Preprocessing and Model Training Pipeline

T316 – Capstone Project - AutoGuru

A Report

Written For

QUT

Prepared by

Jack Langdon

Submitted in partial fulfillment of project requirements for

QUT - T316 - Capstone Project - AutoGuru

Document Control

Document Title	Preprocessing Pipeline
Project	Vantage
Revision	
Date	5/08/25
Prepared By	Alvin Luu
Checked By	Theo Dela Cruz
Approved By	

Revision History

Revision	Date	Author	Description	Approved

Table of Contents

1 Document Purpose

The purpose of this document is to explain the reasoning behind the steps taken in the preprocessing pipeline and model training. It details the rationale behind each preprocessing step, from data cleaning to feature engineering, and explains its impact on the overall model development and performance. This documentation serves as a key reference for understanding the project's methodology, ensuring that the solution can be accurately interpreted and analysed by stakeholders and future collaborators.

2 Scope

This documentation outlines the specific preprocessing steps employed to prepare the dataset for the project's machine learning models. It covers all data manipulation techniques, including handling missing values, feature scaling, and the integration of any external datasets, with a focus on justifying each decision. The scope is intentionally narrow, concentrating solely on the preprocessing pipeline and model training rather than the complete project methodology. This documentation is intended to be a creference for evaluating the final model's precision and reliability for both the client and the end consumer.

3 Preprocessing Pipeline

To create and develop the final model, numerous preprocessing techniques were implemented to suit the project requirements. This can be depicted in a step-by-step process alongside the raw code in the boxes below.

3.1 Step 1: Feature Engineering

To reduce the impact of data drift, inflation consumer price index (CPI) data was gathered and used to create the AdjustedPrice column from the raw price. This was a step to account for inflation over time, ensuring our model trains on a more accurate representation of historical data. The original date data was also transformed into a new BCreatedDateAEST column, which is compliant with the libraries used. By matching the booking dates with their corresponding inflation rates, we can ensure the CPI was applied correctly, which ultimately improves model performance on recent data.

We also created the Distance and Months columns by extracting context from the TaskName. This gives the model a deeper understanding of the different booking logs—capped and prescribed—allowing it to learn more meaningful trends. To further enhance the model's performance, a Label column was created by extracting only booking's status after submission. The model will use this column to later clean in step 4 so that the model only learns approved bookings pricing.

```
#Create Columns

df['Distance'] = None df['Months'] = None

mask = df['BTicketType'] == 'Log' split_1 = df.loc[mask,
'TaskName'].str.split(' - ', n=1, expand=True) df.loc[mask,
'TaskName'] = split_1[0] df.loc[mask, 'Rest'] =
split_1[1].fillna('')

split_2 = df.loc[mask, 'Rest'].str.split(' / ', n=1, expand=True)
df.loc[mask, 'Distance'] = split_2[0] df.loc[mask, 'Months'] =
split_2[1]

mask = df['BTicketType'] == 'Capped' split_1 = df.loc[mask,
'TaskName'].str.split(' - ', n=1, expand=True) df.loc[mask,
'TaskName'] = split_1[0] df.loc[mask, 'Distance'] =
split_1[1].fillna('')
```

```
mask = df['BTicketType'] == 'Prescribed' split_1 = df.loc[mask,
'TaskName'].str.split(' - ', n=1, expand=True) df.loc[mask,
'TaskName'] = split_1[0] df.loc[mask, 'Distance'] =
split_1[1].fillna('')

df = df.drop(columns=['Rest'])

#Create Adjusted Price based on CPI

#Create date column

df['BCreatedDateAEST'] = pd.to_datetime(df['BCreatedDateAEST'],
format='mixed', dayfirst=True); df['Date'] =
df['BCreatedDateAEST'].dt.date

#Price Adjustment

df = df[df['PriceIncGSTRaw'] != 0] df['Date'] =
pd.to_datetime(df['Date']) # ensure 'date' is in datetime format
df['Quarter'] = df['Date'].dt.to_period('Q').astype(str) df =
df.merge(cpi_index, on='Quarter', how='left') base_cpi =
cpi_index['CPI'].iloc[-1] df['AdjustedPrice'] =
round(df['PriceIncGSTRaw'] * (base_cpi / df['CPI']), 2) # round to
2 decimal places for consistency df['AdjustedPrice'] =
round(df['AdjustedPrice'].fillna(df['PriceIncGSTRaw'])) # use
original price where CPI is missing (for current quarter which
does not need adjustment)

df['Label'] = df['BStatusAfterSubmitted'].map({'33. Approved': 1,
'16. Requires Changes': 0, '29. Rejected': 0}) # create label
column for model training
```

3.2 Step 2: Data Column Deletion

Deleting irrelevant columns was crucial for ensuring the data was both relevant and free of anomalies. To minimize the effect of irrelevant trends on the model's learning, we reviewed the data and removed less informative columns. This included the removal of the original raw price data, which was replaced by the new inflation-adjusted data, and BStatusAfterSubmitted, which became redundant after we extracted the relevant information for the new Label column. Crucially the Label column is initially retained until step 4 where it is deleted after it subsequent use for cleaning the rows.

The columns FCID and VMid were also removed as they did not provide any meaningful data beyond what was already available in other columns. Including these redundant or anomalous data columns could lead the model to learn skewed or false trends.

```
#Rename Columns

df = df.rename(columns={ 'cVMake': 'Make', 'cVMakeModel': 'Model',
'cVYear': 'Year', 'idFuel': 'FuelType', 'idLitres': 'EngineSize',
'idTransmission': 'Transmission', 'idDrive': 'DriveType',
'idIsHybrid': 'IsHybrid', 'BOdoNum' : 'Odometer', })

#Drop Columns that will not be used

columns_to_drop = ['ï»¿FCID', 'BOdoText', 'VMid'] df.drop(columns
= columns_to_drop, inplace=True) df.head()

#Remove Columns that will not be used

columns_to_drop = [ 'BCreatedDateAEST', 'CPI', 'Quarter',
'PriceIncGSTRaw' , 'BShopID', 'BShopRegionName', 'BShopPostcode',
'IsDeleted', 'BStatusFromDateTimeAEST', 'Date', 'BStatusFinal', ]
df.drop(columns = columns_to_drop, inplace=True) df.head()

df = df.drop(columns=['BStatusAfterSubmitted'], axis=1) # drop the
original status column as it is no longer needed

#Final Column Removal

columns_to_drop = ['BookingID', 'IsCustomService',
'IsCustomRepair', 'BShopRegionClass', 'BShopState', 'IsHybrid',
'Label', 'Odometer'] df.drop(columns = columns_to_drop,
inplace=True) df.head()
```

3.3 Step 3: Outlier Removal

Outliers were removed based on task name to remove extreme values.

```
#Functions
def remove_outliers(df, group_by="TaskName", value_col="price"):
def filter_group(group): Q1 = group[value_col].quantile(0.25) Q3 =
group[value_col].quantile(0.75) IQR = Q3 - Q1 lower = Q1 - 1.5 *
IQR upper = Q3 + 1.5 * IQR return group[(group[value_col] >=
lower) & (group[value_col] <= upper)]

return df.groupby(group_by, group_keys=False).apply(filter_group)
```

```
print(f"size before: {df.shape}" ) df = remove_outliers(df,
group_by="TaskName", value_col="AdjustedPrice") print(f"size
after: {df.shape}" )
```

3.4 Step 4: Data Row Cleaning

A key pre-processing operation involved identifying and removing missing or invalid data, as well as dropping duplicate entries. Similarly, rows flagged as a IsCustomService or IsCustomRepair were removed entirely, as these entries lacked the common patterns needed for the model to learn effectively. As our predictive model will only require approved data entries, all rejected and stagnant require changes entries were removed. This was done through filtering the Label column to remove all row that weren't approved. Each booking managed by AutoGuru can also have multiple tickets. When a booking is modified, every ticket within that booking is updated and appended as a new data entry, even if a particular ticket was not modified. If left in the dataset, these duplicate entries could introduce a bias toward the duplicated data.

To address this, we implemented logic to group and drop all unchanged duplicate data entries from the dataset, which prevents the model from learning false relationships. Following the removal of these false negatives, we also cleaned up missing or invalid data, such as negative or zero prices. We also discussed with AutoGuru which tasks require manual scrutiny; thus, entries with the task names 'product' and 'tyres' were also dropped from the model's training data. This ensures that only valid data is used and that no invalid trends are learned by the model.

After this final cleaning process, the following columns were retained as the most useful features for our model: BTicketType, TaskName, Make, Model, Year, FuelType, Engine Size, Transmission, DriveType, Distance, Months and AdjustedPrice.

```
#Remove custom services and repairs

df = df[df['IsCustomService'] != '1'] df = df[df['IsCustomRepair']
!= '1'] df = df[df['TaskName'] != 'Custom Repair']

#Remove products and tyres as they are unpredictable without
quantity information

df = df[df['TaskName'] != '((Products))'] df = df[df['TaskName']
!= '((Tyres))'] df = df[df['TaskName'] != 'Tyre Replacement']
```

```
#Remove Rare ticket types

df = df[df['BTicketType'] != 'OtherTicket'] df =
df[df['BTicketType'] != 'Custom'] df = df[df['BTicketType'] !=
'Basic']

#Remove Tickets with zero or negative prices

df = df[df['PriceIncGSTRaw'] > 0]

#Remove Duplicate Rows

print(f"size before: {df.shape}" ) df =
df.drop_duplicates(subset=[col for col in df.columns if col not in
['BookingID', 'BTicketID', 'BCreatedDateAEST',
'BStatusFromDateTimeAEST']]) print(f"size after: {df.shape}" )

#Remove False Negatives

False negatives are rows which have a status of '16. Requires
Changes' when they should have a status of '33. Approved'. This
occurs in the dataset as bookings contain multiple tickets. If a
single ticket in a booking requires changes, the entire booking is
marked as 'Requires Changes', thus marking tickets that do not
require changes incorrectly. These false negatives can be detected
and removed by checking if a duplicate entry exists where only the
status changes.

print(f"size before: {df.shape}" ) df['StatusPriority'] =
df['BStatusAfterSubmitted'].apply(lambda x: 0 if x == '33.
Approved' else 1) # Assign priority: approved gets highest
priority (lowest number) dedup_cols = [col for col in df.columns
if col not in ['BStatusAfterSubmitted',
'StatusPriority','BCreatedDateAEST', 'BStatusFromDateTimeAEST',
'Date' ]] # Define columns to check for duplicates df =
df.sort_values(by=dedup_cols + ['StatusPriority']) # Sort so
approved status is first df =
df.drop_duplicates(subset=dedup_cols, keep='first') # Keep the
first occurrence (which is the approved status) df =
df.drop(columns='StatusPriority') # drop the temporary column used
for sorting

df['Label'] = df['BStatusAfterSubmitted'].map({'33. Approved': 1,
'16. Requires Changes': 0, '29. Rejected': 0}) # create label
column for model training df =
df.drop(columns=['BStatusAfterSubmitted'], axis=1) # drop the
```

```
original status column as it is no longer needed print(f"size
after: {df.shape}" )

#Remove Rejected Cases

df = df[df['Label'] == 1];

#Fill in missing distances with odometer

df["Distance"] = df["Distance"].fillna(df["Odometer"])
```

3.5 Step 5: Data Splitting

With the decision to investigate each service independently, a final step was taken to split the dataframe by ticket type and export. This created separate CSV files for the different task types: Capped, Log, Prescribed, and Repair.

```
#Split data frame by ticket type

df_repair = df[df['BTicketType'] ==
'Repair'].drop(columns=['BTicketType'])

df_log = df[df['BTicketType'] ==
'Log'].drop(columns=['BTicketType'])

df_capped = df[df['BTicketType'] ==
'Capped'].drop(columns=['BTicketType'])

df_prescribed = df[df['BTicketType'] ==
'Prescribed'].drop(columns=['BTicketType'])

import os

#Define paths

paths = { "repair":
"../backend/data/preprocessed_repair_data.csv", "log":
"../backend/data/preprocessed_log_data.csv", "capped":
"../backend/data/preprocessed_capped_data.csv", "prescribed":
"../backend/data/preprocessed_prescribed_data.csv", }

#Make sure the directories exist

for path in paths.values(): dir_path = os.path.dirname(path)
os.makedirs(dir_path, exist_ok=True)

#Save CSVs
```

```
df_repair.to_csv(paths["repair"], index=False)
df_log.to_csv(paths["log"], index=False)
df_capped.to_csv(paths["capped"], index=False)
df_prescribed.to_csv(paths["prescribed"], index=False)

print("All files saved successfully:") for name, path in
paths.items(): print(f"{name}: {os.path.abspath(path)}")
```

3.6 Step 6: Removing Rare Categories

To reduce redundant features and help the model focus on predicting common bookings, we removed rare tasks with high variance. Instead of grouping them, we used a threshold to determine which categories were considered rare. For repair tasks, the threshold was set at 100, and for Log, Capped, and Prescribed tickets, the vehicle Model threshold was set at 20. Any entries that fell below these thresholds were removed. This data reduction manages the cardinality of the dataset, encouraging the model to find trends in more common outliers while reducing potential noise.

```
#Function

def drop_rare(df, column, threshold, df_name="DataFrame"):
print(f"----Dropping Rare {column}s from {df_name}----")
print(f"{column}s before: {len(df[column].unique())}") counts =
df[column].value_counts() rare_values = counts[counts <
threshold].index df = df[~df[column].isin(rare_values)].copy()
print(f"{column}s after: {len(df[column].unique())}") return df

#Dropping rare cases

df_repair = drop_rare(df_repair, 'TaskName', 100, df_name="Repair
DF") df_log = drop_rare(df_log, 'Model', 20, df_name="Log DF")
df_capped = drop_rare(df_capped, 'Model', 20, df_name="Capped DF")
df_prescribed = drop_rare(df_prescribed, 'Model', 20,
df_name="Prescribed DF")
```

3.7 Step 7: Prescribed Data Cleaning

To improve the performance of the prescribed model, the prescribed data set was cleaned to only contain road vehicles. This was done as the variations in non-road vehicles was hurting performance and had inconsistent features.

```
#Cleaning Prescribed Data
```

```
non_road_vehicles = [ "MAXICUBE 24 PALLET", "SCHMITZ CARGOBULL
REFRIGERATED TRAILER", "MAXICUBE 12 PALLET", "EQUIPMENT FRIDGE",
"MAXITRANS MAXICUBE", "TOMMYGATE TAILGATE LOADER", "UNDEFINED
TRAILE", "CATERPILLAR 3.5T COUNTER BALANCE", "CATERPILLAR 5T
COUNTER BALANCE", "CATERPILLAR 5.5T COUNTER BALANCE", "CARRIER
VECTOR", "ACMC PO6 CO2", "MAXICUBE 22 PALLET", "MAXICUBE 16
PALLET", "EQUIPMENT TAILGATE 12M", "CROWN 2.5T COUNTER BALANCE",
"THERMO KING V-600 MAX", "FREIGHTER 24 PALLET TRI AXLE", "TRAILER
AUST 8 X 5 BOX", "EQUIPMENT CRANE", "EQUIPMENT VAWTRA", "HYSTER 2T
COUNTER BALANCE", "TOYOTA 3T COUNTER BALANCE", "DHOLLANDIA DH-
LMA.20", "EQUIPMENT VAWTRA" ]

df_prescribed =
df_prescribed[~df_prescribed['Model'].isin(non_road_vehicles)]

df_prescribed['Model'].unique()
```

4 Model Training

Following the preprocessing pipeline, the cleaned data was then used to train the four models. Each dataset was further split into training, validation and test sets with the following random split:

- Training Set: 70%
- Validation Set: 15%
- Testing Set: 15%

A grid search was performed for each model across the following parameter grid:

Hyperparameter	Values
Depth	6, 8, 10
Learning Rate	0.01, 0.03, 0.05, 0.1
L2 Leaf Regularisation	1, 3, 5, 7
Iterations	1000, 1500
Subsample	0.8, 0.9, 1.0
Column sample by level	0.8, 0.9, 1.0
Minimum child samples	5, 10, 20

The following sections details the selected hyperparameter values and what columns were used along with the performance achieved.

4.1 Capped Price Service Model

The capped price service model was trained on the following columns: Make, Model, Year, FuelType, EngineSizem Transmission, DriveType and Distance.

Hyperparameter Selection

Hyperparameter	Value
Depth	10
Learning Rate	0.1
L2 Leaf Regularisation	7
Iterations	1500
Subsample	1
Column sample by level	0.8
Minimum child samples	10

Performance on Test Set

Metric	Result
MAE	40.16
RMSE	69.23
MAPE	10.16%

4.2 Logbook Service Model

The logbook price service model was trained on the following columns: Make, Model, Year, FuelType, EngineSize, Transmission, DriveType, Distance and Months.

Hyperparameter Selection

Hyperparameter	Value
Depth	10

Learning Rate	0.1
L2 Leaf Regularisation	7
Iterations	1500
Subsample	1
Column sample by level	0.8
Minimum child samples	10

Performance on Test Set

Metric	Result
MAE	76.78
RMSE	114
MAPE	16.34%

4.3 Prescribed Service Model

The prescribed service model was trained on the following columns: Make, Model, Year, FuelType, EngineSize, Transmission, DriveType and Distance

Hyperparameter Selection

Hyperparameter	Value
Depth	10
Learning Rate	0.1
L2 Leaf Regularisation	7
Iterations	1500
Subsample	1
Column sample by level	0.8
Minimum child samples	10

Performance on Test Set

Metric	Result
MAE	236.86
RMSE	352.61
MAPE	22.94%

4.4 Repair Service Model

The repair service model was trained on the following columns: TaskName, Make, Model, Year, FuelType, EngineSize, Transmission, DriveType and Distance.

Hyperparameter Selection

Hyperparameter	Value
Depth	10
Learning Rate	0.1
L2 Leaf Regularisation	7
Iterations	1500
Subsample	1
Column sample by level	0.8
Minimum child samples	10

Performance on Test Set

Metric	Result
MAE	86.94
RMSE	222.21
MAPE	49.34%

5 Conclusion

The pre-processing techniques used in this project maximised the informative relationships and trends learnt by the final model, whilst removing potential noise and outliers that may

negatively impact the results. Each of these steps were integral to ensuring not only an accurate model, but also one with high validity and precision.