Linear Regression - by Ali Ghasemi

1. Introduction to Linear Regression

Linear regression models the relationship between a dependent variable (target) and one or more independent variables (features) by fitting a linear equation to observed data. The basic formula for simple linear regression is:

$$y = mx + b$$

Where:

- y is the dependent variable (target)
- x is the independent variable (feature)
- m is the slope of the line
- b is the y-intercept

Python Code Example:

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 1) * 10
y = 2 * X + 1 + np.random.randn(100, 1) * 2

# Plot the data
plt.scatter(X, y)
plt.xlabel("X")
plt.ylabel("y")
plt.title("Simple Linear Regression Example")
plt.show()
```

2. Mathematical Foundations

2.1 Multiple Linear Regression Equation

Multiple linear regression extends the concept of simple linear regression to multiple independent variables (features). It models the relationship between a dependent variable y and multiple independent variables $x_1, x_2, \ldots, x_p$ by fitting a linear equation. The equation takes the form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon$$

Where:

- y is the dependent variable (target).

- $x_1, x_2, ..., x_p$ are the independent variables (features).
- $\beta_0, \beta_1, \beta_2, ..., \beta_p$ are the coefficients corresponding to each feature.
- ε represents the error term, capturing the difference between the actual and predicted values.

In this equation, the goal is to find the coefficients $\beta_0, \beta_1, \beta_2, ..., \beta_p$ that minimize the sum of squared differences between the actual and predicted values of y.

Python Code Example:

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 3) * 10  # 3 features
y = 2 * X[:, 0] + 3 * X[:, 1] + 4 * X[:, 2] + 1 + np.random.randn(100) * 2

# Create and train the multiple linear regression model
model = LinearRegression()
model.fit(X, y)

# Print the coefficients (β0, β1, β2, β3)
coefficients = model.coef_
intercept = model.intercept_
print("Coefficients:", coefficients)
print("Intercept:", intercept)
```

In this example, we generate synthetic data with 3 features and the corresponding target values. The `LinearRegression` model is then used to fit the data, and the coefficients are printed, representing the relationship between each feature and the target. The intercept (β_0) indicates the predicted value of y when all features are zero.

Multiple linear regression is a powerful tool for modeling relationships involving multiple variables. It's commonly used in various fields for predicting outcomes based on multiple input factors. By estimating the coefficients, you can gain insights into how each feature impacts the target variable, and make predictions on new data points.

2.2 Gradient Descent for Coefficient Estimation

Gradient descent is an optimization algorithm used to find the optimal coefficients that minimize the cost function in linear regression. The basic idea is to iteratively update the coefficients in the direction of the steepest decrease in the cost function. This process continues until convergence is reached.

Mathematical Background

In linear regression, we aim to minimize the residual sum of squares (RSS) as the cost function. For a multiple linear regression model, the cost function is:

$$J(\beta) = \Sigma(y_i - \beta_0 - \Sigma(\beta_j x_{ij}))^2$$

The gradient descent update rule for coefficient β_j is:

```
βⱼ_new = βⱼ - α * ∂(J(β)) / ∂(βⱼ)
```

Where:

- `α` is the learning rate, controlling the step size in each iteration.
- `J(β)` is the cost function.
- `∂(J(β)) / ∂(βⱼ)` is the partial derivative of the cost function with respect to coefficient `βⱼ`.

Gradient Descent Algorithm

The gradient descent algorithm involves the following steps:

1. Initialize coefficients `β₀, β₁, β₂, ..., βₚ` randomly or with some initial values.
2. Calculate the predicted values `y_pred` using the current coefficients.
3. Calculate the error term `error = y_pred - y`.
4. Calculate the gradient of the cost function with respect to each coefficient.
5. Update each coefficient using the gradient and learning rate: `βⱼ_new = βⱼ - α * gradient`.
6. Repeat steps 2 to 5 until convergence or a predetermined number of iterations.

Python Code Example:

```python
import numpy as np

def gradient_descent(X, y, coefficients, learning_rate, num_iterations):
    m = len(y)
    for _ in range(num_iterations):
        y_pred = np.dot(X, coefficients)
        error = y_pred - y
        gradient = (1/m) * np.dot(X.T, error)
        coefficients -= learning_rate * gradient
    return coefficients

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 3) * 10  # 3 features
y = 2 * X[:, 0] + 3 * X[:, 1] + 4 * X[:, 2] + 1 + np.random.randn(100) * 2

# Initialize coefficients
coefficients = np.random.randn(3)

# Set hyperparameters
learning_rate = 0.01
num_iterations = 1000

# Apply gradient descent
coefficients_final = gradient_descent(X, y, coefficients, learning_rate,
num_iterations)
print("Final Coefficients:", coefficients_final)
```

In this example, we generate synthetic data with 3 features and corresponding target values. The `gradient_descent` function implements the gradient descent algorithm to estimate the coefficients. The algorithm iteratively updates the coefficients based on the calculated

gradients and learning rate. After convergence, the final coefficients are printed, which should ideally be close to the true coefficients used to generate the data.

Gradient descent is a fundamental optimization technique that plays a crucial role in training various machine learning algorithms, including linear regression. It's particularly useful when dealing with large datasets or complex models where analytical solutions are impractical.

3. Polynomial Regression

Polynomial regression is an extension of linear regression that models relationships between variables by fitting a polynomial equation to the observed data points. It's particularly useful when the relationship between the variables is not linear and a linear model won't fit the data well.

Mathematical Background

The polynomial regression equation of degree `d` takes the form:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_n x^d + \varepsilon$$

Where:

- `y` is the dependent variable (target).
- `x` is the independent variable (feature).
- $\beta_0, \beta_1, \beta_2, \ldots, \beta_n$ are the coefficients corresponding to the polynomial terms.
- ε represents the error term.

By increasing the degree of the polynomial, the model can capture more complex relationships between the variables. However, using a very high degree can lead to overfitting.

Implementing Polynomial Regression

Python Code Example:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 1) * 10
y = 2 * X + 1 + np.random.randn(100, 1) * 2

# Transform features to polynomial features
degree = 2
poly = PolynomialFeatures(degree=degree)
X_poly = poly.fit_transform(X)
```

```
# Create and train the polynomial regression model
model = LinearRegression()
model.fit(X_poly, y)

# Generate points for the polynomial curve
X_range = np.linspace(min(X), max(X), 100).reshape(-1, 1)
X_range_poly = poly.transform(X_range)
y_pred = model.predict(X_range_poly)

# Plot the data and polynomial curve
plt.scatter(X, y)
plt.plot(X_range, y_pred, color='r', label=f'Degree {degree} Polynomial')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Polynomial Regression Example")
plt.legend()
plt.show()
```

In this example, we generate synthetic data with one feature and corresponding target values. We then transform the feature into polynomial features using the `PolynomialFeatures` class. The transformed features are used to fit a linear regression model, effectively fitting a polynomial curve to the data. The plotted graph shows the original data points and the polynomial curve.

Polynomial regression allows for flexible modeling of non-linear relationships in the data. While it can provide a good fit to complex data patterns, it's important to balance the degree of the polynomial to avoid overfitting.

4. Regularization Techniques

Regularization techniques are used in linear regression to prevent overfitting by adding a penalty term to the cost function. The penalty discourages large coefficient values, leading to simpler and more generalizable models.

1. Ridge Regression

Ridge regression (L2 regularization) adds the sum of squared coefficients as a penalty term to the cost function. The cost function becomes:

$$J(\beta) = \Sigma(y_i - \beta_0 - \Sigma(\beta_j x_{ij}))^2 + \alpha\Sigma\beta_j^2$$

Where α is the regularization parameter. Ridge regression encourages small coefficient values without necessarily setting them to zero.

Python Code Example:

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

# Generate example data
np.random.seed(0)
```

```
X = np.random.rand(100, 3) * 10
y = 2 * X[:, 0] + 3 * X[:, 1] + 4 * X[:, 2] + 1 + np.random.randn(100) * 2

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

# Ridge regression
alpha = 0.1
ridge_reg = Ridge(alpha=alpha)
ridge_reg.fit(X_train, y_train)
y_pred = ridge_reg.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error (Ridge):", mse)
```

2. Lasso Regression

Lasso regression (L1 regularization) adds the sum of absolute values of coefficients as a penalty term:

$$J(\beta) = \Sigma(y_i - \beta_0 - \Sigma(\beta_j x_{ij}))^2 + \alpha\Sigma|\beta_j|$$

Lasso has a feature selection property where it can force some coefficients to become exactly zero, effectively selecting a subset of features.

Python Code Example:

```
from sklearn.linear_model import Lasso

# Lasso regression
alpha = 0.1
lasso_reg = Lasso(alpha=alpha)
lasso_reg.fit(X_train, y_train)
y_pred = lasso_reg.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error (Lasso):", mse)
```

3. Elastic Net Regression

Elastic Net combines both L1 and L2 regularization, providing a balance between ridge and lasso. The cost function is a combination of both penalty terms:

$$J(\beta) = \Sigma(y_i - \beta_0 - \Sigma(\beta_j x_{ij}))^2 + \alpha_1\Sigma\beta_j^2 + \alpha_2\Sigma|\beta_j|$$

Elastic Net can handle situations where there are correlated features and benefits from both L1's feature selection and L2's coefficient shrinking.

Python Code Example:

```
from sklearn.linear_model import ElasticNet

# Elastic Net regression
alpha = 0.1
l1_ratio = 0.5  # Mix of L1 and L2 regularization
elastic_net_reg = ElasticNet(alpha=alpha, l1_ratio=l1_ratio)
elastic_net_reg.fit(X_train, y_train)
y_pred = elastic_net_reg.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error (Elastic Net):", mse)
```

Regularization techniques like Ridge, Lasso, and Elastic Net are crucial for avoiding overfitting and improving model generalization. The choice of regularization parameter `α` and other hyperparameters can significantly impact model performance. Cross-validation is often used to determine optimal values for these parameters.

5. Feature Engineering for Linear Regression

Feature engineering involves creating new features or modifying existing ones to improve the performance of machine learning models. In the context of linear regression, feature engineering aims to enhance the relationship between the independent variables and the dependent variable, leading to more accurate predictions.

1. Categorical Variables and One-Hot Encoding

When dealing with categorical variables, such as gender or product categories, they need to be converted into numerical format for linear regression. One-hot encoding is a common technique that represents each category as a binary vector.

Python Code Example:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Sample data with a categorical variable
data = {'color': ['red', 'blue', 'green', 'red', 'blue']}
df = pd.DataFrame(data)

# One-hot encode categorical variable
encoder = OneHotEncoder()
encoded = encoder.fit_transform(df[['color']])
encoded_df = pd.DataFrame(encoded.toarray(),
columns=encoder.get_feature_names(['color']))

print("Original Data:")
print(df)
print("\nOne-Hot Encoded Data:")
print(encoded_df)
```

2. Interaction Terms and Feature Transformation

Interaction terms capture relationships between features that might have a combined effect on the target variable. Feature transformation involves applying mathematical functions to features to create new ones that better capture underlying patterns.

Python Code Example:

```
import numpy as np

# Sample data with interaction terms and transformed features
X = np.random.rand(100, 2) * 10
y = 2 * X[:, 0] + 3 * X[:, 1] + 1 + np.random.randn(100) * 2

# Interaction term
interaction_term = X[:, 0] * X[:, 1]

# Feature transformation
sqrt_feature = np.sqrt(X[:, 1])

print("Original Features (X):")
print(X[:5])
print("\nInteraction Term:")
print(interaction_term[:5])
print("\nSquare Root Transformed Feature:")
print(sqrt_feature[:5])
```

Feature engineering helps linear regression capture complex relationships between variables. Creating new features or transforming existing ones can enhance the predictive power of the model and lead to better insights.

Remember that while feature engineering can greatly improve model performance, it's also important to be mindful of overfitting. Adding too many features or performing too complex transformations can lead to models that perform well on the training data but poorly on new, unseen data. Always evaluate the impact of feature engineering through proper validation and testing procedures.

6. Implementing Linear Regression with Statsmodels

Statsmodels is a powerful library that provides comprehensive statistical models, including linear regression. It offers detailed insights into model statistics, hypothesis testing, and p-values, making it an excellent tool for understanding the relationships between variables.

1. Importing and Preparing Data

Before implementing linear regression with Statsmodels, you need to import your data and prepare it. Ensure that your data is organized in a pandas DataFrame with the target variable and independent variables.

Python Code Example:

```
import pandas as pd
import statsmodels.api as sm
```

```
# Sample data
data = {'X1': [1, 2, 3, 4, 5],
        'X2': [2, 3, 4, 5, 6],
        'y': [3, 5, 7, 9, 11]}

df = pd.DataFrame(data)
X = df[['X1', 'X2']]
y = df['y']

# Add constant to the features
X = sm.add_constant(X)
```

2. Fitting the Linear Regression Model

Statsmodels' OLS (Ordinary Least Squares) class is used to fit a linear regression model. This class provides a wide range of statistics and insights into the model.

Python Code Example:

```
model = sm.OLS(y, X).fit()
```

3. Model Summary and Interpretation

The `summary` method of the model provides a detailed summary of the regression results. It includes information such as coefficient values, standard errors, t-values, p-values, R-squared, and more.

Python Code Example:

```
print(model.summary())
```

4. Hypothesis Testing and p-values

Statsmodels' summary includes p-values associated with each coefficient. These p-values help you determine whether the coefficients are statistically significant.

Python Code Example:

```
# Extract p-values from the summary
p_values = model.summary().tables[1]['P>|t|']

print("P-values for Coefficients:")
print(p_values)
```

5. Making Predictions

Once the model is fitted, you can use it to make predictions on new data.

Python Code Example:

```
# New data for prediction
new_data = {'X1': [6, 7],
            'X2': [7, 8]}

new_df = pd.DataFrame(new_data)
```

```
new_X = sm.add_constant(new_df[['X1', 'X2']])

# Make predictions
predictions = model.predict(new_X)
print("Predictions:", predictions)
```

6. Residual Analysis

Statsmodels also allows you to perform various diagnostics, including residual analysis, normality tests, and homoscedasticity tests.

Python Code Example:

```
# Residual analysis
residuals = y - model.predict(X)
```

Implementing linear regression with Statsmodels provides you with in-depth insights into the statistical aspects of the model. The detailed summary, hypothesis testing, and p-values empower you to understand the significance of the coefficients and evaluate the overall performance of your model.

7. Assumption Checking and Diagnostics

Assumption checking and diagnostics are crucial steps in ensuring the reliability of a linear regression model. These steps involve evaluating whether the underlying assumptions of linear regression hold true for the given data. Common assumptions include linearity, homoscedasticity, normality of residuals, and absence of multicollinearity.

1. Residual Analysis and Normality

One important assumption is that the residuals (differences between observed and predicted values) should be normally distributed. You can visualize this using residual plots and Q-Q plots.

Python Code Example:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 1) * 10
y = 2 * X + 1 + np.random.randn(100, 1) * 2

# Fit linear regression model
model = sm.OLS(y, X).fit()

# Calculate residuals
residuals = model.resid

# Residual plot
plt.scatter(model.fittedvalues, residuals)
plt.xlabel("Fitted Values")
```

```
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.axhline(y=0, color='r', linestyle='--')
plt.show()

# Q-Q plot
stats.probplot(residuals, plot=plt)
plt.title("Q-Q Plot")
plt.show()
```

2. Homoscedasticity and Heteroscedasticity Tests

Homoscedasticity assumes that the variability of residuals is consistent across all levels of the predictor variables. Heteroscedasticity, on the other hand, occurs when the variability changes. You can use tests like the Breusch-Pagan test to detect heteroscedasticity.

Python Code Example:

```
from statsmodels.stats.diagnostic import het_breuschpagan

_, p_value, _, _ = het_breuschpagan(residuals, X)
if p_value < 0.05:
    print("Heteroscedasticity detected.")
else:
    print("Homoscedasticity detected.")
```

3. Cook's Distance for Outlier Detection

Cook's distance measures the influence of each data point on the model. Large values indicate influential points that might affect the model's fit.

Python Code Example:

```
from statsmodels.stats.outliers_influence import OLSInfluence

influence = OLSInfluence(model)
cooks_distance = influence.cooks_distance[0]
outliers = np.where(cooks_distance > threshold)[0]
```

Assumption checking and diagnostics are essential to ensure that the linear regression model is appropriate for the data and provides reliable insights. By evaluating assumptions and identifying potential issues, you can make informed decisions about model validity and take appropriate actions to address any concerns.

8. Advanced Applications and Extensions

8.1 Time Series Forecasting with Linear Regression

Time series forecasting involves predicting future values based on historical data that is ordered by time. Linear regression can be used for time series forecasting by incorporating lagged values of the target variable as features.

1. Creating Lagged Features

One common approach is to create lagged features, which are past values of the target variable as predictors. These lagged features capture the temporal patterns that can help predict future values.

Python Code Example:

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Generate example time series data
np.random.seed(0)
time_steps = 100
time_series = np.cumsum(np.random.randn(time_steps))

# Create lagged features
lags = 3
X = np.zeros((time_steps - lags, lags))
y = np.zeros(time_steps - lags)
for i in range(lags, time_steps):
    X[i - lags] = time_series[i - lags:i]
    y[i - lags] = time_series[i]

# Split into train and test sets
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Fit linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

2. Forecasting Future Values

After fitting the model with lagged features, you can use it to forecast future values by providing lagged values of the target variable for which you want to make predictions.

Python Code Example:

```
# Use the last `lags` values of the time series as input
input_data = time_series[-lags:]

# Predict the next value
forecast = model.predict([input_data])
print("Forecasted Value:", forecast)
```

Time series forecasting with linear regression provides a simple yet effective way to capture temporal patterns in the data. However, it's important to note that more complex time

series forecasting methods, such as autoregressive integrated moving average (ARIMA) or seasonal decomposition of time series (STL), are often used for more accurate predictions in real-world scenarios. Linear regression can serve as a baseline method or a starting point for exploring time series forecasting techniques.

8.2 Bayesian Linear Regression

Bayesian linear regression is a probabilistic approach to linear regression that provides not only point estimates of coefficients but also their full probability distributions. This allows us to quantify the uncertainty associated with each coefficient, making it a powerful technique for both prediction and uncertainty estimation.

Mathematical Background

In Bayesian linear regression, we treat the coefficients as random variables following certain prior distributions. The goal is to find the posterior distribution of the coefficients given the observed data and the prior distributions. The posterior distribution gives us the updated beliefs about the coefficients after observing the data.

Implementing Bayesian Linear Regression

For Bayesian linear regression, we can use libraries like `pymc3` to easily implement the model.

Python Code Example:

```
import numpy as np
import pymc3 as pm
import matplotlib.pyplot as plt

# Generate example data
np.random.seed(0)
X = np.random.rand(100, 1) * 10
y = 2 * X + 1 + np.random.randn(100, 1) * 2

# Define the Bayesian linear regression model
with pm.Model() as model:
    # Prior distributions for coefficients
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=X.shape[1])
    sigma = pm.HalfNormal('sigma', sd=1)

    # Linear regression equation
    mu = alpha + pm.math.dot(X, beta)

    # Likelihood
    y_pred = pm.Normal('y_pred', mu=mu, sd=sigma, observed=y)

    # Perform sampling
    trace = pm.sample(draws=1000, tune=1000)

# Plot posterior distributions
```

```
pm.traceplot(trace)
plt.show()
```

In this example, we use `pymc3` to define the Bayesian linear regression model. We specify the prior distributions for the coefficients and the likelihood function. The `sample` function performs MCMC sampling to estimate the posterior distributions of the coefficients. The resulting trace plot visualizes the posterior distributions.

Making Predictions

To make predictions using Bayesian linear regression, we can extract samples from the posterior distribution of the coefficients and calculate predictions for new data points.

Python Code Example:

```
# New data for prediction
new_X = np.array([[7]])

# Extract samples from the posterior
alpha_samples = trace['alpha']
beta_samples = trace['beta']

# Calculate predictions for each sample
predictions = alpha_samples + np.dot(new_X, beta_samples.T)

# Plot the predicted values
plt.hist(predictions, bins=20, density=True, alpha=0.5)
plt.xlabel("Predicted Values")
plt.ylabel("Density")
plt.title("Predicted Values Distribution")
plt.show()
```

Bayesian linear regression provides a comprehensive view of uncertainty in linear regression modeling. By estimating the posterior distributions of coefficients, you can make predictions while also considering the inherent uncertainty in the model. This makes Bayesian linear regression a valuable tool in scenarios where understanding and quantifying uncertainty are critical.

8.3 Robust Linear Regression

Robust linear regression is a variation of linear regression that aims to provide more reliable estimates of coefficients when the assumptions of linear regression, such as normality and homoscedasticity, are violated. It is designed to handle outliers and data points that do not conform to the typical linear relationship.

Robust Regression Techniques

There are several techniques for performing robust linear regression. One common method is the Huber loss, which combines the benefits of least squares and absolute loss functions. Another technique is the RANSAC algorithm, which iteratively fits the model while excluding outliers.

Implementing Robust Linear Regression

For robust linear regression, we can use the `sklearn` library, which provides the `HuberRegressor` class for Huber loss and the `RANSACRegressor` class for the RANSAC algorithm.

Python Code Example (Huber Regression):

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import HuberRegressor
from sklearn.datasets import make_regression

# Generate example data with outliers
X, y, coef = make_regression(n_samples=100, n_features=1, noise=4,
coef=True, random_state=0)
X_outliers = np.array([[-2], [2]])
y_outliers = np.array([-200, 250])
X = np.vstack((X, X_outliers))
y = np.concatenate((y, y_outliers))

# Fit robust linear regression model
model = HuberRegressor()
model.fit(X, y)

# Predictions for plotting
X_test = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
y_pred = model.predict(X_test)

# Plot the data and robust linear regression line
plt.scatter(X, y, color='b', label='Data')
plt.plot(X_test, y_pred, color='r', label='Robust Linear Regression')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Robust Linear Regression Example")
plt.legend()
plt.show()
```

Python Code Example (RANSAC Regression):

```
from sklearn.linear_model import RANSACRegressor

# Fit RANSAC linear regression model
ransac = RANSACRegressor()
ransac.fit(X, y)

# Inliers mask
inliers_mask = ransac.inlier_mask_

# Predictions for plotting
y_pred_ransac = ransac.predict(X_test)

# Plot the data and RANSAC linear regression line
plt.scatter(X[inliers_mask], y[inliers_mask], color='b', label='Inliers')
plt.scatter(X[~inliers_mask], y[~inliers_mask], color='r',
label='Outliers')
plt.plot(X_test, y_pred_ransac, color='g', label='RANSAC Linear
Regression')
plt.xlabel("X")
```

```
plt.ylabel("y")
plt.title("RANSAC Linear Regression Example")
plt.legend()
plt.show()
```

Robust linear regression provides more accurate estimates of coefficients when dealing with outliers and data that doesn't conform to linear assumptions. It's a valuable technique for ensuring the reliability of linear models in real-world scenarios.

9. Conclusion

Mastering advanced aspects of linear regression empowers you to handle diverse modeling challenges. This comprehensive guide, complete with detailed explanations and extensive Python code examples, equips you with the skills to effectively implement linear regression in various scenarios. As you continue your data science journey, you'll find that a deep understanding of linear regression paves the way for tackling more complex modeling techniques and real-world problems.