Data Cleaning and Preprocessing for Premium Prediction

This lecture details the data cleaning and preprocessing steps performed on a dataset for premium prediction. We'll walk through each step, explaining the rationale and the code used. The overall goal is to prepare the data for a machine learning model.

1. Importing Libraries and Loading Data:

The first step involves importing necessary libraries like Pandas and loading the dataset into a Pandas DataFrame. It's recommended to organize your project by creating a dedicated folder containing both the dataset file and the Jupyter Notebook.

2. Standardizing Column Names:

Consistent column names are crucial for readability and ease of use. We'll convert all column names to Python's snake_case convention.

```python
import pandas as pd

# ... (Code to load the dataframe 'df')

df.columns = df.columns.str.replace(' ', '_').str.lower()
print(df.head())
```

This code snippet replaces spaces with underscores and converts all characters to lowercase, ensuring uniformity.

3. Handling Missing Values (NA):

Checking for and handling missing values is a critical step.

```python
print(df.isna().sum())  # Check for missing values in each column

df.dropna(inplace=True)  # Drop rows with any missing values

print(df.isna().sum())  # Verify that missing values are handled
```

We first assess the extent of missing data using `isna().sum()`. Given the relatively small number of missing values compared to the dataset size, we opted to drop the rows containing them using `dropna(inplace=True)`. For datasets with a significant number of missing values, imputation techniques (using mean, median, or mode) would be more appropriate.

4. Handling Duplicate Rows:

While our dataset didn't contain duplicates, it's good practice to include a check and removal step for future-proofing the code.

```python
print(df.duplicated().sum())  # Check for duplicate rows

df.drop_duplicates(inplace=True)  # Drop duplicate rows (if any)

print(df.duplicated().sum()) # Verify
```

This code snippet checks for duplicates using `duplicated().sum()` and removes them using `drop_duplicates(inplace=True)`.

5. Initial Data Exploration and Outlier Detection:

We use descriptive statistics and visualizations to understand the data and identify potential outliers.

```python
print(df.describe())  # Display summary statistics for numeric columns
```

The `describe()` function provides insights into the distribution of numeric columns, revealing potential issues like the unrealistic maximum age and negative values in the number of dependents.

6. Addressing Specific Data Errors:

We identified negative values in the 'number_of_dependents' column. After investigation and discussion with the data source, we decided to convert these to positive values, assuming a data entry error.

```
print(df[df['number_of_dependents'] < 0].shape)  # Number of rows with negative dependents
print(df['number_of_dependents'].unique())  # Unique values in the column

df['number_of_dependents'] = abs(df['number_of_dependents'])  # Convert to positive values

print(df['number_of_dependents'].describe()) # Verify
```

7. Visualizing Outliers with Box Plots:

Box plots are helpful for visualizing outliers based on the Interquartile Range (IQR) method.

```
import seaborn as sns
import matplotlib.pyplot as plt

numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns

for col in numeric_cols:
    sns.boxplot(x=df[col])
    plt.title(f"Box plot of {col}")
    plt.show()
```

This code iterates through numeric columns and generates a box plot for each, highlighting potential outliers.

8. Treating Outliers in 'age' Column:

For the 'age' column, a simple threshold of 100 was used to remove unrealistic values.

```
df1 = df[df['age'] <= 100].copy() # Create a new dataframe with filtered age
print(df1.describe()) # Verify
```

9. Treating Outliers in 'income' Column:

The 'income' column was analyzed using a histogram and the IQR method. However, the IQR upper bound was deemed too restrictive based on domain knowledge. Instead, the 99.9th percentile was used as a threshold.

```
def get_iqr_bounds(column):
    q1 = column.quantile(0.25)
    q3 = column.quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    return lower_bound, upper_bound

# Example of IQR calculation (though not used in the final approach)
lower_bound, upper_bound = get_iqr_bounds(df1['income_lacks'])
print(f"Lower Bound: {lower_bound}, Upper Bound: {upper_bound}")

threshold = df1['income_lacks'].quantile(0.999) # Using 99.9th percentile
print(f"Threshold: {threshold}")

df2 = df1[df1['income_lacks'] <= threshold].copy() # Create new dataframe with filtered income
print(df2.describe())  # Verify changes
```

This concludes the data cleaning and preprocessing steps. The resulting DataFrame `df2` is now ready for further analysis and model building. Remember, data cleaning is an iterative process and often requires domain expertise and careful consideration of the data's context.

Exploratory Data Analysis (EDA) Continued: Univariate and Bivariate Analysis

This lecture builds upon the previous data cleaning steps and delves into exploratory data analysis (EDA), specifically univariate and bivariate analysis. We'll examine the distribution of numeric variables, explore relationships between variables, and prepare the data for feature engineering.

1. Distribution of Numeric Columns (Univariate Analysis):

Histograms are used to visualize the distribution of numeric data. This helps understand the data's characteristics, such as skewness and central tendency.

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# ... (Code to load the cleaned dataframe 'df2')

for col in numeric_cols:
    plt.figure()  # Create a new figure for each plot
    sns.histplot(df2[col])
    plt.title(f"Distribution of {col}")
    plt.show()
```

This code iterates through the numeric columns and generates a histogram for each. We've added `plt.figure()` to ensure each histogram is displayed separately, improving clarity.

2. Improving Visualization Layout:

To enhance readability, especially with multiple plots, we can arrange them in a grid-like structure. While tools like ChatGPT can assist with generating the code for this, it's crucial to understand the code and not rely solely on copy-pasting.

```python
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(2, 2, figsize=(12, 8)) # 2 rows, 2 columns of plots

sns.histplot(df2['age'], ax=axes[0, 0])
axes[0, 0].set_title('Distribution of Age')

# ... (Similar code for other numeric columns - income_lacks, number_of_dependents, annual_premium_amount)

plt.tight_layout() # Adjusts subplot params so that subplots are nicely fit in the figure
plt.show()
```

This revised code uses `matplotlib.pyplot.subplots` to create a grid of subplots and assigns each histogram to a specific location within the grid.

3. Bivariate Analysis with Scatter Plots:

Scatter plots are used to visualize the relationship between two numeric variables.

```python
sns.scatterplot(x='age', y='annual_premium_amount', data=df2)
plt.title("Scatter Plot of Age vs. Annual Premium Amount")
plt.show()
```

This code creates a scatter plot showing the relationship between 'age' and 'annual_premium_amount'. The observed positive correlation confirms the intuitive understanding that premium amounts tend to increase with age.

4. Bivariate Analysis for Multiple Variables:

We can extend this to visualize relationships between other numeric independent variables and the target variable ('annual_premium_amount').

```
independent_numeric_cols = ['age', 'number_of_dependents', 'income_lacks']

fig, axes = plt.subplots(1, len(independent_numeric_cols), figsize=(15, 5))

for i, col in enumerate(independent_numeric_cols):
    sns.scatterplot(x=col, y='annual_premium_amount', data=df2, ax=axes[i])
    axes[i].set_title(f"Scatter Plot of {col} vs. Annual Premium Amount")


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

This code generates multiple scatter plots, arranged horizontally, to visualize the relationship between each independent numeric variable and the target variable.

5. Analyzing Categorical Columns:

We then shift our focus to categorical columns, exploring their unique values and distributions.

```
categorical_cols = ['gender', 'region', 'marital_status', 'smoking_status', 'income_level', 'medical_history', 'insurance_plan']

for col in categorical_cols:
    print(f"Unique values in {col}: {df2[col].unique()}")
```

This code prints the unique values for each categorical column, revealing inconsistencies in the 'smoking_status' column.

6. Handling Inconsistent Categorical Values:

We address the inconsistencies in 'smoking_status' by replacing redundant categories with a standardized value ('no_smoking').

```
df2['smoking_status'].replace({'smoking_zero': 'no_smoking', 'does_not_smoke': 'no_smoking', 'not_smoking': 'no_smoking'}, inplace
print(f"Unique values in smoking_status after cleaning: {df2['smoking_status'].unique()}")
```

This code replaces variations of "no smoking" with a single consistent value.

7. Univariate Analysis of Categorical Columns:

We then analyze the frequency distribution of categorical variables using `value_counts()` and visualize them using bar plots.

```
percentage_count = df2['gender'].value_counts(normalize=True)
sns.barplot(x=percentage_count.index, y=percentage_count.values)
plt.title("Gender Distribution")
plt.ylabel("Percentage")
plt.show()


# ... (Similar code for other categorical columns using subplots for better layout)
```

This code calculates and visualizes the distribution of the 'gender' column. Similar code can be applied to other categorical columns using subplots for a more organized display.

8. Bivariate Analysis of Categorical Columns:

We use `pd.crosstab` to analyze the relationship between two categorical variables. This generates a contingency table showing the frequency of each combination of categories.

```python
cross_tab = pd.crosstab(df2['income_level'], df2['insurance_plan'])
print(cross_tab)


cross_tab.plot(kind='bar', stacked=True)
plt.title("Income Level vs. Insurance Plan")
plt.ylabel("Count")
plt.show()


sns.heatmap(cross_tab, annot=True, fmt="d")
plt.title("Income Level vs. Insurance Plan (Heatmap)")
plt.show()
```

This code demonstrates how to generate a cross-tabulation and visualize it using both stacked bar plots and heatmaps. The `fmt="d"` argument ensures integer values are displayed in the heatmap.

This concludes the EDA portion, providing a comprehensive understanding of both numeric and categorical variables and their relationships. The next step is feature engineering, which will prepare the data for machine learning model training.