1. To fit a cubic spline, we define basis functions such that the spline is cubic between each pair of knots and continuous at the knots. A cubic spline regression model can be expressed as:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \sum_{j=1}^{K} \gamma_j (x_i - \xi_j)_+^3 + \epsilon_i$$

where $(x_i - \xi_j)_+^3$ is the truncated power basis function for the cubic spline. Here, β are the coefficients for the polynomial terms, and γ are the coefficients for the basis functions associated with the knots.

The task of fitting this model can be solved as a linear regression problem by constructing a design matrix with columns for x_i, x_i^2, x_i^3, and each $(x_i - \xi_j)_+^3$. The equality constraints ensure that the function and its derivatives are continuous at each knot. These constraints can be added to the linear regression formulation, transforming the problem into a constrained linear regression task.

2.

- Piecewise Polynomial Regression divides the data range into intervals and fits a separate polynomial in each interval. The intervals are defined by knots, and continuity constraints may be applied at the knots. This method is often used in spline regression.
- Local Polynomial Regression, on the other hand, fits a polynomial regression in a neighborhood around each point, weighting the nearby points more heavily. The size of this neighborhood is determined by a bandwidth parameter h. Local polynomial regression is used for nonparametric smoothing.

The main difference is that piecewise polynomial regression is globally segmented, while local polynomial regression is locally adaptive to each data point.

3.

- Bias: Increasing the bandwidth h typically increases the bias in local polynomial regression because a larger h means that more distant points are included, leading to a smoother, less flexible fit that may not capture local variations well.
- Variance: Increasing the bandwidth h decreases the variance because the model is less sensitive to fluctuations in the data. With a larger h, individual data points have less impact, and the model becomes more stable.

Thus, there is a bias-variance trade-off in choosing the bandwidth: a smaller h leads to lower bias but higher variance, and a larger h leads to higher bias but lower variance.

4. Answer: (b) Piecewise constant regression

Explanation: A regression tree divides the data space into regions and fits a constant value in each region, similar to piecewise constant regression. In both methods, the data range is segmented into intervals, and within each segment, a constant (usually the mean of the response variable within that segment) is predicted.

The difference is that regression trees automatically determine the splits based on minimizing the prediction error, whereas piecewise constant regression usually requires predefined segments.

5.

- Bias: As the tree gets finer (deeper), it becomes more flexible, reducing bias because the model can better capture small patterns and fit closely to the training data.
- Variance: A finer tree has higher variance because it fits closely to the training data, which may include noise. This can make the model more sensitive to variations in the data and reduce its ability to generalize to new data.

Therefore, a finer tree has lower bias but higher variance, leading to overfitting if the tree is too fine.

6. Answer: (c) Natural splines

Explanation: Natural splines have the smallest bias among the three because they allow for more flexibility in fitting the data, especially in regions with high variability, while controlling for excessive oscillation at the boundaries. Linear regression is constrained by a linear form and has a high bias if the true relationship is nonlinear. Regression trees are flexible but tend to have piecewise constant predictions, which can also lead to bias in capturing smooth trends.

Natural splines provide a balance of flexibility and smoothness, allowing for low bias in capturing complex relationships.

7. Variable Importance Definition

In a random forest, variable importance measures how much each feature contributes to the overall model accuracy:

1. For Regression Trees: The importance of a predictor j is determined by the reduction in residual sum of squares (RSS) across all trees. If $RSS_b(j)$ is the decrease in RSS for splits on j in the b-th tree, the importance score is:

$$\mathrm{VI}(j) = \frac{1}{B} \sum_{b=1}^{B} \sum_{\text{splits on } j} RSS_b(j)$$

A larger $\mathrm{VI}(j)$ suggests a more significant predictor.

2. For Classification Trees: Importance is measured by the reduction in the Gini index. If $G_b(j)$ is the decrease in the Gini index for splits on j in the b-th tree, then:

$$\mathrm{VI}(j) = \frac{1}{B} \sum_{b=1}^{B} \sum_{\text{splits on } j} G_b(j)$$

A higher $\mathrm{VI}(j)$ value indicates greater predictor importance.

Variable Selection/Model Selection in Random Forests

To select important variables, calculate each predictor's importance score, rank them, and filter out those with low scores. This process can be repeated to refine the model with only the most significant predictors, enhancing both efficiency and performance.

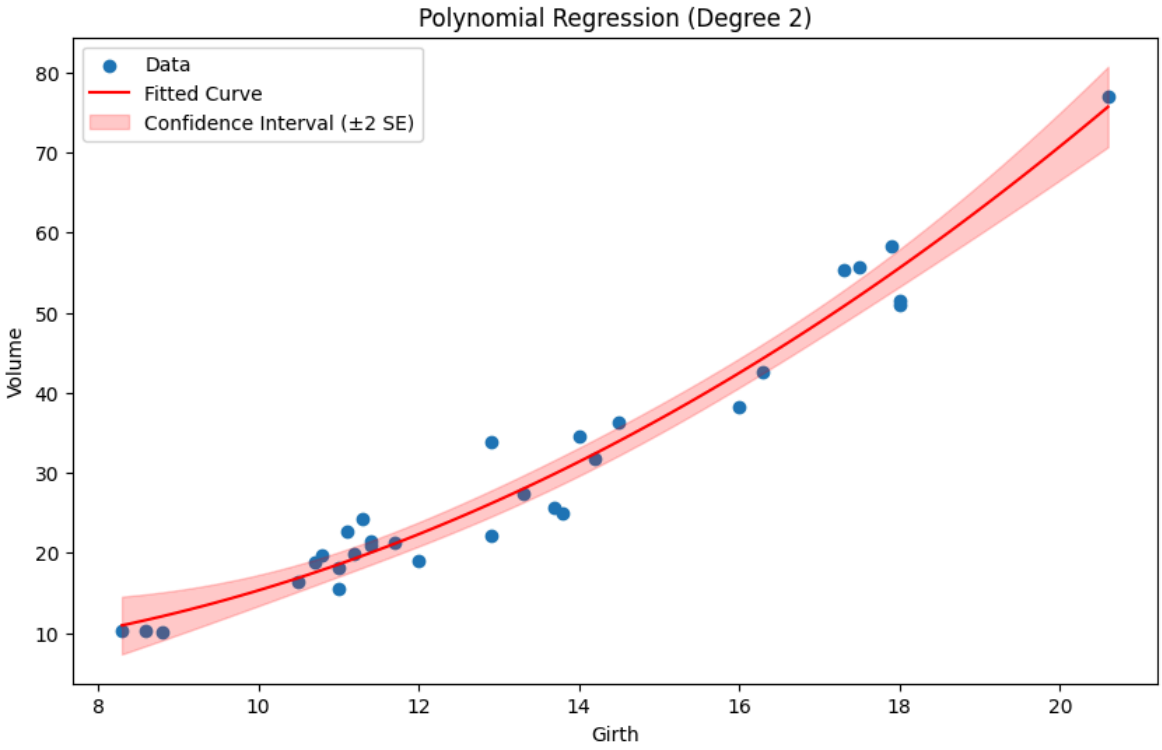
In [7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error
from patsy import dmatrix
from pygam import LinearGAM, LogisticGAM, s
import warnings
warnings.filterwarnings('ignore')

data = pd.read_csv('trees.csv')

# Task 1: Fit a polynomial model (deg=1,2,3,4) to the relationship betwee
X = data['Girth'].values.reshape(-1, 1)
y = data['Volume'].values

degrees = [1, 2, 3, 4]
adj_r2_list = []
models = {}

for degree in degrees:
    poly = PolynomialFeatures(degree)
    X_poly = poly.fit_transform(X)
    model = sm.OLS(y, X_poly).fit()
    adj_r2 = model.rsquared_adj
    adj_r2_list.append(adj_r2)
    # Save model and poly together
    models[degree] = (model, poly)
    print(f'Degree {degree}: Adjusted R-squared = {adj_r2}')

# Select the model with the highest adjusted R-squared
best_degree = degrees[np.argmax(adj_r2_list)]
best_model, best_poly = models[best_degree]
print(f'Best model degree: {best_degree}')

# Plot the polynomial function and confidence interval
X_plot = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
X_plot_poly = best_poly.transform(X_plot)

predictions = best_model.get_prediction(X_plot_poly)
pred_summary = predictions.summary_frame()
```

```
plt.figure(figsize=(10, 6))
plt.scatter(X, y, label='Data')
plt.plot(X_plot, pred_summary['mean'], color='red', label='Fitted Curve')
plt.fill_between(X_plot.flatten(),
                 pred_summary['mean_ci_lower'],
                 pred_summary['mean_ci_upper'],
                 color='red', alpha=0.2, label='Confidence Interval (±2 S
plt.xlabel('Girth')
plt.ylabel('Volume')
plt.title(f'Polynomial Regression (Degree {best_degree})')
plt.legend()
plt.show()

# Select model by 5-fold cross-validation error
cv_errors = []

for degree in degrees:
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    scores = cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared
    mse = -scores.mean()
    cv_errors.append(mse)
    print(f'Degree {degree}: 5-fold CV MSE = {mse}')

best_cv_degree = degrees[np.argmin(cv_errors)]
print(f'Best model degree by CV: {best_cv_degree}')

# Task 2: Use polynomial logistic regression (deg=2) to predict whether V
data['Volume_gt_30'] = (data['Volume'] > 30).astype(int)
y_binary = data['Volume_gt_30'].values

poly = PolynomialFeatures(2)
X_poly = poly.fit_transform(X)

# Fit logistic regression model
logit_model = sm.Logit(y_binary, X_poly).fit()
print(logit_model.summary())

# Plot the relationship between π(x) and Girth with confidence interval
X_plot_poly = poly.transform(X_plot)
predictions = logit_model.get_prediction(X_plot_poly)
pred_summary = predictions.summary_frame()

print("pred_summary:", pred_summary)

# 提取概率预测和标准误差
pi = pred_summary['predicted']  # 预测概率
se = pred_summary['se']         # 标准误差

# 计算置信区间上下界
ci_lower = pi - 2 * se
ci_upper = pi + 2 * se

# 绘图
plt.figure(figsize=(10, 6))
plt.scatter(X, y_binary, label='Data', alpha=0.5)
plt.plot(X_plot, pi, color='red', label='Predicted Probability π(x)')
plt.fill_between(X_plot.flatten(), ci_lower, ci_upper, color='red', alpha
                 label='Confidence Interval (±2 SE)')
plt.xlabel('Girth')
```

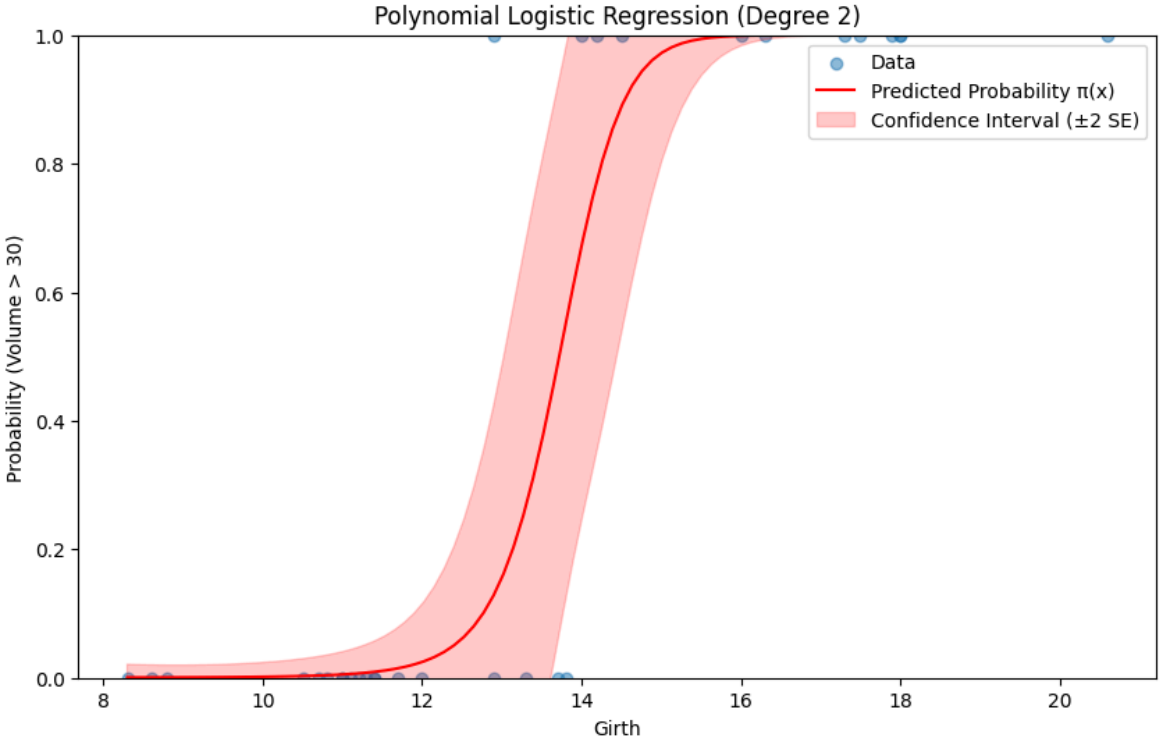
```
plt.ylabel('Probability (Volume > 30)')
plt.title('Polynomial Logistic Regression (Degree 2)')
plt.ylim(0, 1)  # 确保 y 轴范围在 [0, 1] 之间
plt.legend()
plt.show()

# Task 3: Use regression splines (deg=2) to fit the relationship between
knots = [10, 14, 18]

# Create spline basis functions
transformed_x = dmatrix("bs(x, knots=knots, degree=2, include_intercept=F
                        {"x": X.flatten(), "knots": knots}, return_type='

# Fit model
spline_model = sm.OLS(y, transformed_x).fit()
print(spline_model.summary())

# Plot function and confidence interval
X_plot_flat = X_plot.flatten()
transformed_x_plot = dmatrix("bs(x, knots=knots, degree=2, include_interc
                             {"x": X_plot_flat, "knots": knots}, return_t

predictions = spline_model.get_prediction(transformed_x_plot)
pred_summary = predictions.summary_frame()

plt.figure(figsize=(10, 6))
plt.scatter(X.flatten(), y, label='Data')
plt.plot(X_plot_flat, pred_summary['mean'], color='red', label='Regressio
plt.fill_between(X_plot_flat, pred_summary['mean_ci_lower'], pred_summary
                 color='red', alpha=0.2, label='Confidence Interval (±2 S
plt.xlabel('Girth')
plt.ylabel('Volume')
plt.title('Regression Spline (Degree=2) with Knots at 10, 14, 18')
plt.legend()
plt.show()

# Task 4: Use smoothing splines to fit the relationship between Volume an
from pygam import LinearGAM, s

X = data['Girth'].values.reshape(-1, 1)
y = data['Volume'].values

# Build GAM model and select smoothing parameter by cross-validation
gam = LinearGAM(s(0)).gridsearch(X, y)

# Get effective degrees of freedom
edf = gam.statistics_['edof']
print(f'Effective Degrees of Freedom: {edf:.2f}')

# Generate predictions and confidence interval
X_plot = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
preds = gam.predict(X_plot)
confi = gam.confidence_intervals(X_plot, width=0.95)

# Plot results
plt.figure(figsize=(10, 6))
plt.scatter(X, y, label='Data')
plt.plot(X_plot, preds, color='red', label='Smoothing Spline')
plt.fill_between(X_plot.flatten(),
                 confi[:, 0],
```

```
                confi[:, 1],
                color='red', alpha=0.2, label='Confidence Interval (±2 S
plt.xlabel('Girth')
plt.ylabel('Volume')
plt.title(f'Smoothing Spline (Effective Degrees of Freedom: {edf:.2f})')
plt.legend()
plt.show()

# Task 5: Use Girth and Height as variables to predict Volume with a gene
X_multi = data[['Girth', 'Height']].values
y = data['Volume'].values

# Build GAM model and specify degrees of freedom for smoothing splines
gam = LinearGAM(s(0, n_splines=4) + s(1, n_splines=5)).gridsearch(X_multi

fig, axs = plt.subplots(1, 2, figsize=(15, 5))
terms = ['Girth', 'Height']

for i, ax in enumerate(axs):
    XX = gam.generate_X_grid(term=i)
    # 使用 predict 而不是 partial_dependence 来保持一致性
    pdep = gam.predict(XX)

    # 获取 95% 或更宽的置信区间
    confi = gam.confidence_intervals(XX, width=0.95)

    ax.plot(XX[:, i], pdep, label='Partial Effect', color='red')
    ax.fill_between(XX[:, i], confi[:, 0], confi[:, 1], color='red', alph
                    label='Confidence Interval (±2 SE)')
    ax.set_title(f'Effect of {terms[i]}')
    ax.set_xlabel(terms[i])
    ax.set_ylabel('Partial Dependence')
    ax.legend()

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

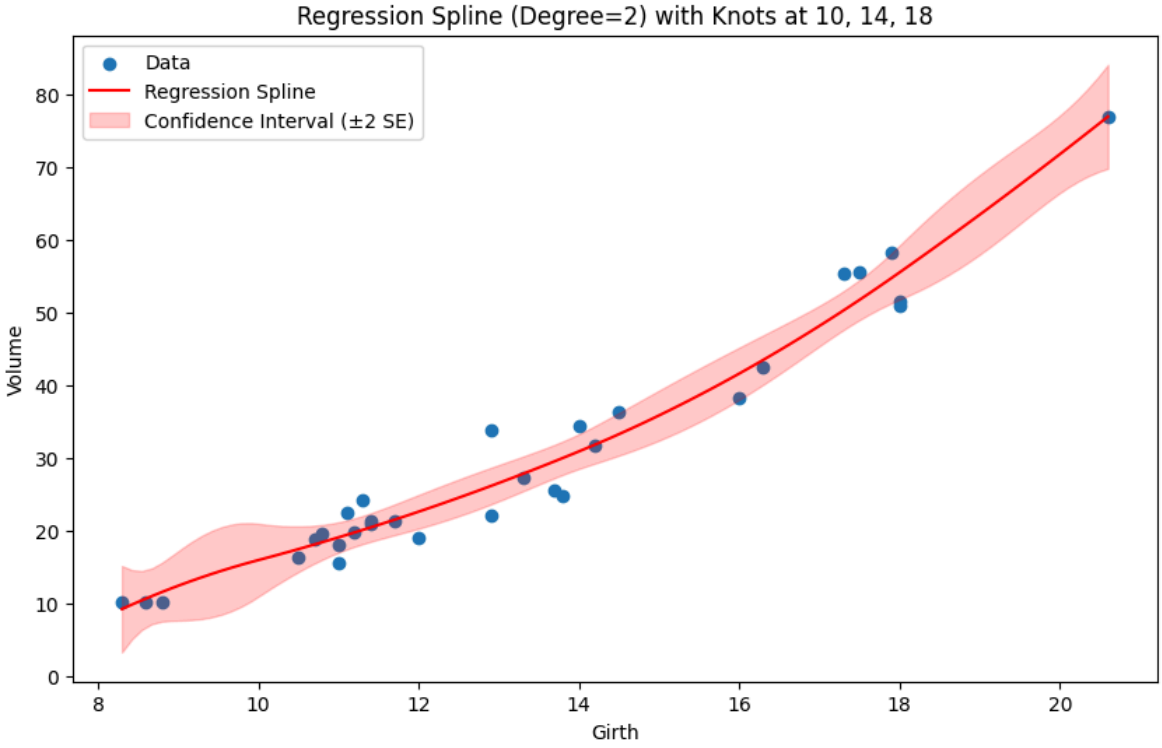
```
Degree 1: Adjusted R-squared = 0.9330895232294862
Degree 2: Adjusted R-squared = 0.9588428034831344
Degree 3: Adjusted R-squared = 0.9585798195179375
Degree 4: Adjusted R-squared = 0.9577192072341696
Best model degree: 2
```


Polynomial Regression (Degree 2)
Data
Fitted Curve
Confidence Interval (±2 SE)
Volume
Girth
80
70
60
50
40
30
20
10
8
10
12
14
16
18
20

```
Degree 1: 5-fold CV MSE = 44.79033337056526
Degree 2: 5-fold CV MSE = 26.83829725823664
Degree 3: 5-fold CV MSE = 22.11465817962842
Degree 4: 5-fold CV MSE = 38.22357803772734
Best model degree by CV: 3
Optimization terminated successfully.
         Current function value: 0.155953
         Iterations 11
                           Logit Regression Results
==============================================================================
====
Dep. Variable:                      y   No. Observations:
31
Model:                          Logit   Df Residuals:
28
Method:                           MLE   Df Model:
2
Date:                Fri, 08 Nov 2024   Pseudo R-squ.:                  0.
7663
Time:                        23:16:07   Log-Likelihood:                -4.
8345
converged:                       True   LL-Null:                       -2
0.690
Covariance Type:            nonrobust   LLR p-value:                1.300
e-07
==============================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
------------------------------------------------------------------------------
----
const          5.6239    120.503      0.047      0.963    -230.558     24
1.806
x1            -3.3205     18.765     -0.177      0.860     -40.099      3
3.458
x2             0.2121      0.729      0.291      0.771      -1.216
1.641
==============================================================================
====

Possibly complete quasi-separation: A fraction 0.19 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.
pred_summary:     predicted            se      ci_lower  ci_upper
0    0.000660  1.017983e-02  4.872362e-17       1.0
1    0.000679  9.925777e-03  2.419934e-16       1.0
2    0.000703  9.728412e-03  1.156187e-15       1.0
3    0.000733  9.584008e-03  5.313290e-15       1.0
4    0.000769  9.489726e-03  2.348308e-14       1.0
..        ...           ...           ...       ...
95   1.000000  7.902019e-10  1.408445e-22       1.0
96   1.000000  4.266638e-10  2.005741e-23       1.0
97   1.000000  2.287384e-10  2.751320e-24       1.0
98   1.000000  1.217605e-10  3.635273e-25       1.0
99   1.000000  6.435697e-11  4.626593e-26       1.0

[100 rows x 4 columns]
```

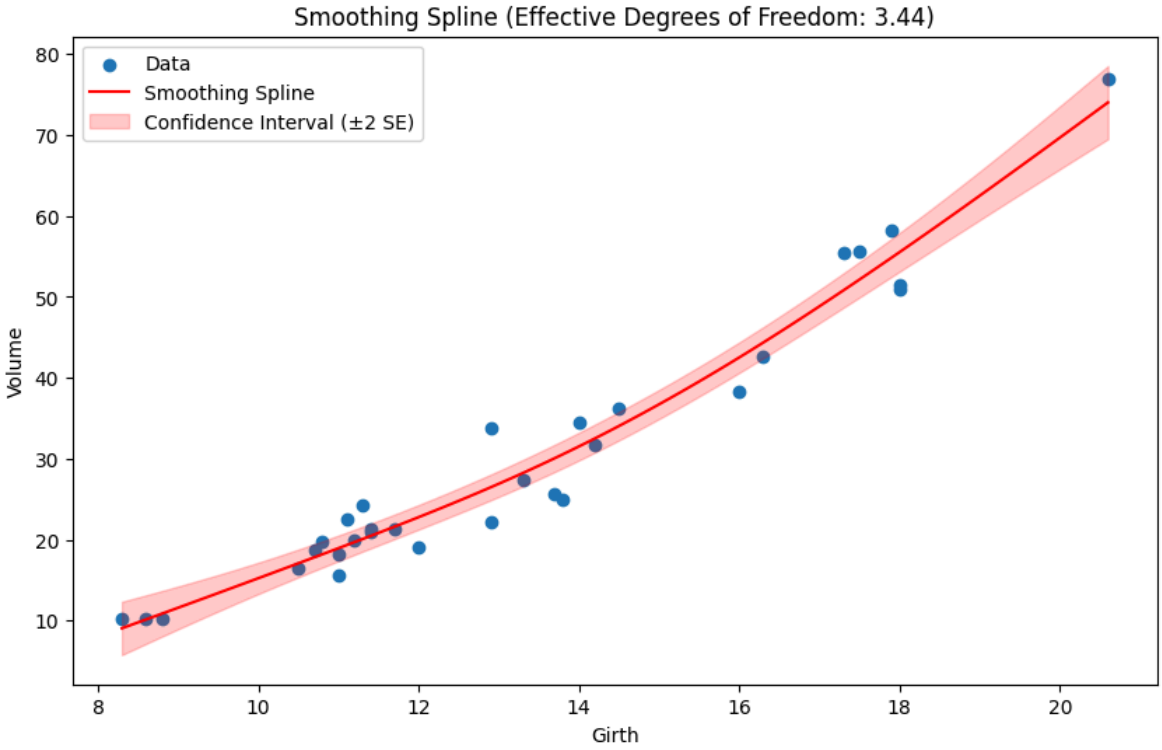

Polynomial Logistic Regression (Degree 2)
Data
Predicted Probability π(x)
Confidence Interval (±2 SE)
Probability (Volume > 30)
Girth
0.0
0.2
0.4
0.6
0.8
1.0
8
10
12
14
16
18
20

```
                            OLS Regression Results
==============================================================================
====
Dep. Variable:                        y   R-squared:
0.963
Model:                              OLS   Adj. R-squared:
0.956
Method:                   Least Squares   F-statistic:                      1
30.1
Date:                Fri, 08 Nov 2024   Prob (F-statistic):           4.60
e-17
Time:                        23:16:08   Log-Likelihood:                 -7
9.163
No. Observations:                  31   AIC:                              1
70.3
Df Residuals:                      25   BIC:                              1
78.9
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
==============================================
                                                                 coef    std
err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
------------------------------------------------
Intercept                                                      9.2913
2.876      3.231      0.003       3.369      15.214
bs(x, knots=knots, degree=2, include_intercept=False)[0]     4.2414
6.069      0.699      0.491      -8.258      16.741
bs(x, knots=knots, degree=2, include_intercept=False)[1]    12.5718
3.297      3.813      0.001       5.781      19.363
bs(x, knots=knots, degree=2, include_intercept=False)[2]    30.7711
4.810      6.397      0.000      20.864      40.678
bs(x, knots=knots, degree=2, include_intercept=False)[3]    56.3652
4.720     11.942      0.000      46.644      66.086
bs(x, knots=knots, degree=2, include_intercept=False)[4]    67.7087
4.502     15.040      0.000      58.437      76.980
==============================================================================
====
Omnibus:                        0.692   Durbin-Watson:
1.925
Prob(Omnibus):                  0.708   Jarque-Bera (JB):
0.755
Skew:                           0.301   Prob(JB):
0.686
Kurtosis:                       2.529   Cond. No.
14.9
==============================================================================
====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is cor
rectly specified.
```

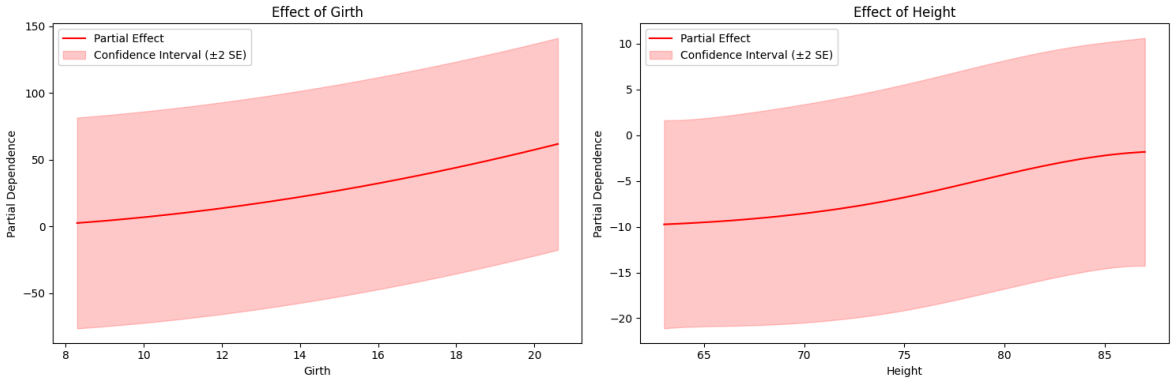



```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:
00:000:00
```

```
Effective Degrees of Freedom: 3.44
```




```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:
00:000:00
```

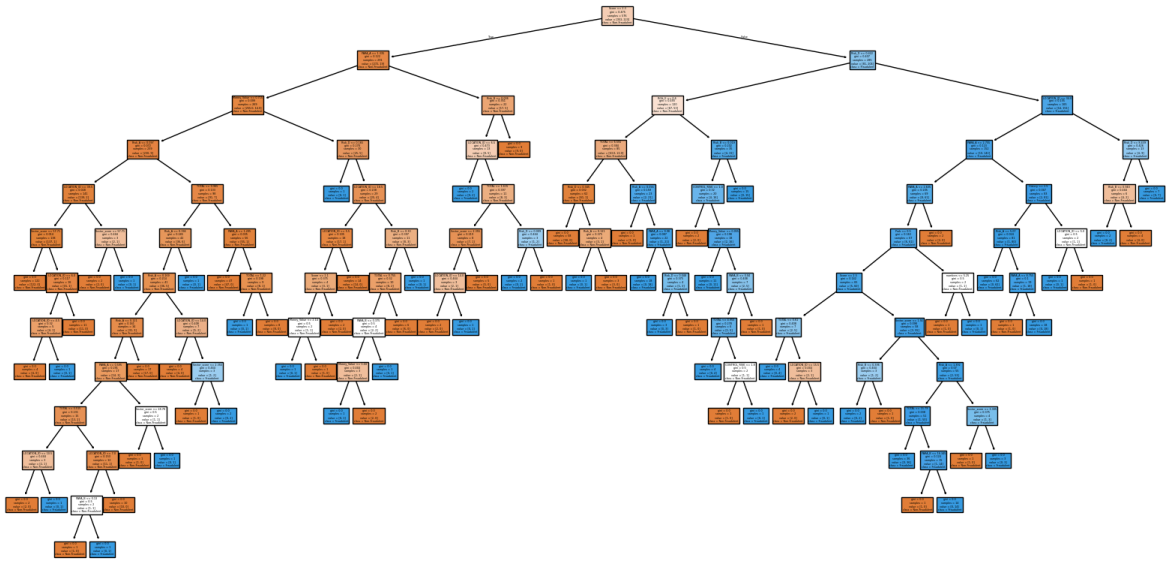


In [2]:
```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt

# Load datasets
train_data = pd.read_csv("audit_train.csv")
test_data = pd.read_csv("audit_test.csv")

# Check and process non-numeric data
# Identify non-numeric columns in the training set
non_numeric_cols = train_data.select_dtypes(include=['object']).columns
print("Non-numeric columns in training data:", non_numeric_cols)

# Convert non-numeric columns to numeric encoding if necessary
for col in non_numeric_cols:
    train_data[col] = pd.to_numeric(train_data[col], errors='coerce')
    test_data[col] = pd.to_numeric(test_data[col], errors='coerce')

# Fill missing values with mean (you can choose other methods based on ne
train_data.fillna(train_data.mean(), inplace=True)
test_data.fillna(test_data.mean(), inplace=True)

# Define features and target variable
X_train = train_data.drop("Risk", axis=1)
y_train = train_data["Risk"]
X_test = test_data.drop("Risk", axis=1)
y_test = test_data["Risk"]

# 1. Train a classification tree and plot the decision tree
clf_tree = DecisionTreeClassifier(random_state=0)
clf_tree.fit(X_train, y_train)

# Plot the decision tree
plt.figure(figsize=(20, 10))
plot_tree(clf_tree, filled=True, feature_names=X_train.columns, class_nam
plt.show()

# Calculate training error
y_train_pred = clf_tree.predict(X_train)
train_accuracy = accuracy_score(y_train, y_train_pred)
train_error = 1 - train_accuracy
print("Training Error (Decision Tree):", train_error)

# Evaluate performance on the test set and report the confusion matrix
```

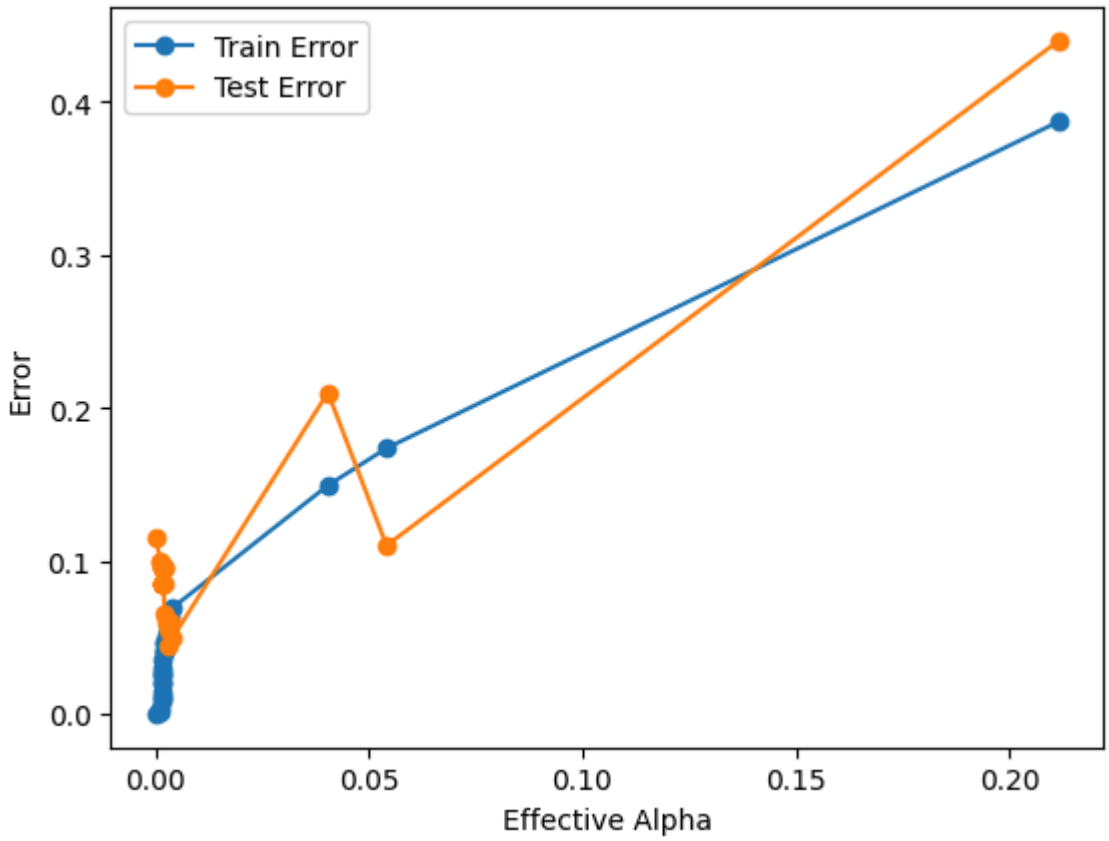
```
y_test_pred = clf_tree.predict(X_test)
test_conf_matrix = confusion_matrix(y_test, y_test_pred)
print("Confusion Matrix (Test):\n", test_conf_matrix)
test_accuracy = accuracy_score(y_test, y_test_pred)
test_error = 1 - test_accuracy
print("Test Error (Decision Tree):", test_error)

# 2. Prune the classification tree using cross-validation
path = clf_tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

train_errors = []
test_errors = []

for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)

    # Calculate training error
    y_train_pred = clf.predict(X_train)
    train_errors.append(1 - accuracy_score(y_train, y_train_pred))

    # Calculate test error
    y_test_pred = clf.predict(X_test)
    test_errors.append(1 - accuracy_score(y_test, y_test_pred))

# Plot the relationship between training error and tree size
plt.figure()
plt.plot(ccp_alphas, train_errors, marker='o', label='Train Error')
plt.plot(ccp_alphas, test_errors, marker='o', label='Test Error')
plt.xlabel("Effective Alpha")
plt.ylabel("Error")
plt.legend()
plt.show()

# Select the best pruned tree
optimal_alpha = ccp_alphas[np.argmin(test_errors)]
pruned_tree = DecisionTreeClassifier(random_state=0, ccp_alpha=optimal_al
pruned_tree.fit(X_train, y_train)

# Report test error after pruning
y_test_pred_pruned = pruned_tree.predict(X_test)
pruned_test_error = 1 - accuracy_score(y_test, y_test_pred_pruned)
print("Test Error (Pruned Tree):", pruned_test_error)

# 3. Use random forest for classification and report training error
rf_clf = RandomForestClassifier(n_estimators=25, max_features=13, random_
rf_clf.fit(X_train, y_train)

# Calculate training error
y_train_pred_rf = rf_clf.predict(X_train)
train_accuracy_rf = accuracy_score(y_train, y_train_pred_rf)
train_error_rf = 1 - train_accuracy_rf
print("Training Error (Random Forest, m=13):", train_error_rf)

# 4. Try different values of m and select the best random forest model
m_values = [8, 12, 14, 16, 18]
best_m = None
best_rf_model = None
lowest_train_error = float('inf')
```

```
for m in m_values:
    rf_clf = RandomForestClassifier(n_estimators=25, max_features=m, rand
    rf_clf.fit(X_train, y_train)

    # Calculate training error
    y_train_pred_rf = rf_clf.predict(X_train)
    train_accuracy_rf = accuracy_score(y_train, y_train_pred_rf)
    train_error_rf = 1 - train_accuracy_rf
    print(f"Training Error (Random Forest, m={m}):", train_error_rf)

    # Select the value of m with the lowest training error
    if train_error_rf < lowest_train_error:
        lowest_train_error = train_error_rf
        best_m = m
        best_rf_model = rf_clf

# Evaluate the best model on the test set
y_test_pred_rf = best_rf_model.predict(X_test)
test_accuracy_rf = accuracy_score(y_test, y_test_pred_rf)
test_error_rf = 1 - test_accuracy_rf
print(f"Test Error (Random Forest, best m={best_m}):", test_error_rf)

# 5. Compare the results of the above methods
print("\nComparison of Methods:")
print("Decision Tree Test Error:", test_error)
print("Pruned Decision Tree Test Error:", pruned_test_error)
print(f"Random Forest Test Error (best m={best_m}):", test_error_rf)
```

```
Non-numeric columns in training data: Index(['LOCATION_ID'], dtype='objec
t')
```



```
Training Error (Decision Tree): 0.0
Confusion Matrix (Test):
 [[99 13]
 [10 78]]
Test Error (Decision Tree): 0.11499999999999999
```




```
Test Error (Pruned Tree): 0.04500000000000004
Training Error (Random Forest, m=13): 0.00347222222222221
Training Error (Random Forest, m=8): 0.00520833333333337
Training Error (Random Forest, m=12): 0.00347222222222221
Training Error (Random Forest, m=14): 0.0017361111111111605
Training Error (Random Forest, m=16): 0.0017361111111111605
Training Error (Random Forest, m=18): 0.00347222222222221
Test Error (Random Forest, best m=14): 0.050000000000000044

Comparison of Methods:
Decision Tree Test Error: 0.11499999999999999
Pruned Decision Tree Test Error: 0.04500000000000004
Random Forest Test Error (best m=14): 0.050000000000000044
```

In [14]:
```python
import pandas as pd
import numpy as np
from sklearn.metrics import mean_absolute_error

class CustomDecisionTreeRegressor:
    def __init__(self, max_leaf_nodes=None, v_gain=0):
        self.max_leaf_nodes = max_leaf_nodes
        self.v_gain = v_gain
        self.tree = None
        self.leaf_count = 0  # To count the number of leaf nodes

    def fit(self, X, y):
        self.leaf_count = 0  # Reset leaf count for each fit
        self.tree = self._fit_tree(X, y)

    def _fit_tree(self, X, y):
        # Base case: if we have reached the maximum number of leaf nodes
        if len(X) == 0 or (self.max_leaf_nodes is not None and self.leaf_
            self.leaf_count += 1
            return np.mean(y)
```

```
        # Find the best split based on absolute loss reduction
        best_feature, best_threshold, min_loss = None, None, float('inf')
        for feature in range(X.shape[1]):
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_mask = X[:, feature] <= threshold
                right_mask = X[:, feature] > threshold
                left_y, right_y = y[left_mask], y[right_mask]

                # Calculate absolute loss for left and right splits
                left_loss = np.sum(np.abs(left_y - np.mean(left_y))) if l
                right_loss = np.sum(np.abs(right_y - np.mean(right_y))) i
                total_loss = left_loss + right_loss

                # Update best split if we have a lower total loss
                if total_loss < min_loss:
                    min_loss = total_loss
                    best_feature, best_threshold = feature, threshold

        # Check stopping criteria
        if min_loss >= self.v_gain:
            self.leaf_count += 1
            return np.mean(y)

        # Split the data into left and right branches
        left_mask = X[:, best_feature] <= best_threshold
        right_mask = X[:, best_feature] > best_threshold
        left_branch = self._fit_tree(X[left_mask], y[left_mask])
        right_branch = self._fit_tree(X[right_mask], y[right_mask])

        # Return a node with the best split and branches
        return {"feature": best_feature, "threshold": best_threshold, "le

    def predict(self, X):
        return np.array([self._predict_tree(x, self.tree) for x in X])

    def _predict_tree(self, x, tree):
        if not isinstance(tree, dict):
            return tree
        if x[tree["feature"]] <= tree["threshold"]:
            return self._predict_tree(x, tree["left"])
        else:
            return self._predict_tree(x, tree["right"])

# Load the data
train_data = pd.read_csv("Hitters_train.csv")
test_data = pd.read_csv("Hitters_test.csv")

# Drop rows with missing values in the target variable
train_data = train_data.dropna(subset=['Salary'])
test_data = test_data.dropna(subset=['Salary'])

# Use feature variables and split data
X_train = train_data[['Years', 'Hits', 'RBI', 'Walks', 'PutOuts', 'Runs']
y_train = train_data['Salary'].values
X_test = test_data[['Years', 'Hits', 'RBI', 'Walks', 'PutOuts', 'Runs']].
y_test = test_data['Salary'].values

# Instantiate and train the custom decision tree
```

```
model = CustomDecisionTreeRegressor(max_leaf_nodes=50, v_gain=0.01)
model.fit(X_train, y_train)

# Generate predictions
y_pred = model.predict(X_test)

# Calculate test error
test_error = mean_absolute_error(y_test, y_pred)
print("Test Error (MAE):", test_error)
```

```
Test Error (MAE): 0.7806312148709367
```

In [ ]: