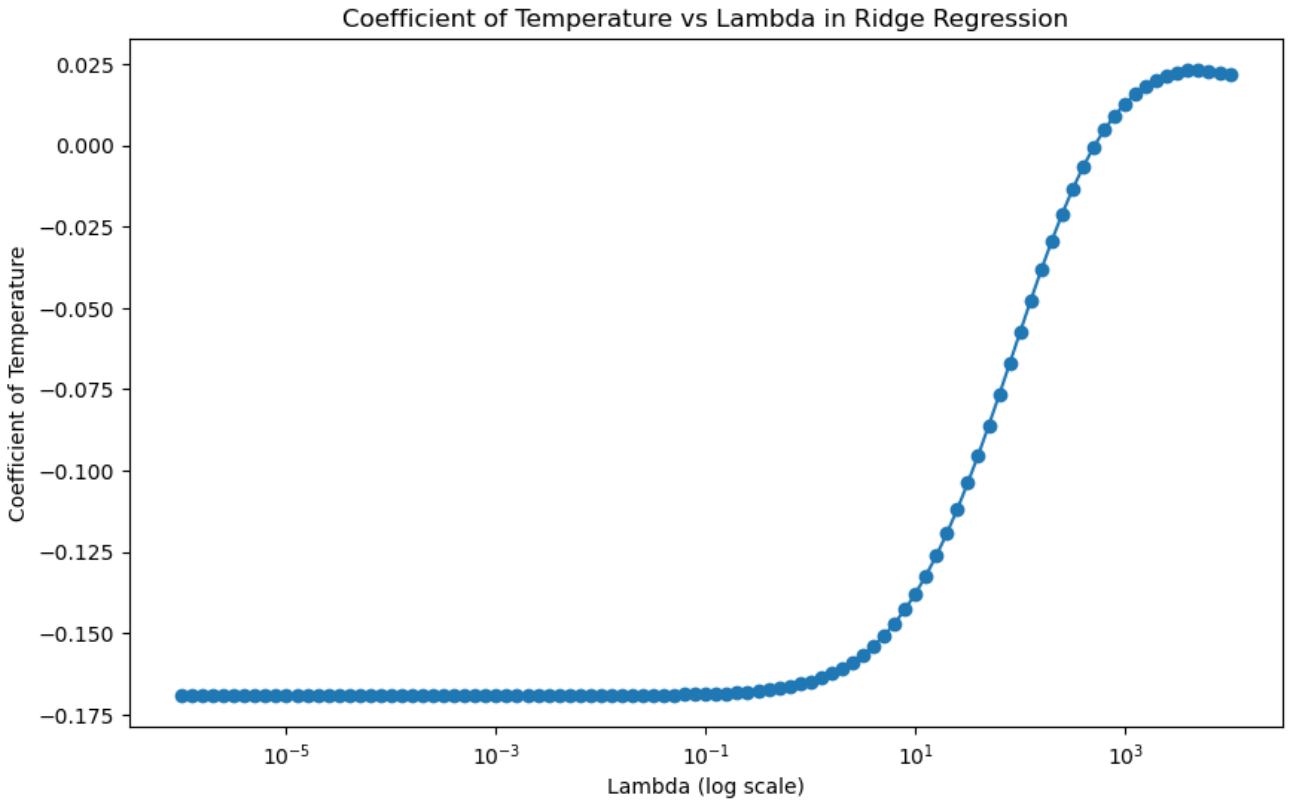
```
from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
```

In [291…

```
lambdas = 10**np.linspace(4, -6, 101)

kf = KFold(n_splits=10, shuffle=True, random_state=123)# assigning the menti
cv_errors = []
temperature_coefficients = []
```

```
for lambda_val in lambdas:
    fold_errors = []
    temp_coefs = []

    for train_index, val_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[v
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[v

        ridge_model = Ridge(alpha=lambda_val)
        ridge_model.fit(X_train_fold, y_train_fold)

        y_val_pred = ridge_model.predict(X_val_fold)
        fold_mse = mean_squared_error(y_val_fold, y_val_pred)
        fold_errors.append(fold_mse)

        temp_coefs.append(ridge_model.coef_[X_train.columns.get_loc("Tempera

    cv_errors.append(np.mean(fold_errors))
    temperature_coefficients.append(np.mean(temp_coefs))

optimal_lambda = lambdas[np.argmin(cv_errors)]
print(f"Optimal lambda (ridge): {optimal_lambda}")

plt.figure(figsize=(10, 6))
plt.plot(lambdas, temperature_coefficients, marker='o')
plt.xscale('log')
plt.xlabel("Lambda (log scale)")
plt.ylabel("Coefficient of Temperature")
plt.title("Coefficient of Temperature vs Lambda in Ridge Regression")
plt.show()
```

```
Optimal lambda (ridge): 0.0630957344480193
```



As λ increases, the ridge penalty becomes stronger, forcing the Temperature coefficient towards zero. This behavior is typical in ridge regression, where increasing λ reduces the absolute values of the coefficients, effectively shrinking them.For very small values of λ (around 10^-5), the coefficient of Temperature is close to the value it would have in an unregularized linear regression. As λ grows, the coefficient initially increases slightly before being driven down to zero. This slight increase might indicate some complex relationship with other variables or noise suppression effects at specific λ values.

Part b

In [297…

```
ridge_final_model = Ridge(alpha=optimal_lambda)
ridge_model.fit(X_train, y_train)

y_train_pred = ridge_model.predict(X_train)
y_test_pred = ridge_model.predict(X_test)

# Calculate MSE for training and test sets
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

# Print the MSE
print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

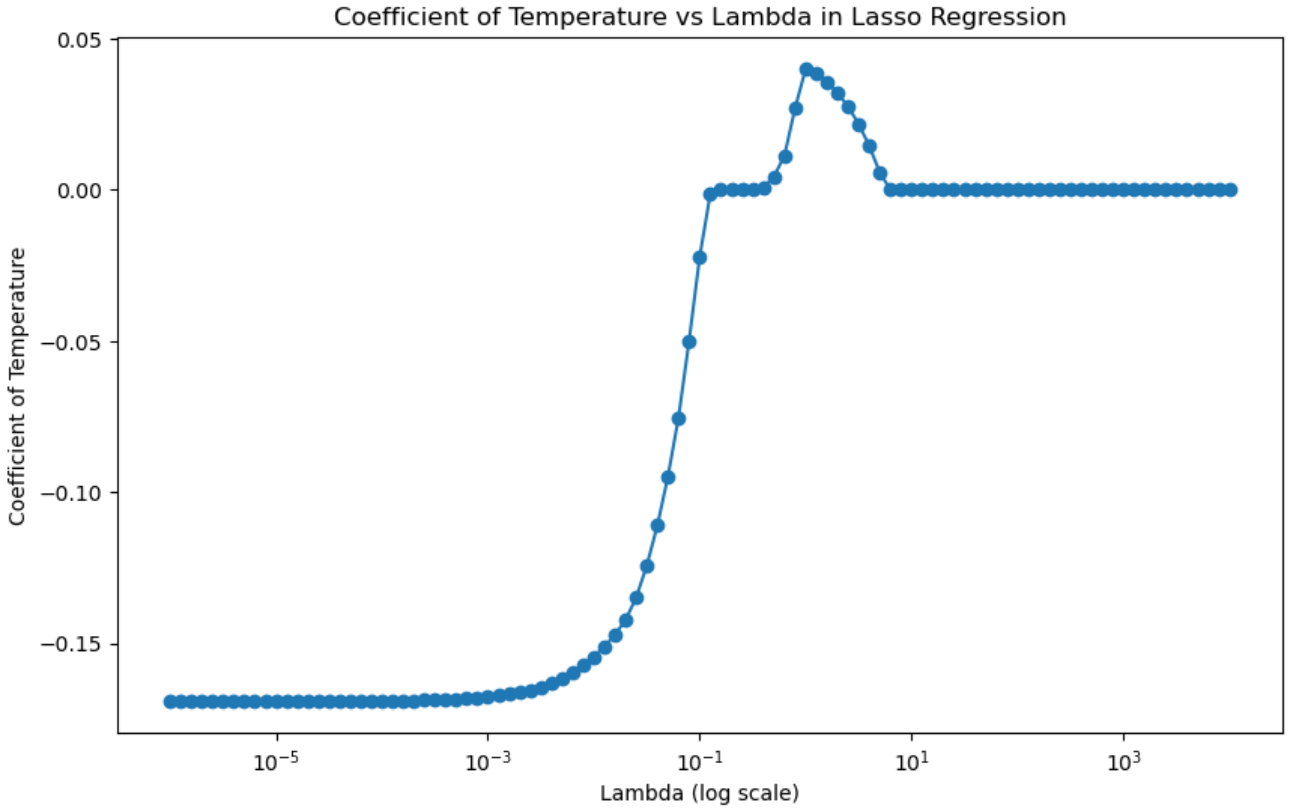
```
Training MSE: 0.3789139931937403
Test MSE: 0.3252878820049078
```

Part c

In [299…

```
cv_errors = []
temperature_coefficients = []

for lambda_val in lambdas:
    fold_errors = []
    temp_coefs = []

    for train_index, val_index in kf.split(X_train):
        X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[v
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[v

        lasso_model = Lasso(alpha=lambda_val, max_iter=10000)
        lasso_model.fit(X_train_fold, y_train_fold)

        y_val_pred = lasso_model.predict(X_val_fold)
        fold_mse = mean_squared_error(y_val_fold, y_val_pred)
        fold_errors.append(fold_mse)

        temp_coefs.append(lasso_model.coef_[X_train.columns.get_loc("Tempera

    cv_errors.append(np.mean(fold_errors))
    temperature_coefficients.append(np.mean(temp_coefs))

optimal_lambda = lambdas[np.argmin(cv_errors)]
print(f"Optimal lambda (lasso): {optimal_lambda}")

lasso_final = Lasso(alpha=optimal_lambda, max_iter=10000)
lasso_final.fit(X_train_scaled, y_train)
temperature_coefficient = lasso_final.coef_[X_train.columns.get_loc("Tempera
print(f"Coefficient of Temperature at optimal lambda: {temperature_coefficie

plt.figure(figsize=(10, 6))
plt.plot(lambdas, temperature_coefficients, marker='o')
plt.xscale('log')
plt.xlabel("Lambda (log scale)")
plt.ylabel("Coefficient of Temperature")
plt.title("Coefficient of Temperature vs Lambda in Lasso Regression")
plt.show()
```

```
Optimal lambda (lasso): 0.00630957344480193
Coefficient of Temperature at optimal lambda: -0.6010788295008944
```



As λ increases, the coefficient of Temperature decreases rapidly and then plateaus. This is a common pattern when using Lasso regression: as λ increases, the regularization penalty becomes more dominant, forcing the model to shrink some coefficients towards zero. The coefficients that are most important to the model will be the last to be shrunk to zero. In this case, the coefficient of Temperature appears to be fairly important until λ reaches a certain threshold, at which point it is effectively removed from the model.

Part d

In [302…

```
lasso_model = Lasso(optimal_lambda)  # Use optimal lambda
lasso_model.fit(X_train, y_train)

# Make predictions on training and test sets
y_train_pred = lasso_model.predict(X_train)
y_test_pred = lasso_model.predict(X_test)

# Calculate MSE for training and test sets
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

# Print the MSE
print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

```
Training MSE: 0.38616596992490754
Test MSE: 0.32571133216890297
```

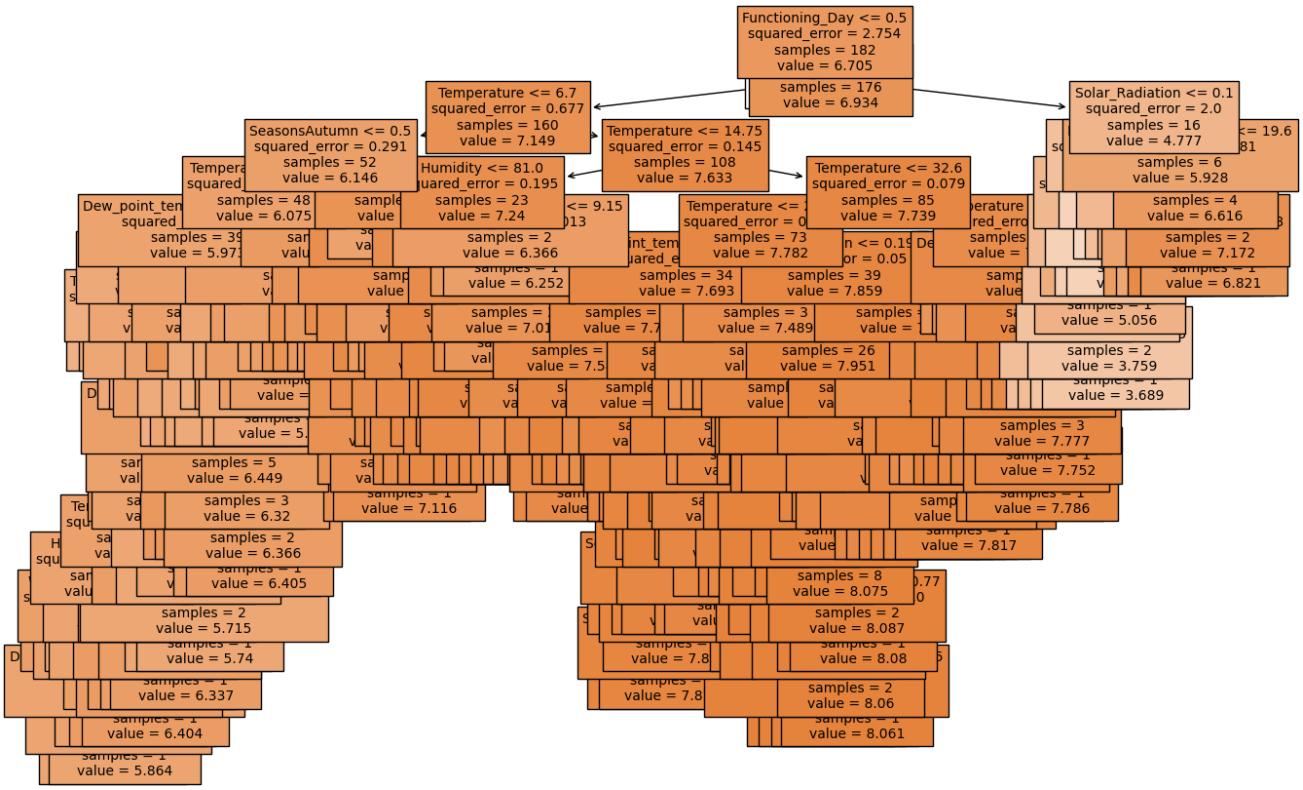
```
/opt/anaconda3/lib/python3.12/site-packages/sklearn/linear_model/_coordinate
_descent.py:678: ConvergenceWarning: Objective did not converge. You might w
ant to increase the number of iterations, check the scale of the features or
consider increasing regularisation. Duality gap: 5.980e-02, tolerance: 5.012
e-02
  model = cd_fast.enet_coordinate_descent(
```

PROBLEM 4

Part a

In [109…
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree
from sklearn.ensemble import RandomForestRegressor
```

In [316…
```
regressor = DecisionTreeRegressor(random_state=123)
regressor.fit(X_train, y_train)

plt.figure(figsize=(15,10))
plot_tree(regressor, filled=True, feature_names=X_train.columns, fontsize=10
plt.show()
```



Interpretation: This tree is very complex, with a significant number of splits and branches. This complexity suggests a potential overfitting issue, as the tree might be

capturing noise in the data instead of general trends. Pruning techniques can be applied to simplify the tree, removing less important branches to improve generalization.

In [305…
```
y_train_pred = regressor.predict(X_train)
y_test_pred = regressor.predict(X_test)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print("Training MSE:", train_mse)
print("Test MSE:", test_mse)
```

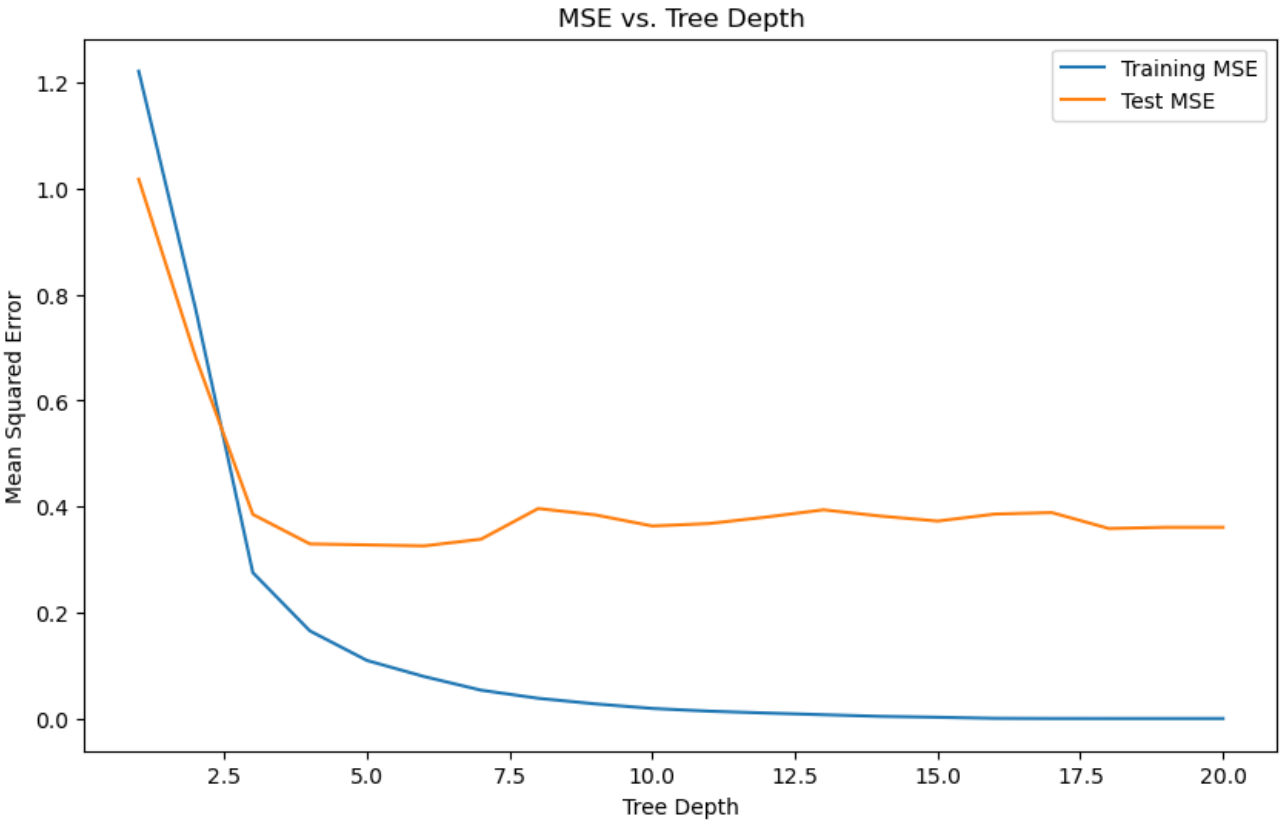
```
Training MSE: 0.0
Test MSE: 0.3608175370730489
```

Part b

Pruning in decision trees can be done by limiting the maximum depth of the tree, or by increasing the minimum number of samples required to be at a leaf node. we shall do this by varying the max_depth parameter and evaluating the MSE on both the training and test sets

In [103…
```
depths = np.arange(1, 21)

train_mse = []
test_mse = []

for depth in depths:
    # Fit the model with the given max_depth
    regressor = DecisionTreeRegressor(random_state=123, max_depth=depth)
    regressor.fit(X_train, y_train)

    # Predict and calculate the MSE for training and test sets
    y_train_pred = regressor.predict(X_train)
    y_test_pred = regressor.predict(X_test)

    train_mse.append(mean_squared_error(y_train, y_train_pred))
    test_mse.append(mean_squared_error(y_test, y_test_pred))

# Plot the MSE for different depths
plt.figure(figsize=(10, 6))
plt.plot(depths, train_mse, label="Training MSE")
plt.plot(depths, test_mse, label="Test MSE")
plt.xlabel("Tree Depth")
plt.ylabel("Mean Squared Error")
plt.legend()
plt.title("MSE vs. Tree Depth")
plt.show()
```

```
optimal_depth = depths[np.argmin(test_mse)]
print("Optimal Depth:", optimal_depth)
```




```
Optimal Depth: 6
```

The point where the test MSE stabilizes or starts to increase indicates the optimal depth for pruning

In [105…

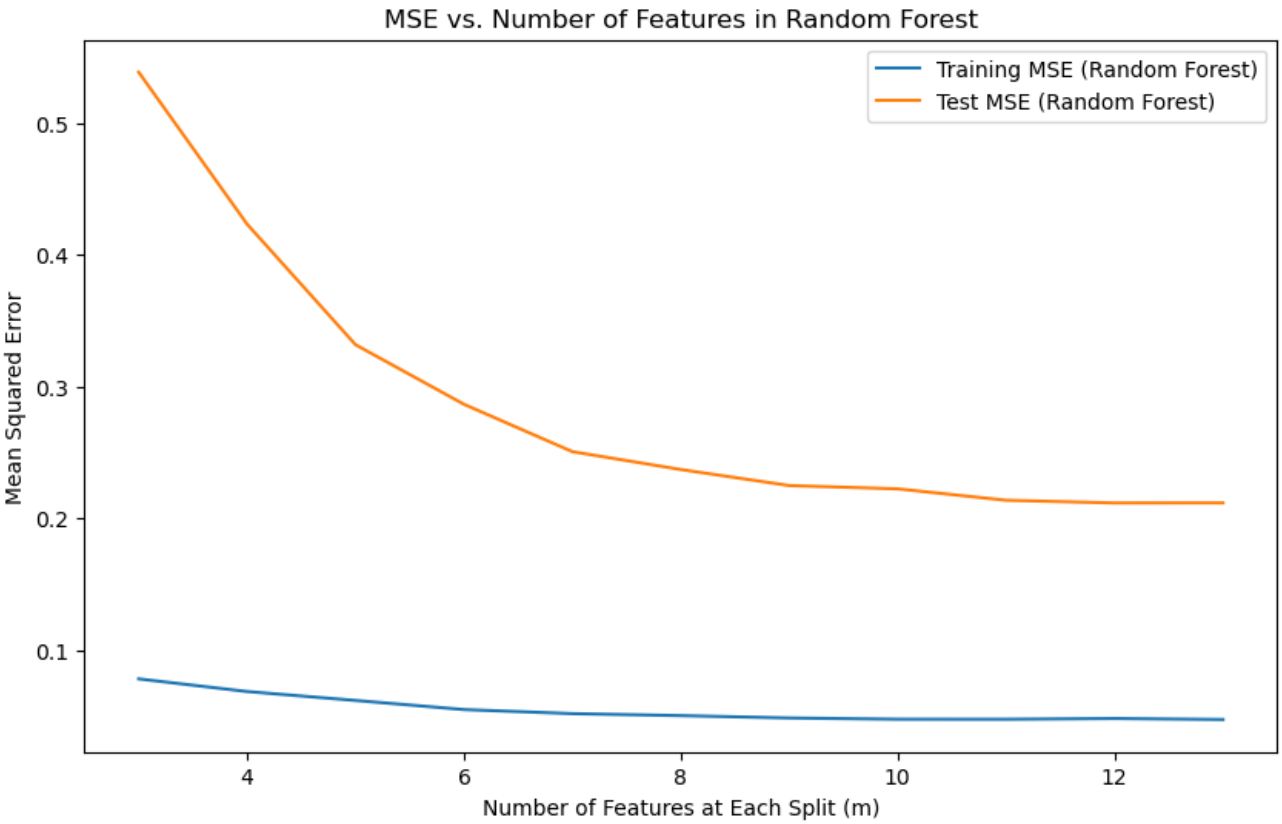
```
pruned_regressor = DecisionTreeRegressor(random_state=123, max_depth=optimal
pruned_regressor.fit(X_train, y_train)

# Predictions and MSE for pruned tree
y_train_pred_pruned = pruned_regressor.predict(X_train)
y_test_pred_pruned = pruned_regressor.predict(X_test)

train_mse_pruned = mean_squared_error(y_train, y_train_pred_pruned)
test_mse_pruned = mean_squared_error(y_test, y_test_pred_pruned)

# Output the MSE
print("Training MSE (Pruned):", train_mse_pruned)
print("Test MSE (Pruned):", test_mse_pruned)
```

```
Training MSE (Pruned): 0.07920199254516838
Test MSE (Pruned): 0.32568392424604925
```

Part c

In [112…

```
m_values = np.arange(3, 14)

train_mse_rf = []
test_mse_rf = []
important_features_rf = []

for m in m_values:
    # Train a random forest with m features at each split
    rf_regressor = RandomForestRegressor(n_estimators=400, max_features=m, r
    rf_regressor.fit(X_train, y_train)

    # Predict and calculate the MSE for training and test sets
    y_train_pred_rf = rf_regressor.predict(X_train)
    y_test_pred_rf = rf_regressor.predict(X_test)

    train_mse_rf.append(mean_squared_error(y_train, y_train_pred_rf))
    test_mse_rf.append(mean_squared_error(y_test, y_test_pred_rf))

    # Get the most important features
    feature_importances = rf_regressor.feature_importances_
    important_features_idx = np.argsort(feature_importances)[-3:]  # Indices
    important_features_rf.append(X_train.columns[important_features_idx])

# Plot the MSE for different m values
plt.figure(figsize=(10, 6))
plt.plot(m_values, train_mse_rf, label="Training MSE (Random Forest)")
plt.plot(m_values, test_mse_rf, label="Test MSE (Random Forest)")
plt.xlabel("Number of Features at Each Split (m)")
plt.ylabel("Mean Squared Error")
plt.legend()
plt.title("MSE vs. Number of Features in Random Forest")
plt.show()

# Print the optimal m and corresponding most important features
optimal_m_rf = m_values[np.argmin(test_mse_rf)]
print("Optimal m (number of features):", optimal_m_rf)
print("\nTop 3 important features for each m:")

for i, m in enumerate(m_values):
    print(f"m = {m}: {', '.join(important_features_rf[i])}")

# Comparing Random Forest with Regression Tree:
print("\nRandom Forest vs Regression Tree:")
train_mse_tree = mean_squared_error(y_train, y_train_pred)
test_mse_tree = mean_squared_error(y_test, y_test_pred)
print(f"Regression Tree - Training MSE: {train_mse_tree}, Test MSE: {test_ms
print(f"Random Forest - Training MSE: {min(train_mse_rf)}, Test MSE: {min(te
```




```
Optimal m (number of features): 12

Top 3 important features for each m:
m = 3: Solar_Radiation, Temperature, Functioning_Day
m = 4: Solar_Radiation, Temperature, Functioning_Day
m = 5: Temperature, Solar_Radiation, Functioning_Day
m = 6: Solar_Radiation, Temperature, Functioning_Day
m = 7: Solar_Radiation, Temperature, Functioning_Day
m = 8: Rainfall, Temperature, Functioning_Day
m = 9: Rainfall, Temperature, Functioning_Day
m = 10: Rainfall, Temperature, Functioning_Day
m = 11: Rainfall, Temperature, Functioning_Day
m = 12: Rainfall, Temperature, Functioning_Day
m = 13: Rainfall, Temperature, Functioning_Day

Random Forest vs Regression Tree:
Regression Tree - Training MSE: 0.0, Test MSE: 0.3608175370730489
Random Forest - Training MSE: 0.04745892526885836, Test MSE: 0.2118138020485
6373
```

Yes, the Random Forest significantly improves upon the regression tree. The test MSE of the Random Forest is almost half that of the regression tree, which indicates better generalization and less overfitting. Random forests combine multiple decision trees (each with a different set of randomly selected features). This averaging effect reduces variance and improves generalization. The random selection of features at each split

prevents any single tree from being too strongly influenced by a particular feature, helping to avoid overfitting. Random forests are robust to outliers and noise in the data due to the averaging of multiple trees.

In [ ]: