A Conditional Logic

When I work with real-world datasets, I often run into challenges like missing values, outliers, and inconsistencies that can mess up my analysis. I know that addressing these issues is crucial because ignoring them can lead to inaccurate results. To handle these problems, I use MATLAB's powerful tools for data cleaning and conditional logic. By doing this, I can make sure my data is in good shape and ready for analysis.

The first thing I do when I start is simulate or load a dataset. I think of this as preparing a playground for testing. Real-world datasets are rarely perfect, and I often find missing values caused by incomplete measurements, errors, or gaps in the data collection process. If I ignore these missing values, they can ruin averages, distort trends, and even break statistical models. I decide whether to remove these entries entirely or replace them with reasonable estimates, like the mean or median. MATLAB makes it easy for me to handle this because I can use functions to spot and fill missing values quickly.

Next, I focus on outliers. These are those extreme values that stick out and can completely throw off my results. I've learned that outliers could be errors, or they could be meaningful, depending on the context of my data. To deal with them, I use the interquartile range (IQR) method. I calculate the 25th and 75th percentiles and identify values that fall way outside this range. MATLAB is really handy here because it lets me compute these metrics and clean up the outliers with just a few lines of code. By doing this, I feel confident that my analysis isn't being skewed by these extreme values.

Once I've cleaned my data, I use conditional logic to validate and transform it. This is where I start making my data smarter. With conditional logic, I can check specific rules, classify data into groups, and even make corrections when needed. MATLAB's `if`, `if-else`, and `if-elseif` statements are tools I rely on heavily for this. I use simple conditions to flag suspicious values or validate entries, while more complex nested conditions let me handle multiple layers of checks. For example, I can classify values into categories like "low," "medium," or "high" and even apply extra rules to those categories. This makes my data much more meaningful and ready for deeper analysis.

What I love about MATLAB is how it lets me combine logical checks with vectorized operations. Instead of going through each data point one by one, I can apply my conditions to entire arrays at once. This is a game-changer for me because it saves time and makes my workflow so much smoother. I feel like I can handle even large datasets efficiently when I use these techniques.

I've realized that cleaning data and applying conditional logic is more than just a technical step—it's about setting the stage for accurate and reliable analysis. When I take the time to clean my data and apply logical rules, I know I'm building a strong foundation for everything that comes after. I've found that these steps not only improve the quality of my data but also give me confidence in the insights I draw from it.

In the end, I've learned that mastering data cleaning and conditional logic in MATLAB is essential for me to handle messy, real-world datasets. By addressing missing values, removing outliers, and applying logical transformations, I can make sure my data is clean, validated, and ready for meaningful analysis. Every time I tackle these challenges, I feel like I'm getting better at transforming messy data into something I can trust and use to uncover valuable insights.

Simulating a Big DatasetExplanation

To work with data that feels real, I decided to include missing values, inconsistent entries, and outliers. This dataset will provide the perfect sandbox for me to test cleaning and logical operations.

```
% I am creating a dataset with mixed data types, missing values, and outliers.
data = {
    'ID', 'Age', 'Income', 'City';  % Header row describing the dataset fields
    1, 25, 50000, 'New York';       % A normal data row
    2, NaN, 62000, 'Los Angeles';   % Missing value in 'Age'
    3, 32, NaN, 'San Francisco';    % Missing value in 'Income'
    4, 45, 1000000, 'New York';     % Outlier in 'Income'
    5, 55, 150000, 'Chicago';       % Normal data row
    };

% I want to see the original dataset to verify what I'm working with.
disp('Original Dataset:');
```

```
Original Dataset:
```

```
disp(data);
```

```
    {'ID'}    {'Age'}    {'Income' }    {'City'         }
    {[ 1]}    {[ 25]}    {[  50000]}    {'New York'     }
    {[ 2]}    {[NaN]}    {[  62000]}    {'Los Angeles'  }
    {[ 3]}    {[ 32]}    {[    NaN]}    {'San Francisco'}
    {[ 4]}    {[ 45]}    {[1000000]}    {'New York'     }
    {[ 5]}    {[ 55]}    {[ 150000]}    {'Chicago'      }
```

Cleaning the DatasetHandling Missing ValuesExplanation

I need to address missing values for `Age` and `Income`. For `Age`, I'll use the average of known values to fill in the gaps. For `Income`, I'll replace missing values with zeros, as it might represent no income or incomplete reporting.

```
% I am initializing a cleaned dataset to keep the original intact.
cleanedData = data;

% I am looping through each row of the dataset to check for missing values.
for i = 2:size(data, 1)  % I start at row 2 since row 1 contains headers.
    % If 'Age' is missing, I replace it with the mean of the available 'Age' values.
    if isnan(data{i, 2})
        % I extract numeric values for 'Age' and calculate the mean, ignoring NaNs.
        ageValues = cell2mat(data(2:end, 2));
        cleanedData{i, 2} = mean(ageValues(~isnan(ageValues)));  % Replace with mean age.
    end

    % If 'Income' is missing, I replace it with 0.
    if isnan(data{i, 3})
        cleanedData{i, 3} = 0;  % I set missing incomes to zero.
    end
end

% I am displaying the dataset after handling missing values.
disp('After Handling Missing Values:');
```

```
After Handling Missing Values:
```

```
disp(cleanedData);
```

```
    {'ID'}    {'Age'    }    {'Income' }    {'City'         }
    {[ 1]}    {[     25]}    {[  50000]}    {'New York'     }
    {[ 2]}    {[39.2500]}    {[  62000]}    {'Los Angeles'  }
    {[ 3]}    {[     32]}    {[      0]}    {'San Francisco'}
    {[ 4]}    {[     45]}    {[1000000]}    {'New York'     }
    {[ 5]}    {[     55]}    {[ 150000]}    {'Chicago'      }
```

Detecting and Removing OutliersExplanation

Outliers, like extremely high incomes, can skew analysis. To address this, I cap all income values above $500,000.

```
% I loop through the rows to identify and handle outliers in the 'Income' column.
for i = 2:size(cleanedData, 1)
    % If the 'Income' is greater than $500,000, I cap it to $500,000.
    if cleanedData{i, 3} > 500000
        cleanedData{i, 3} = 500000;  % I chose this threshold as it aligns with typical income limits.
    end
end

% I am displaying the dataset after removing outliers.
disp('After Removing Outliers:');
```

```
After Removing Outliers:
```

```
disp(cleanedData);
```

```
    {'ID'}    {'Age'    }    {'Income'}    {'City'         }
    {[ 1]}    {[     25]}    {[ 50000]}    {'New York'     }
    {[ 2]}    {[39.2500]}    {[ 62000]}    {'Los Angeles'  }
    {[ 3]}    {[     32]}    {[     0]}    {'San Francisco'}
    {[ 4]}    {[     45]}    {[500000]}    {'New York'     }
    {[ 5]}    {[     55]}    {[150000]}    {'Chicago'      }
```

Applying Conditional LogicSection 3.1: Simple `IF` ConditionsExplanation

I use simple `IF` conditions to validate entries in the dataset, such as ensuring `Age` falls within a realistic range.

```
% I loop through the dataset to validate the 'Age' column.
for i = 2:size(cleanedData, 1)
    % I check if 'Age' is outside a realistic range (0-120).
    if cleanedData{i, 2} < 0 || cleanedData{i, 2} > 120
        cleanedData{i, 2} = NaN;  % I flag invalid ages as NaN.
        disp(['Invalid age detected for ID: ', num2str(cleanedData{i, 1})]);
    end
end
```

`IF-ELSE` ConditionsExplanation

I categorize income levels into "High" or "Low" based on a threshold of $50,000 using `IF-ELSE`.

```
% I loop through the dataset to categorize income levels.
for i = 2:size(cleanedData, 1)
    if cleanedData{i, 3} < 50000
        cleanedData{i, 5} = 'Low Income';  % I label as 'Low Income' if below $50,000.
    else
        cleanedData{i, 5} = 'High Income';  % Otherwise, I label as 'High Income'.
    end
end

% I display the dataset with the new income categories.
disp('After Categorizing Income:');
```

```
After Categorizing Income:
```

```
disp(cleanedData);
```

```
    {'ID'}    {'Age'    }    {'Income'}    {'City'         }    {0×0 double   }
    {[ 1]}    {[     25]}    {[ 50000]}    {'New York'     }    {'High Income'}
    {[ 2]}    {[39.2500]}    {[ 62000]}    {'Los Angeles'  }    {'High Income'}
    {[ 3]}    {[     32]}    {[     0]}    {'San Francisco'}    {'Low Income' }
    {[ 4]}    {[     45]}    {[500000]}    {'New York'     }    {'High Income'}
    {[ 5]}    {[     55]}    {[150000]}    {'Chicago'      }    {'High Income'}
```

`IF-ELSEIF` ConditionsExplanation

I refine income categorization into multiple brackets using `IF-ELSEIF`.

```
% I loop through the dataset for more detailed income categorization.
for i = 2:size(cleanedData, 1)
    if cleanedData{i, 3} < 50000
        cleanedData{i, 6} = 'Low Income';
    elseif cleanedData{i, 3} <= 100000
        cleanedData{i, 6} = 'Middle Income';
    else
        cleanedData{i, 6} = 'High Income';
    end
end

% I display the dataset with detailed income brackets.
disp('After Assigning Income Brackets:');
```

```
After Assigning Income Brackets:
```

```
disp(cleanedData);
```

```
    {'ID'}    {'Age'    }    {'Income'}    {'City'         }    {0×0 double   }    {0×0 double     }
    {[ 1]}    {[     25]}    {[ 50000]}    {'New York'     }    {'High Income'}    {'Middle Income'}
    {[ 2]}    {[39.2500]}    {[ 62000]}    {'Los Angeles'  }    {'High Income'}    {'Middle Income'}
    {[ 3]}    {[     32]}    {[     0]}    {'San Francisco'}    {'Low Income' }    {'Low Income'   }
    {[ 4]}    {[     45]}    {[500000]}    {'New York'     }    {'High Income'}    {'High Income'  }
    {[ 5]}    {[     55]}    {[150000]}    {'Chicago'      }    {'High Income'}    {'High Income'  }
```

Nested ConditionsExplanation

I use nested conditions to create hierarchical categories combining city and income level.

```
% I loop through the dataset to assign combined city and income labels.
for i = 2:size(cleanedData, 1)
    if strcmp(cleanedData{i, 4}, 'New York')  % Check if the person lives in New York.
        if cleanedData{i, 3} > 100000
            cleanedData{i, 7} = 'NYC High Earner';  % High earner in NYC.
        else
            cleanedData{i, 7} = 'NYC Moderate Earner';  % Moderate earner in NYC.
        end
    else
        cleanedData{i, 7} = 'Non-NYC Resident';  % For everyone outside NYC.
    end
end

% I display the final dataset with nested conditions applied.
disp('After Applying Nested Conditions:');
```

```
After Applying Nested Conditions:
```

```
disp(cleanedData);
```

```
    {'ID'}    {'Age'    }    {'Income'}    {'City'         }    {0×0 double   }    {0×0 double     }    {0×0 double
    {[ 1]}    {[     25]}    {[ 50000]}    {'New York'     }    {'High Income'}    {'Middle Income'}    {'NYC Moderate Ea
    {[ 2]}    {[39.2500]}    {[ 62000]}    {'Los Angeles'  }    {'High Income'}    {'Middle Income'}    {'Non-NYC Residen
    {[ 3]}    {[     32]}    {[     0]}    {'San Francisco'}    {'Low Income' }    {'Low Income'   }    {'Non-NYC Residen
    {[ 4]}    {[     45]}    {[500000]}    {'New York'     }    {'High Income'}    {'High Income'  }    {'NYC High Earner
    {[ 5]}    {[     55]}    {[150000]}    {'Chicago'      }    {'High Income'}    {'High Income'  }    {'Non-NYC Residen
```