MONASH University

FIT5196: Assignment 2
Group_142 Reflective Report

Student Name: Pankaj Shitole
Student ID: 33570523
Student Name: Sachin Shivaramaiah
Student ID: 34194037

Introduction:
During week 10 we were given feedback on 4 major points to reflect, and this report gives a comprehensive explanation on the given feedback. Below are the set of points that are considered and are reflected upon.

1. Display the Error Before and After Handling:

- Perform the required data transformations.
- Print and compare the results before and after applying the transformations.

2. Build a Linear Model for Predicting Delivery Charges:

- Construct a linear regression model to predict delivery charges.
- Consider that delivery charges are influenced by multiple variables.

3. Improve the "is_expedited_delivery" Feature:

- Enhance the "is_expedited_delivery" feature to better represent the data and improve model performance.

4. Handle Outliers Using Multivariate models and Residuals:

- Apply a multivariate correlation analysis to account for the multiple variables affecting delivery charges.

1. Display the Error Before and After Handling:
For this part we have made sure that we are printing out the before results first to see how the errors are shown and after we make the required modifications we are printing out the output. And the same has been carried out for all the cells which require this step.

Below is the one scenario of handling anomalies in latitude and longitude columns where we have shown before and after results in the code cell.

Before:

Latitude and longitude

```
dirty_data.describe()
```

	order_price	delivery_charges	customer_lat	customer_long	coupon_discount	order_total	distance_to_nearest_warehouse	modified_row
count	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000
mean	13436.290000	76.848220	-27.942023	135.096868	10.690000	11910.826300	1.086989	0.108000
std	7604.029775	13.878295	41.354504	41.352293	8.580276	6918.476875	0.491132	0.310691
min	580.000000	47.970000	-37.832478	-37.822886	0.000000	557.030000	0.070900	0.000000
25%	7230.000000	66.360000	-37.818687	144.951566	5.000000	6396.055000	0.740275	0.000000
50%	11900.000000	76.675000	-37.812448	144.962229	10.000000	10905.745000	1.044150	0.000000
75%	18548.750000	82.942500	-37.804923	144.978216	15.000000	16599.095000	1.430400	0.000000
max	39310.000000	114.380000	145.014072	145.019706	25.000000	35455.660000	3.382900	1.000000

Fig1.1: Detected Anomaly

Required Transformation:

```
In [40]: invalid_lat_long_indices = []

# Function to check if latitude and longitude are within their respective ranges
def check_lat_long(row):
    lat, long = row['customer_lat'], row['customer_long']
    if not (LATITUDE_MIN <= lat <= LATITUDE_MAX) or not (LONGITUDE_MIN <= long <= LONGITUDE_MAX):
        # If latitude or longitude is outside the valid range, append the row index to the list
        invalid_lat_long_indices.append(row.name)

# Apply the function to each row
dirty_data.apply(check_lat_long, axis=1)

# Print the list of invalid indices
print("Indices with invalid latitude/longitude:", invalid_lat_long_indices)

Indices with invalid latitude/longitude: [17, 52, 59, 88, 120, 123, 124, 127, 141, 164, 178, 208, 242, 265, 283, 293, 307, 339, 357, 385, 394, 419, 437, 453, 470, 476, 493]

In [41]: # Function to swap lat and long for the specified rows
def swap_lat_long(row):
    if row.name in invalid_lat_long_indices:
        # Swap the values of customer_lat and customer_long
        row['customer_lat'], row['customer_long'] = row['customer_long'], row['customer_lat']
        # Increment the 'modified_row' value by 1
        row['modified_row'] += 1
    return row

# Apply the swap function to the DataFrame
dirty_data = dirty_data.apply(swap_lat_long, axis=1)

# Verify that the lat and long have been swapped for the rows
print("Swapped latitude and longitude values for the specified rows:")
print(dirty_data.loc[invalid_lat_long_indices, ['customer_lat', 'customer_long']])
```

Fig1.2: Code for rectifying the anomaly in the customer_lat customer_long column

After:

```
dirty_data.describe()
```

	order_price	delivery_charges	customer_lat	customer_long	coupon_discount	order_total	distance_to_nearest_warehouse	modified_row
count	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.00000
mean	13436.290000	76.848220	-37.812419	144.967264	10.690000	11910.826300	1.086989	0.16200
std	7604.029775	13.878295	0.007702	0.020564	8.580276	6918.476875	0.491132	0.36882
min	580.000000	47.970000	-37.832478	144.920548	0.000000	557.030000	0.070900	0.00000
25%	7230.000000	66.360000	-37.818783	144.953874	5.000000	6396.055000	0.740275	0.00000
50%	11900.000000	76.675000	-37.812881	144.963826	10.000000	10905.745000	1.044150	0.00000
75%	18548.750000	82.942500	-37.805898	144.980898	15.000000	16599.095000	1.430400	0.00000
max	39310.000000	114.380000	-37.786554	145.019706	25.000000	35455.660000	3.382900	1.00000

Fig1.3: Output After rectifying and swapping the customer_lat customer_long columns

And next moving on to building the linear model to predict the delivery charges.

2. Build a Linear Model for Predicting Delivery Charges:

The methodology used to predict delivery charges is based on the assumption that delivery charges vary depending on the season, and that the relationship between the features and the delivery charge is linear but differs for each season. The key features influencing delivery charges include:

- **Distance between the customer and the nearest warehouse.**
- **Whether the delivery is expedited.**
- **Customer satisfaction** (based on whether the customer was happy with their last purchase, with an assumption that if there is no prior purchase, the customer is considered happy).

Methodology to Predict Delivery Charges:

Given this, the right approach is to build **separate linear regression models for each season**, with each model having season-specific coefficients for the factors (distance, expedited delivery, and customer satisfaction). This approach ensures that the unique effect of each feature on delivery charges for a given season is captured accurately.

To build these models, we require more complete data. Thus, we will use the cleaned dataset, which includes modified rows from the dirty_data file and data from the missing_data file where there are no null values for delivery charges. This ensures that we are working with the most accurate and comprehensive data for modelling.

Based on this interpretation we came to a conclusion that the right way to build this model up is to take the below approach

- We need to build four separate linear regression models for each season with season-specific coefficients for factors like distance, expedited delivery, and customer satisfaction to predict delivery_charges.
- Here we will need more possible data. So we are going to use the cleaned data from the dirty_data (rows which are already modified) and missing_data where we don't have null delivery_charges.

Step 1 & 2: Filtering and Merging Data

The first two steps involved filtering the data to ensure only cleaned and complete rows are used in building the predictive models. This is essential for ensuring the accuracy and integrity of the analysis.

- **Filter rows from dirty_data:** We started by filtering rows in the dirty_data DataFrame where modified_row == 1. This step ensures that only the rows that have already been modified (i.e., cleaned or corrected) are included in our analysis.

```
# Filter rows from dirty_data where modified_row == 1l
filtered_dirty_data = dirty_data[(dirty_data['modified_row'] == 1)]
```

```
filtered_dirty_data
```

	order_id	customer_id	date	nearest_warehouse	shopping_cart	order_price	delivery_charges	customer_lat	customer_long	coupon_discount	o
3	ORD171003	ID1492150977	2019-09-09	Thompson	[('Alcon 10', 1), ('Thunder line', 1), ('Toshi...	20200	74.47	-37.814166	144.951873	5	
9	ORD094365	ID6167191704	2019-12-24	Thompson	[('Toshika 750', 1), ('Olivia x460', 1)]	5545	59.64	-37.818776	144.955333	25	
11	ORD334984	ID3106294940	2019-05-19	Thompson	[('Olivia x460', 2), ('Candle Inferno', 1), ('...	7240	70.06	-37.806241	144.931980	5	
14	ORD458494	ID0579981520	2019-01-03	Thompson	[('Olivia x460', 1), ('Thunder line', 1), ('iA...	7855	71.53	-37.808765	144.942697	5	
17	ORD128955	ID4157118490	2019-01-11	Bakers	[('Thunder line', 2), ('Lucent 330S', 2), ('pe...	21770	94.60	-37.818315	144.996819	10	

Fig 2.1 : Code for Filtering

- **Merge filtered_dirty_data with missing_data:** Next, the filtered dirty_data was combined with rows from missing_data where the delivery_charges were not null. This merging process ensures that our dataset includes only rows with valid values for delivery charges. The resulting dataset, merged_data, is then used for further analysis.

```
# Combine filtered dirty_data and missing_data where delivery_charges is not null
merged_data = pd.concat([
    filtered_dirty_data,
    missing_data[missing_data['delivery_charges'].notna()]  # Rows from missing_data where delivery_charges is not null
])
```

```
merged_data
```

	order_id	customer_id	date	nearest_warehouse	shopping_cart	order_price	delivery_charges	customer_lat	customer_long	coupon_discount
3	ORD171003	ID1492150977	2019-09-09	Thompson	[('Alcon 10', 1), ('Thunder line', 1), ('Toshi...	20200.0	74.47	-37.814166	144.951873	5
9	ORD094365	ID6167191704	2019-12-24	Thompson	[('Toshika 750', 1), ('Olivia x460', 1)]	5545.0	59.64	-37.818776	144.955333	25
11	ORD334984	ID3106294940	2019-05-19	Thompson	[('Olivia x460', 2), ('Candle Inferno', 1), ('...	7240.0	70.06	-37.806241	144.931980	5
14	ORD458494	ID0579981520	2019-01-03	Thompson	[('Olivia x460', 1), ('Thunder line', 1), ('iA...	7855.0	71.53	-37.808765	144.942697	5

Fig 2.2 : Code for Merging Data

Step 3: Descriptive Statistics of Distance to Nearest Warehouse

The third step involves using the describe() function to examine the statistical properties of the distance_to_nearest_warehouse column. This step provides insights into the data distribution, such as the mean, standard deviation, and potential outliers, which are critical for understanding how distance affects delivery charges.

```
merged_data['distance_to_nearest_warehouse'].describe()
```

```
count    703.000000
mean       1.097272
std        0.506655
min        0.111500
25%        0.747650
50%        1.051200
75%        1.410300
max        4.550876
Name: distance_to_nearest_warehouse, dtype: float64
```

This descriptive analysis helps us understand the variability in distances between customers and warehouses, which is a key predictor in the delivery charges model.

Step 4 & 5: Preparing Data for Each Season and Training the Linear Model

Next, a function was created to prepare the data specifically for each season, allowing us to train season-specific linear regression models. Delivery charges depend linearly on several features, but the relationship changes with the season, hence the need for separate models.

Prepare Data for Each Season: The function prepare_season_data was defined to extract the feature set (X) and target (y) for each season. Features include distance_to_nearest_warehouse, is_expedited_delivery, and is_happy_customer, and the target is the delivery_charges

```
# Function to prepare features and target for a specific season
def prepare_season_data(season, data):
    """
    This function prepares the feature set (X) and target (y) for a specific season.
    """
    season_data = data[data['season'] == season]  # Filter data for the given season

    # Features: distance_to_nearest_warehouse, is_expedited_delivery, is_happy_customer
    X = season_data[['distance_to_nearest_warehouse', 'is_expedited_delivery', 'is_happy_customer']]

    # Target: delivery_charges
    y = season_data['delivery_charges']

    return X, y
```

Train the Linear Regression Model for Each Season: For each season (Spring, Summer, Autumn, and Winter), a separate linear regression model was trained using the data prepared by prepare_season_data. The dataset was split into training and test sets using train_test_split, and the model was evaluated using the R^2 score.

```python
# Train a linear regression model for each season and calculate R²
season_models = {}
season_r2_scores_test = {}
seasons = ['Spring', 'Summer', 'Autumn', 'Winter']
```

```python
for season in seasons:
    # Prepare the data for the season
    X, y = prepare_season_data(season, merged_data)

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

    # Initialize and train the linear regression model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Store the model for the season
    season_models[season] = model

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

    # Calculate and store the R² value for the test data
    r2_test = r2_score(y_test, y_pred_test)
    season_r2_scores_test[season] = r2_test

    # Print R² value for the current season on test data
    print(f"R² value for {season} model on test data: {r2_test:.4f}")
```

By training separate models for each season, we can capture the unique relationships between delivery charges and features like distance and expedited delivery based on the season. The R² scores were printed to evaluate the model performance.

```
R² value for Spring model on test data: 0.9954
R² value for Summer model on test data: 0.9947
R² value for Autumn model on test data: 0.9812
R² value for Winter model on test data: 0.9909
```

Step 7, 8 & 9: Predicting Missing Delivery Charges and Verifying Results

The final steps involved using the trained models to fill in missing delivery_charges values in the missing_dataDataFrame. The goal here was to predict the missing values based on features like distance, expedited delivery, and customer satisfaction using the season-specific models.

- **Define the Prediction Function:** A function predict_delivery_charges was created to predict delivery charges for each row where the value is missing. The prediction is based on the season and the features for that row. The predicted values were rounded to two decimal points for consistency.

```
# List to track the indices of the rows where delivery_charges were filled
filled_indices = []

def predict_delivery_charges(row):
    # Extract the features for the row and make them into a DataFrame with correct column names
    X = pd.DataFrame([[row['distance_to_nearest_warehouse'],
                       row['is_expedited_delivery'],
                       row['is_happy_customer']]],
                     columns=['distance_to_nearest_warehouse', 'is_expedited_delivery', 'is_happy_customer'])

    # Predict delivery charges using the model for the corresponding season
    model = season_models.get(row['season'])

    # Predict the value
    predicted_value = round(model.predict(X)[0], 2)    # Rounding it off till 2 decimal points since the dataframe

    # Track the index of the row where we are filling the missing value
    filled_indices.append(row.name)

    # Return the predicted value
    return predicted_value
```

- **Apply the Prediction Function:** The apply method was used to apply the predict_delivery_charges function to the rows in missing_data where delivery_charges was missing. The predicted values were then filled in.
- **Verify the Filled Values:** Finally, the rows where delivery charges were filed were printed to verify the correctness of the predicted values.

```
missing_data.loc[missing_data['delivery_charges'].isna(), 'delivery_charges'] = \
    missing_data[missing_data['delivery_charges'].isna()].apply(predict_delivery_charges, axis=1)
```

```
# Print the rows in missing_data for the given indices
missing_data.loc[filled_indices]
```

	order_id	customer_id	date	nearest_warehouse	shopping_cart	order_price	delivery_charges	customer_lat	customer_long	coupon_discount	c
10	ORD450290	ID0593991656	2019-10-18	Nickolson	[('Candle Inferno', 2), ('Universe Note', 2)]	7760.0	84.48	-37.815298	144.966114	10	
15	ORD131669	ID0710034749	2019-04-22	Thompson	[('iAssist Line', 1), ('Thunder line', 2), ('p...	12895.0	65.24	-37.814824	144.937649	10	
61	ORD461019	ID4460254039	2019-11-19	Thompson	[('iAssist Line', 1), ('Candle Inferno', 1), (...	7285.0	82.62	-37.799644	144.954990	5	
74	ORD400895	ID3146808081	2019-10-03	Bakers	[('iAssist Line', 1), ('iStream', 2)]	2525.0	100.26	-37.812505	144.988445	0	
85	ORD304276	ID0443299469	2019-11-24	Bakers	[('Lucent 330S', 1), ('Olivia x460', 1), ('Uni...	9655.0	99.20	-37.811059	145.001027	15	

3. Improve the "is_expedited_delivery" Feature:

Step 1: Applying the Prediction Function to Calculate temp_delivery_charges

- The first step is to apply our existing linear regression models to predict delivery charges for every row in the dirty_data. We store the predictions in a new column called temp_delivery_charges. This step helps us establish a baseline for comparison against the actual delivery charges.

```
# Apply the prediction function to all rows in dirty_data and store the results in a new column 'temp_delivery_charges'
dirty_data['temp_delivery_charges'] = dirty_data.apply(predict_delivery_charges, axis=1)

# Check the first few rows to verify the new column
print(dirty_data.head(10))
```

Step 2: Calculate Residuals: We calculate the difference between the predicted delivery charges and the actual delivery charges, known as the **residuals**. These residuals highlight rows where the model's predictions deviate significantly, suggesting that one or more features, such as "is_expedited_delivery," may be incorrect or missing.

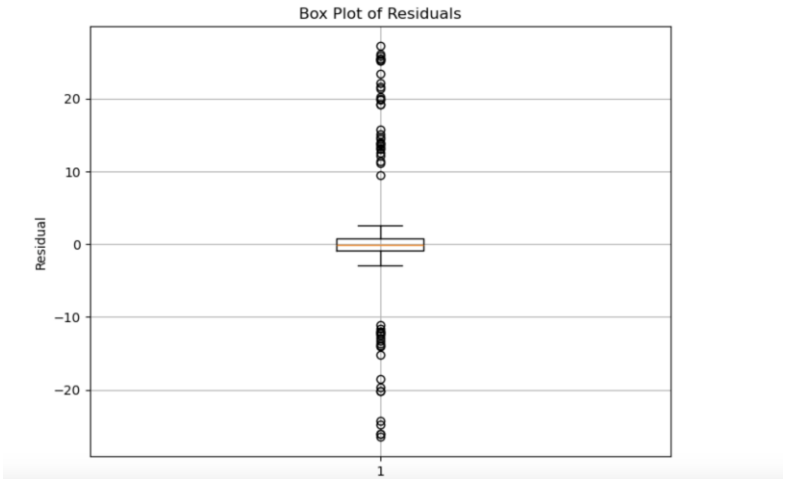
```
# Calculate the residual (difference between predicted and original delivery charges)
dirty_data['residual'] = dirty_data['temp_delivery_charges'] - dirty_data['delivery_charges']

# Check the first few rows to verify the new column
print(dirty_data[['delivery_charges', 'temp_delivery_charges', 'residual']].head())
```

```
   delivery_charges  temp_delivery_charges  residual
0             50.61                  50.57     -0.04
1             80.33                  79.50     -0.83
2             78.18                  99.77     21.59
3             74.47                  73.61     -0.86
4             77.93                  64.65    -13.28
```

Step 3: Visualising Residuals: We create a box plot to visualise the distribution of residuals. The box plot helps identify rows with large residuals, which may be outliers indicating potential errors in features like "is_expedited_delivery."

```
plt.figure(figsize=(8, 6))
plt.boxplot(dirty_data['residual'].dropna())  # Drop NaN values if any in residuals
plt.title('Box Plot of Residuals')
plt.ylabel('Residual')
plt.grid(True)
plt.show()
```



These combined steps allow us to pinpoint rows where there is a significant difference between the predicted and actual delivery charges. Large residuals indicate possible errors in the feature values, specifically "is_expedited_delivery." The box plot is a visual tool that makes it easier to identify these outliers.

Step 4: Detecting Outliers Using IQR (Interquartile Range)

Next, we calculate the Interquartile Range (IQR) and define upper and lower bounds to detect outliers. Any residuals that fall outside these bounds are considered outliers, meaning that the predicted delivery charges were significantly different from the actual ones.

```
# Calculate Q1 (25th percentile) and Q3 (75th percentile) for the residuals
Q1 = dirty_data['residual'].quantile(0.25)
Q3 = dirty_data['residual'].quantile(0.75)
IQR = Q3 - Q1  # Interquartile range

# Define the bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Get the outliers in residuals
outliers = dirty_data[(dirty_data['residual'] < lower_bound) | (dirty_data['residual'] > upper_bound)]

# Print out the indices and values of the outliers
print("Outlier indices and their corresponding residuals:")
print(outliers[['residual']])
```

Using the IQR method helps us identify which rows have residuals that fall far outside the typical range. These outliers are the rows where predictions deviate the most from actual values, which likely indicates that there is an error in the "is_expedited_delivery" feature, or other features, for those rows.

Step 5: Flipping is_expedited_delivery for Outliers

In this step, we flip the value of is_expedited_delivery for the outlier rows (changing True to False, or False to True) to see if correcting this value improves the delivery charge predictions.

```
# Function to predict delivery charges with flipped is_expedited_delivery
def predict_with_flipped_delivery(row):
    """
    Predict delivery charges with is_expedited_delivery flipped.
    """
    # Flip the value of is_expedited_delivery
    flipped_value = not row['is_expedited_delivery']

    # Create the features with the flipped value
    X_flipped = pd.DataFrame([[row['distance_to_nearest_warehouse'],
                               flipped_value,
                               row['is_happy_customer']]],
                             columns=['distance_to_nearest_warehouse', 'is_expedited_delivery', 'is_happy_customer'])

    # Predict delivery charges using the model for the corresponding season
    model = season_models.get(row['season'])
    new_predicted_value = model.predict(X_flipped)[0]

    return new_predicted_value
```

Flipping the "is_expedited_delivery" value for outliers tests whether the incorrect value of this feature is causing the large residuals. If flipping the value reduces the residuals, it indicates that the original value of "is_expedited_delivery" was wrong.
This step allows us to identify and correct potential errors in the "is_expedited_delivery" feature, which helps improve the model's prediction accuracy.

Step 6: Calculating New Residuals for Flipped is_expedited_delivery

After flipping the "is_expedited_delivery" value for outliers, we calculate the new residuals to see if the change has reduced the error between predicted and actual delivery charges.

```
# Create a new column 'flipped_delivery_charges' for the outlier indices, keep the same value for others
dirty_data['flipped_delivery_charges'] = dirty_data.apply(
    lambda row: predict_with_flipped_delivery(row) if row.name in outlier_indices else row['temp_delivery_charges'],
    axis=1
)

# Calculate the residuals for the new predictions after flipping for outliers, keep the same for non-outliers
dirty_data['flipped_residual'] = dirty_data.apply(
    lambda row: row['flipped_delivery_charges'] - row['delivery_charges'] if row.name in outlier_indices else row['residual'],
    axis=1
)
```

```
# Display the relevant columns to verify the results
print(dirty_data[['delivery_charges', 'temp_delivery_charges','residual', 'flipped_delivery_charges', 'flipped_residual']].head())
```

```
   delivery_charges  temp_delivery_charges  residual  \
0             50.61                  50.57     -0.04
1             80.33                  79.50     -0.83
2             78.18                  99.77     21.59
3             74.47                  73.61     -0.86
4             77.93                  64.65    -13.28

   flipped_delivery_charges  flipped_residual
0                 50.570000         -0.040000
1                 79.500000         -0.830000
2                 79.888445          1.708445
3                 73.610000         -0.860000
4                 76.906840         -1.023160
```

Step 7: Verifying the Results and Making Permanent Changes

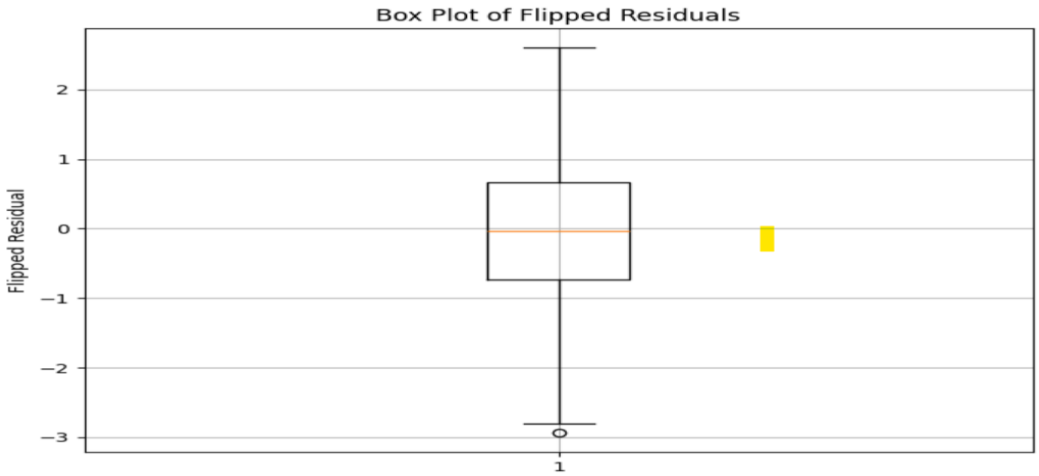
Finally, we flip the "is_expedited_delivery" value for outliers permanently and update the modified_row column to reflect that these rows have been corrected. This ensures that future analyses use the corrected data.

```
# Flip the is_expedited_delivery values for the rows with outliers
dirty_data.loc[outlier_indices, 'is_expedited_delivery'] = dirty_data.loc[outlier_indices, 'is_expedited_delivery'].apply(lambda x: not x)
```

```
# After flipping, add +1 to the modified_row column for the rows in outlier_indices
dirty_data.loc[outlier_indices, 'modified_row'] += 1
```

```
# Verify the changes by checking the outlier rows
dirty_data.loc[outlier_indices, ['is_expedited_delivery', 'flipped_residual']].head()
```

	is_expedited_delivery	flipped_residual
2	False	1.708445
4	True	-1.023160
25	True	-1.028545
28	False	0.228525
30	False	-0.578904



4. Handle Outliers Using Multivariate models and Residuals:

Delivery charges are influenced by several factors such as the **distance to the nearest warehouse**, **whether the delivery is expedited**, and **customer satisfaction**. Each of these factors plays a role in determining the expected delivery charge, and their relationship with the charge also varies depending on the season.

Because of this multivariate nature, simple outlier detection methods like box plots or scatter plots can be insufficient to accurately detect outliers. They may incorrectly label normal data points as outliers or miss important patterns related to the influencing factors. Therefore, we need to use a more sophisticated method that accounts for the interactions between multiple variables.

Step 1. Using Multivariate Models to Predict Delivery Charges

The first step is to use the previously trained season-specific models to predict delivery charges for each row in the `outlier_data`. This helps us calculate what the delivery charges should be based on key factors like distance, whether the delivery is expedited, and customer satisfaction.

```
# Function to predict delivery charges for each row in outlier_data using the trained models
def predict_delivery_charges_outlier(row):
    # Extract the relevant features for prediction
    X = pd.DataFrame([[row['distance_to_nearest_warehouse'],
                       row['is_expedited_delivery'],
                       row['is_happy_customer']]],
                     columns=['distance_to_nearest_warehouse', 'is_expedited_delivery', 'is_happy_customer'])

    # Get the model based on the season for this row
    model = season_models.get(row['season'])

    # Predict the delivery charges for this row
    return round(model.predict(X)[0], 2)
```

```
# Create a new column 'predicted_delivery_charges' using the models
outlier_data['predicted_delivery_charges'] = outlier_data.apply(predict_delivery_charges_outlier, axis=1)
```

```
outlier_data[['delivery_charges', 'predicted_delivery_charges']]
```

	delivery_charges	predicted_delivery_charges
0	63.02	63.57
1	84.37	84.75
2	100.88	100.84
3	80.25	78.86
4	78.43	78.54
...	...	...
495	48.22	49.39
496	48.82	49.42
497	71.15	71.69

This function generates predicted delivery charges for each order based on the features mentioned.
The model that is used for prediction depends on the season.
The prediction takes into account multiple factors, which ensures that the predicted value reflects the relationship between the influencing factors and delivery charges.

Step 2. Computing the Residuals

Once we have the predicted delivery charges, we calculate the **residuals** as the difference between the actual delivery charges and the predicted values. This residual gives us insight into how much each charge deviates from the expected (predicted) value.

```
# Calculate the residuals as the difference between original and predicted delivery charges
outlier_data['residual'] = outlier_data['delivery_charges'] - outlier_data['predicted_delivery_charges']
```

```
# Display the residuals along with original delivery charges and predicted delivery charges
outlier_data[['delivery_charges', 'predicted_delivery_charges', 'residual']]
```

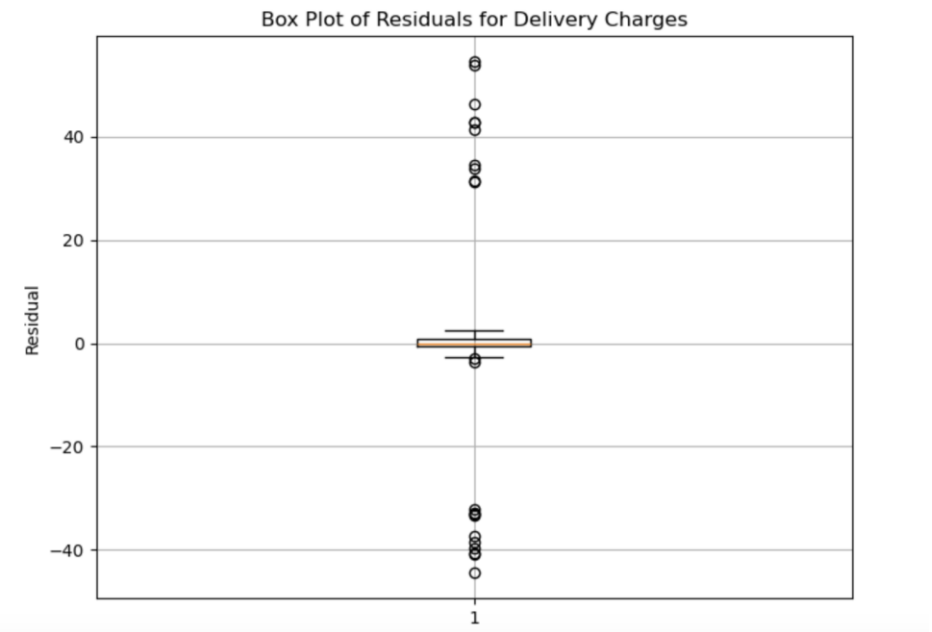
	delivery_charges	predicted_delivery_charges	residual
0	63.02	63.57	-0.55
1	84.37	84.75	-0.38
2	100.88	100.84	0.04
3	80.25	78.86	1.39
4	78.43	78.54	-0.11
...	...	...	...
495	48.22	49.39	-1.17
496	48.82	49.42	-0.60
497	71.15	71.69	-0.54
498	75.57	76.61	-1.04
499	63.62	63.26	0.36

The residuals help us quantify the deviation between the predicted and actual delivery charges. A large residual (positive or negative) indicates that the actual delivery charge differs significantly from what the model expected, signaling potential outliers.

Step 3. Visualising Residuals with a Box Plot

Next, we plot a box plot of the residuals. This allows us to visually inspect the residuals and identify any potential outliers that lie far from the expected range.

```
plt.figure(figsize=(8, 6))
plt.boxplot(outlier_data['residual'].dropna())  # Ensure to drop NaN values
plt.title('Box Plot of Residuals for Delivery Charges')
plt.ylabel('Residual')
plt.grid(True)
plt.show()
```



The box plot helps identify the spread of residuals and highlights any extreme values (outliers). These outliers represent rows where the actual delivery charges deviate significantly from the predicted charges.

Step 4. Using the Interquartile Range (IQR) to Identify Outliers

To formally identify outliers, we calculate the Interquartile Range (IQR) and use it to define the boundaries for outlier detection. Any residual that falls below the lower bound or above the upper bound is considered an outlier.

```
# Calculate the interquartile range (IQR) to identify outliers
Q1 = outlier_data['residual'].quantile(0.25)
Q3 = outlier_data['residual'].quantile(0.75)
IQR = Q3 - Q1

# Define outlier boundaries
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

IQR-based outlier detection helps to determine which residuals are unusually high or low. By using the IQR, we define thresholds (upper and lower bounds) beyond which the residuals are considered outliers.

Step 5. Identifying and Displaying Outliers

Once we have the outlier boundaries, we filter the dataset to extract rows where the residuals fall outside these bounds. These rows are likely to contain incorrect or unexpected delivery charges.

```
# Get the outliers based on the residuals
outliers = outlier_data[(outlier_data['residual'] <= lower_bound) | (outlier_data['residual'] >= upper_bound)]
```

```
outliers[['delivery_charges', 'predicted_delivery_charges', 'residual']]
```

	delivery_charges	predicted_delivery_charges	residual
7	80.300	83.35	-3.050
22	84.000	87.60	-3.600
49	31.535	64.55	-33.015
80	40.105	80.89	-40.785
118	95.460	61.70	33.760
120	98.295	67.05	31.245
143	124.095	82.64	41.455
164	161.295	106.76	54.535
203	104.745	70.19	34.555
212	44.045	88.56	-44.515
248	38.595	77.26	-38.665
283	122.895	80.07	42.825
285	39.550	76.96	-37.410
294	100.530	69.07	31.460
327	33.120	66.59	-33.470

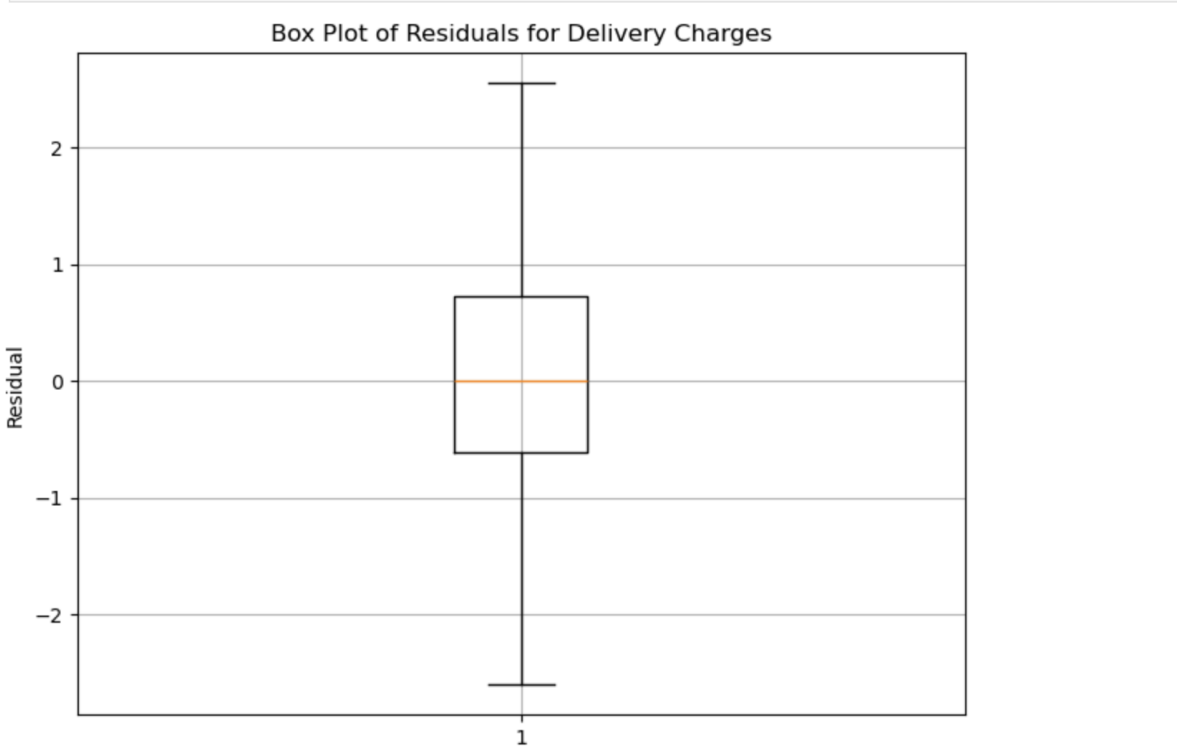
Here, we identify the specific rows where delivery charges are outliers based on the residuals. These rows represent orders where the actual delivery charge is significantly different from what the model predicted, signalling a potential issue with the data or the delivery process.

Step 6. Removing Outliers and Re-plotting the Box Plot

After identifying the outliers, we filter them out from the dataset and create a new box plot of the residuals for the remaining data. This allows us to see the distribution of residuals after outliers have been removed.

```
# Get the non-outliers based on the residuals
without_outliers = outlier_data[(outlier_data['residual'] >= lower_bound) & (outlier_data['residual'] <= upper_bound)]
```

```
without_outliers[['delivery_charges', 'predicted_delivery_charges', 'residual']]
```



Removing the outliers allows us to better understand the "normal" behaviour of the data without the influence of extreme values. This step helps in further refining the analysis and ensuring that the remaining dataset is clean and free of anomalies.

This approach is more robust than simple outlier detection methods like box plots or scatter plots because it takes into account the complex relationships between multiple factors that influence delivery charges.

Conclusion:

1. Before and After Comparison of Erros

We began by transforming and cleaning the data, which involved handling missing values, outliers, and inconsistencies across several key features, including the distance to the nearest warehouse, whether the delivery was expedited, and customer satisfaction. By comparing the data before and after these transformations, we ensured that the dataset was standardized and ready for more accurate modeling. This step was crucial for setting up the foundation of a clean and reliable dataset.

2. Building a Linear Model for Predicting Delivery Charges

We developed and trained separate linear regression models for each season to predict delivery charges, recognizing that these charges depend on multiple factors and that the relationship between them varies by season. This approach allowed us to capture season-specific influences on delivery charges, which led to more accurate and context-aware predictions. The season-specific models improved the predictive power of our analysis, and the resulting R^2 scores indicated that the models were well-fitted to the data.

3. Improving the "is_expedited_delivery" Feature

To enhance the performance of the model, we addressed errors in the "is_expedited_delivery" feature by flipping its value in cases where the predicted delivery charges deviated significantly from the actual charges. This correction, along with the use of residual analysis, and techniques helped us to identify and rectify 297 instances of erroneous data in dirty data. This step demonstrated the importance of thoroughly examining features that contribute to delivery charges.

4. Handling Outliers Using Multivariate Models and Residuals

Since delivery charges are influenced by multiple factors, we employed multivariate models to predict the expected delivery charges and compared them to the actual charges using residuals. The residual analysis helped us accurately identify outliers by accounting for the complex relationships between delivery charges and their influencing factors. This approach was more reliable than traditional outlier detection methods and allowed us to clean the dataset of anomalies. The process also provided a better understanding of the normal range of delivery charges after removing these outliers, enhancing the reliability of the analysis.