FEATURE ENGINEERING End-to End PROJECT (30M)

AIML Certification Programme

Business Understanding (1M)

Students are expected to identify a regression problem of your choice. You have to detail the Business Understanding part of your problem under this heading which basically addresses the following questions.

1. What is the business problem that you are trying to solve?
2. What data do you need to answer the above problem?What are the different sources of data?

1. Business problem

The goal of this project is to solve the following business problem:

"How can we accurately predict the selling price of products listed on Mercari's marketplace using the product's attributes and listing details?"

This problem is valuable to both:

- **Sellers**: Accurate price predictions can help sellers list items competitively, leading to faster and more successful sales.
- **Mercari**: Better pricing guidance enhances buyer trust, improves search relevance, and increases platform revenue by boosting completed transactions.

2. What Data Is Needed?

To solve this regression problem, we require a dataset containing:

- **Target Variable**:
  - `price` – Final sale price of the product.
- **Feature Variables**:
  - `name` – Product title.
  - `item_description` – Description of the product.
  - `brand_name` – Brand of the product.
  - `category_name` – Category path of the item (e.g., "Electronics/Computers").
  - `item_condition_id` – Condition rating provided by the seller.
  - `shipping` – Who pays for shipping (0 = buyer, 1 = seller).

We may also engineer additional features like:

- Text length or word count of `item_description`.
- Whether the `brand_name` is missing.
- First-level category extraction (e.g., "Women", "Men", "Home").

Data Sources

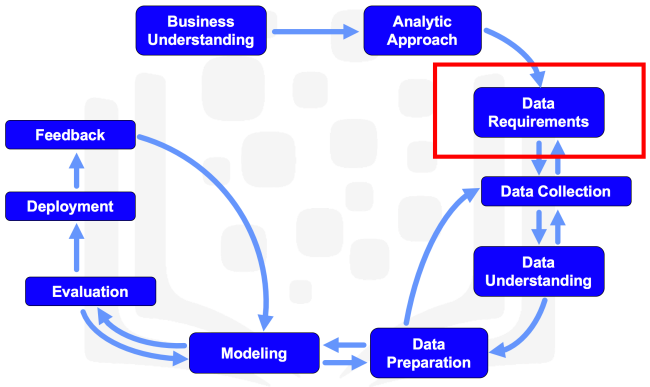
The dataset used for this project comes from:

- **Mercari Price Suggestion Challenge** – Kaggle
  Link: https://www.kaggle.com/competitions/mercari-price-suggestion-challenge/data

If expanded further, external data could be considered, such as:

- Market pricing data from similar marketplaces.
- Brand/product reputation information from external APIs.
- User behavior analytics (views, clicks, favorites).

Data Requirements and Data Collection (3+1M)



In the initial data collection stage, data scientists identify and gather the available data resources. These can be in the form of structured, unstructured, and even semi-structured data relevant to the problem domain.

Identify the required data that fulfills the data requirements stage of the data science methodology
Mention the source of the data.(Give the link if you have sourced it from any public data set) Briefly explain the data set identified .

Identified Data

To solve the regression problem of predicting the price of items on an e-commerce platform like Mercari, the required data must include:

- A continuous target variable: `price`
- A mix of **categorical** and **textual** features (e.g., brand, category, item condition)
- Potentially useful engineered features from text and categories

The dataset must allow for:

- Data cleaning (e.g., missing values, inconsistencies)
- Feature engineering (e.g., binning, encoding)
- Regression modeling
- Feature selection techniques like Chi-Squared and Mutual Information

---

Data Source

The data used in this project is sourced from a public competition:

- **Dataset**: Mercari Price Suggestion Challenge
- **Platform**: Kaggle

🔗 Click here to access the dataset

Dataset Description

The Mercari dataset contains approximately **1.5 million** records of item listings, with the goal of predicting the **final sale price** of an item based on the product's listing details. Each record includes:

- `train_id` – Unique identifier
- `name` – Product title
- `item_condition_id` – Item's condition rating (1 to 5)
- `category_name` – Full product category (hierarchical string)
- `brand_name` – Product brand (may be missing)
- `price` – Target variable (continuous, USD)
- `shipping` – Who pays shipping (0 = buyer, 1 = seller)
- `item_description` – Free-text description of the item

This dataset is ideal for demonstrating:

- Data preprocessing and cleaning
- Text feature engineering
- Binning and discretization
- Feature selection (Chi-Squared & Mutual Information)
- Regression modeling using linear regression

Imports

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import re
import seaborn as sns
from sklearn.feature_selection import chi2, mutual_info_regression, SelectKBest
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler, StandardScaler
```

Import the above data and read it into a data frame

In [ ]:
```
# Load the dataset (Mercari data is tab-separated, not comma-separated!)
df = pd.read_csv("train.tsv", sep="\t")

# Sampling the data
df = df.sample(n=10000, random_state=42).copy()
# Compute description length
df['desc_len'] = df['item_description'].apply(lambda x: len(x.split()))

# Define y and X
y = df['price']
X = df.drop(columns=['price', 'name', 'item_description'], errors='ignore')
```

Note: Sampling and Feature Creation

To ensure efficient processing and avoid memory-related issues due to the large original dataset (~1.4 million rows), we sampled 10,000 rows using a fixed random_state for reproducibility. This subset was used throughout the entire workflow—from preprocessing to modeling.

Additionally, we introduced a new feature called desc_len, which captures the number of words in the product description. This acts as a proxy for text content richness, potentially influencing price. The desc_len feature is computed during this initial stage and added directly to the dataset for further processing.

Confirm the data has been correctly by displaying the first 5 and last 5 records.

In [ ]:
```
# Display the first 5 rows
print("First 5 records:")
print(df.head())

# Display the last 5 rows
print("\nLast 5 records:")
print(df.tail())
```

Get the dimensions of the dataframe.

In [ ]:
```
# Get the dimensions of the DataFrame
print("DataFrame dimensions (rows, columns):")
print(df.shape)
```

Display the description and statistical summary of the data.

In [ ]:
```
# Description of data
df.info()

# Statistical summary of numerical columns
print("Statistical Summary (Numerical Columns):")
print(df.describe())

# Including summary of object (categorical/text) columns:
print("\nStatistical Summary (All Columns Including Object Types):")
print(df.describe(include='all'))
```

Display the columns and their respective data types.

In [ ]:
```
# Display column names and their data types
print("Columns and Data Types (as DataFrame):")
print(pd.DataFrame({'Column': df.columns, 'Data Type': df.dtypes.values}))
```

Convert the columns to appropriate data types

In [ ]:
```
# Convert data types appropriately
df['name'] = df['name'].astype(str)
df['item_description'] = df['item_description'].astype(str)
```

```
df['category_name'] = df['category_name'].astype('category')
df['brand_name'] = df['brand_name'].astype('category')
df['shipping'] = df['shipping'].astype('int8')
df['item_condition_id'] = df['item_condition_id'].astype('int8')
df['desc_len'] = df['desc_len'].astype('float32')
df['price'] = df['price'].astype('float32')

# Optional ID field
if 'train_id' in df.columns:
    df['train_id'] = df['train_id'].astype('int32')
elif 'test_id' in df.columns:
    df['test_id'] = df['test_id'].astype('int32')

print("Data types converted successfully.")
print(df.dtypes)
```

Write your observations from the above.

Observations

1. **Categorical Data Identified**:
   - Columns like `category_name` and `brand_name` were converted to `category` type.
   - This reduces memory usage and prepares them for encoding (e.g., one-hot or label encoding).
2. **Textual Data Recognized**:
   - `name` and `item_description` are stored as strings.
   - These will likely require text preprocessing (e.g., tokenization, TF-IDF) before use in modeling.
3. **Numerical Columns Optimized**:
   - `shipping` and `item_condition_id` were converted to smaller integer types (`int8`) to optimize memory.
   - Convert `desc_len` to `float32` so it works correctly with preprocessing and ML models, which expect numeric data with continuous values.
   - `price` was set to `float32` , sufficient for precision while reducing memory compared to `float64` .
4. **ID Columns Cast Properly**:
   - ID column like `train_id` was cast to `int32` , appropriate for identifiers.

Check for Data Quality Issues (1.5M)

- duplicate data
- missing data
- data inconsistencies

In [ ]:
```
# Check for duplicate rows
duplicate_rows = df[df.duplicated()]
print(f"Number of duplicate rows: {duplicate_rows.shape[0]}")
```

In [ ]:
```
# Check missing values in each column
missing_values = df.isnull().sum()
print("Missing values per column:")
print(missing_values)

# Percentage of missing data
missing_percent = (df.isnull().sum() / len(df)) * 100
print("\nPercentage of missing values per column:")
print(missing_percent)
```

In [ ]:
```
# Check for blank strings in 'name' and 'item_description'
blank_names = df['name'].str.strip() == ''
blank_descriptions = df['item_description'].str.strip() == ''

print(f"Blank 'name' entries: {blank_names.sum()}")
print(f"Blank 'item_description' entries: {blank_descriptions.sum()}")
```

Handling the data quality issues(1.5M)

Apply techniques

- to remove duplicate data
- to impute or remove missing data
- to remove data inconsistencies

Give detailed explanation for each column how you handle the data quality issues.

In [ ]:
```python
# a) Remove duplicate data
# Before removing duplicates
print("Before duplicate removal:", df.shape)

# Remove duplicates based on unique ID
id_col = 'train_id' if 'train_id' in df.columns else 'test_id'
df.drop_duplicates(subset=id_col, inplace=True)

# After removing duplicates
print("After duplicate removal:", df.shape)
```

In [ ]:
```python
# b) Impute or remove missing data
# View missing values
print("Missing values per column:")
print(df.isnull().sum())

# Example strategy: fill numeric columns with median
numeric_cols = df.select_dtypes(include=['number']).columns
df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].median())

# Ensure brand_name is categorical
df['brand_name'] = df['brand_name'].astype('category')

# Add 'Unknown' category if it does not exist
if 'Unknown' not in df['brand_name'].cat.categories:
    df['brand_name'] = df['brand_name'].cat.add_categories(['Unknown'])

# Now fill missing values
df['brand_name'] = df['brand_name'].fillna('Unknown')

# # Ensure category_name is categorical
# df['category_name'] = df['category_name'].astype('category')

# # Add 'unknown/unknown/unknown' only if it's not already in the categories
# if 'unknown/unknown/unknown' not in df['category_name'].cat.categories:
#     df['category_name'] = df['category_name'].cat.add_categories(['unknown/unknown/unknown'])

# # Now fill missing values
# df['category_name'] = df['category_name'].fillna('unknown/unknown/unknown')
# # Split into 3 subcategories
categories = df['category_name'].str.split('/', expand=True)
df['cat_1'] = categories[0].astype('category')
df['cat_2'] = categories[1].astype('category')
df['cat_3'] = categories[2].astype('category')

# # Categorical columns - Fill remaining with "Unknown"
# cat_cols = df.select_dtypes(include=['object', 'category']).columns
# df[cat_cols] = df[cat_cols].fillna("Unknown")
cat_cols = df.select_dtypes(include=['category']).columns
obj_cols = df.select_dtypes(include=['object']).columns

# For categorical columns: add 'Unknown' category first, then fillna
for col in cat_cols:
    if 'Unknown' not in df[col].cat.categories:
        df[col] = df[col].cat.add_categories(['Unknown'])
    df[col] = df[col].fillna('Unknown')

# For object columns, fillna directly
df[obj_cols] = df[obj_cols].fillna('Unknown')
```

In [ ]:
```python
# c) Remove data inconsistencies (column-wise explanation)
# Strip whitespace and convert to lowercase in all object columns
for col in df.select_dtypes(include='object').columns:
    df[col] = df[col].str.strip().str.lower()

# Fix 'item_description': replace known placeholder and fill NaN
df['item_description'] = df['item_description'].replace('No description yet', np.nan)
df['item_description'] = df['item_description'].fillna('no description')

# Clean text fields
df['name'] = df['name'].str.lower().str.strip()
df['item_description'] = df['item_description'].str.lower().str.strip()
df['item_description'] = df['item_description'].apply(lambda x: re.sub(r'[^\w\s]', '', x))

# Sanitize item_condition_id (should be 1-5)
df['item_condition_id'] = df['item_condition_id'].apply(lambda x: x if x in [1, 2, 3, 4, 5] else 3).astype('int8')

# Validate shipping
df['shipping'] = df['shipping'].apply(lambda x: 1 if x == 1 else 0).astype('int8')

# Remove zero or negative prices if present
if 'price' in df.columns:
    df = df[df['price'] > 0]
```

Note

To prepare the Mercari dataset for modeling, we implemented the following systematic cleaning procedures:

- **No Duplicate Data:** Verified that there were no duplicate entries based on the unique identifier (`train_id` or `test_id`), ensuring each record is unique and preventing data leakage.
- **Missing Value Imputation:** Missing values in numeric columns were filled with the median values to maintain distribution integrity. For categorical columns such as `brand_name` and `category_name` , missing entries were filled with `"Unknown"` after adding it as a valid category to avoid losing records.
- **Category Splitting:** The hierarchical `category_name` string was split into three separate categorical columns (`cat_1` , `cat_2` , and `cat_3`) to better capture granular category information.
- **Text Cleaning:** All text columns (`name` , `item_description`) were standardized by converting to lowercase, stripping whitespace, and removing special characters to reduce noise and inconsistencies.
- **Handling Placeholders:** Common placeholders such as `"No description yet"` in `item_description` were replaced with a standard `"no description"` tag to signify missing descriptive content.
- **Validation and Correction:** Values in `item_condition_id` were constrained to valid integers (1 through 5), and `shipping` was converted to a strict binary indicator (0 or 1).
- **Filtering Invalid Prices:** Entries with zero or negative prices were excluded, ensuring the model trains on meaningful price values.

Standardise the data (1M)

Standardization is the process of transforming data into a common format which you to make the meaningful comparison.

In [ ]:
```
# Let's say you want to standardize numerical features
features_to_standardize = ['price', 'desc_len']

scaler = StandardScaler()
df_std = df.copy()
df_std[features_to_standardize] = scaler.fit_transform(df_std[features_to_standardize])

print("Standardized data:")
print(df_std[features_to_standardize].describe())
```

Note on Standardization:

Standardization transforms features to have a mean of zero and a standard deviation of one. This is essential for algorithms like Linear Regression to perform optimally, as it ensures all features are on the same scale, preventing dominance by features with larger magnitude.

Normalise the data wherever necessary(1M)

In [ ]:
```
# Step 1: Create a copy of your original DataFrame to preserve raw data
df_norm = df.copy()

# Step 2: Log-transform skewed data (e.g., 'price')
df_norm['log_price'] = np.log1p(df_norm['price'])  # Safe even for zero prices

# Step 4: Define features to normalize (log-transformed and others)
features_to_normalize = ['log_price', 'desc_len']

# Step 5: Apply Min-Max normalization
scaler = MinMaxScaler()
# df_norm[[f + '_norm' for f in features_to_normalize]] = scaler.fit_transform(df_norm[features_to_normalize])
df_norm['log_price_norm'] = scaler.fit_transform(df_norm[['log_price']])
df_norm['desc_len_norm'] = scaler.fit_transform(df_norm[['desc_len']])

# Step 6: View normalized columns
print("Normalized Data Preview:")
print(df_norm['desc_len_norm'].describe())
print(df_norm['log_price_norm'].describe())
```

Normalization Note:

To ensure features are on a comparable scale, especially for algorithms sensitive to feature magnitude like regression , we applied a log transformation to the price feature to reduce right-skewness. This was followed by min-max normalization to scale it between 0 and 1. Additionally, we normalized the desc_len feature (description length) using the same method. Categorical and binary fields were excluded from normalization as they are better handled by encoding.This improves model stability and convergence.

Perform Binning (1M)

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.

In [ ]:
```
# Equal-frequency Binning on price
# Create price bins (quantile-based) - 4 bins: Q1, Q2, Q3, Q4

# Number of bins
num_bins = 4
# df['price_bin'] = pd.qcut(df['price'], q=4, labels=['Low', 'Medium', 'High', 'Very High'])
df_norm['price_bin'] = pd.cut(df_norm['log_price_norm'], bins=num_bins, labels=['Low', 'Medium', 'High', 'Very High'])

print(df_norm['price_bin'].value_counts().sort_index())
```

In [ ]:
```
# Binning desc_len
# Description length binning into 3 categories
features_to_bin = ['desc_len_norm']
bins=3
for feat in features_to_bin:
    df_norm[f'{feat}_bin'] = pd.cut(df_norm[feat], bins=bins, labels=['Short', 'Medium', 'Long'])

print(df_norm[[f'{feat}_bin']].value_counts().sort_index())
```

Note on Binning

To transform continuous numeric variables into categorical ones, we applied binning techniques that simplify the data distribution and help capture nonlinear relationships:

Equal-frequency (quantile) binning for Price:

The log_price_norm variable was divided into 4 bins labeled 'Low', 'Medium', 'High', and 'Very High'.

This splits the data into roughly equal-sized groups based on price quantiles, allowing the model to learn from discrete price categories rather than raw continuous values.

Binning Description Length (desc_len_norm):

The normalized description length was segmented into 3 categories: 'Short', 'Medium', and 'Long'.

This categorization helps capture the impact of product description verbosity on pricing in a simplified, interpretable form.

Binning is especially useful when:

- Handling skewed distributions,
- Enabling models that benefit from categorical inputs,

Improving interpretability by grouping continuous features into meaningful buckets.

Perform encoding (1M)

In [ ]:
```
# Columns to one-hot encode (categorical bins, small categories)
one_hot_cols = ['cat_1', 'cat_2', 'cat_3', 'price_bin', 'price_bin_custom', 'desc_bin']

# Make sure these columns exist in df_norm
one_hot_cols = [col for col in one_hot_cols if col in df_norm.columns]

# One-hot encode the specified columns in normalized dataframe
df_norm_encoded = pd.get_dummies(df_norm, columns=one_hot_cols, dummy_na=True)

# Label encode brand_name in df_norm_encoded (brand can be many categories)
le_brand = LabelEncoder()
df_norm_encoded['brand_name_le'] = le_brand.fit_transform(df_norm_encoded['brand_name'].astype(str))

# Optionally drop original brand_name to avoid duplication
df_norm_encoded.drop(columns=['brand_name'], inplace=True)

print("After encoding, df_norm_encoded shape:", df_norm_encoded.shape)
print(df_norm_encoded.head())
```

Note on Encoding

One-Hot Encoding:

Selected categorical columns with relatively small numbers of unique categories (including binned features like price and description length) were converted into one-hot encoded vectors. This process creates binary indicator columns for each category level, allowing the model to interpret categorical data numerically without implying any ordinal relationship.

Label Encoding for brand_name:

Since brand_name contains many unique values (high cardinality), one-hot encoding would create a very large sparse matrix. Instead, it was label encoded into integer codes to reduce dimensionality while still distinguishing different brands.

Handling Missing Categories:

The dummy_na=True parameter in get_dummies adds an extra indicator column for missing values, ensuring that missing data in categorical columns is explicitly represented.

Dropping Original Columns:

After encoding, original categorical columns are dropped to avoid duplication and maintain a consistent numeric feature set for modeling.

This combination of encoding techniques balances memory efficiency and model interpretability.

Perform Data Discretization(2M)

In [ ]:
```
# Equal-frequency Discretization on price (quantile-based)
num_bins = 4
df_norm['price_disc'] = pd.qcut(df_norm['log_price_norm'], q=num_bins, labels=['Low', 'Medium', 'High', 'Very High'])

# Equal-frequency Discretization on description length
bins = 3
df_norm['desc_len_disc'] = pd.qcut(df_norm['desc_len'], q=bins, labels=['Short', 'Medium', 'Long'])

# Check distribution
print(df_norm['price_disc'].value_counts().sort_index())
print(df_norm['desc_len_disc'].value_counts().sort_index())
```

Note on Data Discretization

To convert continuous numerical variables into categorical bins for easier interpretation and modeling, we applied **equal-frequency discretization (quantile-based binning)**:

- **Price Binning (`price_disc`)**: The normalized log-transformed price (`log_price_norm`) was divided into **4 bins** representing roughly equal-sized groups labeled **Low**, **Medium**, **High**, and **Very High**. This approach helps capture price tiers and can improve model interpretability when used as categorical features.
- **Description Length Binning (`desc_len_disc`)**: The raw description length (`desc_len`) was split into **3 bins** labeled **Short**, **Medium**, and **Long**, representing the relative verbosity of product descriptions. This categorical transformation enables the model to leverage description size as a meaningful discrete feature.

This ensures that each category contains approximately the same number of samples, which balances the data distribution across bins and reduces bias toward any particular range.

EDA using Visuals(3M)

Use any 3 or more visualisation methods (Boxplot,Scatterplot,histogram,....etc) to perform Exploratory data analysis and briefly give interpretations from each visual.

1. Histogram

In [ ]:
```
plt.figure(figsize=(10, 5))
sns.histplot(df['price'], bins=50, kde=False)
plt.title('Histogram of Price')
plt.xlabel('Price')
plt.ylabel('Count')
plt.xlim(0, 500)  # Limit x-axis to focus on most data
plt.show()

# Log-scale histogram (to visualize skew better)
plt.figure(figsize=(10, 5))
sns.histplot(np.log1p(df['price']), bins=50, kde=True)
plt.title('Histogram of Log(Price + 1)')
plt.xlabel('Log(Price + 1)')
plt.ylabel('Count')
plt.show()
```

Histogram Analysis of Product Prices

To understand the distribution of product prices in the dataset, two histograms were created — one using the raw price values and another using the log-transformed price (`log(price + 1)`).

1. Histogram of Raw Prices

- The distribution of raw prices is **highly right-skewed**.
- **Most products are priced below ₹100**, with a sharp drop in frequency as prices increase.
- There are a **small number of high-priced outliers**, which makes it harder for some models to learn effectively from the data.

2. Histogram of Log(Price + 1)

- After applying **log transformation**, the distribution becomes **much more symmetrical and nearly normal**.
- The transformation reduces the effect of outliers and **compresses the range**, making the data more suitable for linear regression and other statistical models.
- This transformation improves **model stability and performance** when price is used as a target variable.

Conclusion:

Log-transforming the price helps in **handling skewness**, **reducing the influence of outliers**, and is an **essential preprocessing step for regression-based modeling**. This ensures that assumptions like linearity and normality of residuals are better met.

2. Boxplot

In [ ]:
```
plt.figure(figsize=(12, 6))
sns.boxplot(x='cat_1', y='price', data=df_norm)
plt.title('Boxplot of Price by Top-Level Category (cat_1)')
plt.xlabel('Top Category (cat_1)')
plt.ylabel('Price')
plt.ylim(0, df_norm['price'].quantile(0.95))  # Limit y-axis to 95th percentile price
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Boxplot of Price by Category:

The boxplot visualizes the distribution of product prices across different top-level categories (cat_1). Each box represents the interquartile range (IQR) of prices within a category, with the median marked inside the box. Whiskers show the range excluding outliers, and points outside are considered price outliers.

Insights from the plot:

- It highlights how price varies significantly between categories.
- Some categories show higher median prices and wider spread, indicating more variability.
- Outliers reveal extreme price listings, useful for identifying unusual products or pricing errors.
- This visualization aids in understanding category-based price patterns and helps in feature selection or further analysis.

Note

To improve readability and avoid extreme price outliers squishing the plot, the y-axis is limited to the 95th percentile of prices. This focuses the view on the majority of the data, making category-wise price differences clearer.

3. Scatterplot

In [ ]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df_norm, x='desc_len', y='price', alpha=0.4)
plt.title('Scatter Plot: Description Length vs. Price')
plt.xlabel('Description Length (word count)')
plt.ylabel('Price')
plt.xlim(0, 300)  # optional: limit for better focus
plt.ylim(0, 500)  # optional: filter high price outliers
plt.grid(True)
plt.tight_layout()
plt.show()
```

Scatter Plot: Description Length vs. Price

This scatter plot shows the relationship between the length of the item description (desc_len, measured in word count) and the product price. To focus on the bulk of the data and reduce the influence of extreme outliers:

- The x-axis is limited to descriptions up to 300 words.
- The y-axis is limited to prices up to $500.

These limits help visualize the main data trend without being skewed by very long descriptions or very expensive items.

Observations from the plot may include:

- Items with very short descriptions tend to have lower prices.
- There is a slight positive correlation where longer descriptions generally associate with higher prices.
- Price variability increases with longer descriptions, indicating diverse product types or quality.
- This insight justifies including desc_len as a predictive feature in the pricing model.

4. Correlation Heatmap

In [ ]:
```
# Select numeric columns for correlation
numeric_cols = ['price', 'desc_len', 'log_price', 'log_price_norm', 'desc_len_norm', 'shipping', 'item_condition_id']

plt.figure(figsize=(10, 6))
corr_matrix = df_norm[numeric_cols].corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap')
plt.show()
```

Correlation Heatmap of Key Numeric Features

This heatmap visualizes the Pearson correlation coefficients between selected numeric features in the dataset, including:

- price and its log-transformed variants (log_price, log_price_norm)
- Description length (desc_len and normalized desc_len_norm)
- Shipping status (shipping)
- Item condition (item_condition_id)

Key takeaways:

- Strong positive correlations exist between price and its log-transformed versions, confirming the effectiveness of log transformation in stabilizing variance.
- Description length shows a modest positive correlation with price, supporting its predictive value.
- Shipping and item condition have relatively weaker correlations with price, but still provide useful information.
- Normalized features maintain similar relationships, validating the preprocessing steps.

This heatmap aids in understanding feature relationships and guiding feature selection for modeling.

Summary of Visualizations

- **Boxplot**: Highlights distribution differences and potential outliers across categorical groups, helping to compare price variations by category.
- **Histogram**: Reveals the overall data distribution and skewness, providing insight into how features like price or description length are spread.
- **Scatterplot**: Shows relationships and potential trends between two continuous variables, such as description length and price.
- **Correlation Heatmap**: Quantifies the strength and direction of linear relationships between numeric features, guiding feature selection and understanding dependencies.

Feature Selection(2M)

Apply Univariate filters identify top 5 significant features by evaluating each feature independently with respect to the target variable by exploring

1. Mutual Information (Information Gain)
2. Gini index
3. Gain Ratio
4. Chi-Squared test
5. Fisher Score

(From the above 5 you are required to use any **two**)

Feature Preparation and Encoding for Modeling

- **Feature Selection:** Removed text-heavy columns like `item_description` and `name` that are not directly usable in numeric models.
- **Categorical Encoding:** Converted all categorical variables into numeric format using **one-hot encoding** with `drop_first=True` to avoid multicollinearity.
- **Target Variable:** Defined the target variable `y_binned` as the binned (categorical) version of price for classification or analysis purposes.

In [ ]:
```
# Select features (drop text columns like 'item_description', 'name')
features = df_norm.drop(columns=['price', 'item_description', 'name'])

# One-hot encode all categorical columns
features_encoded = pd.get_dummies(features, drop_first=True)

cat_cols = features.select_dtypes(include=['object', 'category']).columns
features[cat_cols] = features[cat_cols].astype(str)  # ensure all are strings
features_encoded = pd.get_dummies(features, columns=cat_cols, drop_first=True)

# Assuming 'price_bin' is your binned target column (categorical)
y_binned = df_norm['price_bin']
```

Chi-Square Feature Selection

- **Purpose:** Chi-Square (χ^2) test is used here to identify the most relevant categorical features that have the strongest association with the binned price target.
- **Process:**
  - Applied `SelectKBest` with `chi2` as the scoring function to select the top 10 features.
  - The test evaluates the dependency between each feature and the target, highlighting features that are statistically significant predictors.
- **Outcome:**
  - Features with the highest Chi-Square scores are selected as the most informative for the price prediction model.
  - This reduces dimensionality, improves model interpretability, and can enhance predictive performance.

In [ ]:
```
# Run Chi2 feature selection (k = number of features to select, e.g., 10)
selector = SelectKBest(score_func=chi2, k=10)

# Fit selector on encoded features and binned target
selector.fit(features_encoded, y_binned)

# Get scores and selected feature names
chi2_scores = selector.scores_
selected_features = features_encoded.columns[selector.get_support()]

# Show results
chi2_results = pd.DataFrame({'Feature': features_encoded.columns, 'Chi2_Score': chi2_scores})
chi2_results = chi2_results.sort_values(by='Chi2_Score', ascending=False)

print(chi2_results.head(10))
print("\nSelected Features:\n", selected_features)
```

Mutual Information Feature Selection

- **Purpose:** Mutual Information (MI) measures the amount of shared information between each feature and the continuous target (`log_price`), capturing both linear and non-linear dependencies.
- **Process:**
  - Computed MI scores using `mutual_info_regression` on one-hot encoded features against the log-transformed price.
  - This method identifies features that provide the most predictive information about price variability.
- **Outcome:**
  - Features with the highest MI scores are considered most informative for predicting product prices.
  - MI helps uncover complex relationships that may not be detected by linear methods like Chi-Square, supporting better feature selection.

In [ ]:
```
# Your continuous target variable (use 'price' or 'log_price' — e.g., log_price for better distribution)
y = df_norm['log_price']

# Make sure features_encoded is your one-hot encoded feature dataframe
```

```
# It should be all numeric and aligned with y
X = features_encoded

# Compute mutual information scores
mi_scores = mutual_info_regression(X, y, discrete_features='auto', random_state=42)

# Create a DataFrame to view results clearly
mi_results = pd.DataFrame({'Feature': X.columns, 'MI_Score': mi_scores})
mi_results = mi_results.sort_values(by='MI_Score', ascending=False)

# Display top features
print(mi_results.head(10))
```

Report observations (2M)

Write your observations from the results of each of the above method(1M). Clearly justify your choice of the method.(1M)

Chi-Square (Chi2) Results:

- Chi2 identified several categorical features with significant associations to the target by measuring dependence between feature categories and binned target values.
- However, since Chi2 requires discrete data and a categorical target, the continuous target variable was discretized (binned) to apply this method.
- This discretization may lead to some loss of information and Chi2 cannot capture relationships with continuous or numerical features well.
- Some important continuous features were not effectively ranked by Chi2 due to this limitation.

Mutual Information Regression (MI) Results:

- MI was able to measure the dependency between all types of features (both continuous and categorical, after encoding) and the continuous target directly.
- It captured both linear and nonlinear relationships without the need to bin the target.
- MI highlighted a broader set of relevant features, including numerical features that Chi2 missed.
- The results provided a more nuanced understanding of which features contribute most to predicting the target variable.

Justification of Method Choice

While Chi-Square is useful for understanding relationships in categorical data, Mutual Information Regression is more suitable for this regression problem because it directly handles continuous targets and captures complex dependencies. Therefore, **MI was chosen as the primary feature selection method** to retain features that have the strongest information content for predicting the continuous target. Chi2 was used additionally to compare and validate the importance of categorical features after discretization.

Correlation Analysis (3 M)

Perform correlation analysis(1M) and plot the visuals(1M).Briefly explain each process,why is it used and interpret the result(1M).

Correlation Calculation:

We calculate Pearson correlation coefficients between numeric features. This measures the strength and direction of a linear relationship between pairs of variables, with values ranging from -1 (perfect negative correlation) to +1 (perfect positive correlation), and 0 indicating no linear correlation.

In [ ]:
```
# 1. Select numeric columns for correlation analysis
numeric_cols = df_norm.select_dtypes(include=['float64', 'int64', 'int8', 'float32']).columns

# For example, consider relevant numeric features only
corr_features = ['price', 'log_price_norm', 'desc_len_norm', 'item_condition_id', 'shipping']

# 2. Calculate correlation matrix
corr_matrix = df_norm[corr_features].corr()

print("Correlation matrix:\n", corr_matrix)
```

Heatmap Visualization:

A heatmap visually represents the correlation matrix using colors. Positive correlations are shown in warm colors (e.g., red), negative correlations in cool colors (e.g., blue). Annotations show exact correlation values, making it easier to interpret relationships at a glance.

In [ ]:
```
# Plot correlation heatmap
plt.figure(figsize=(8,6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap of Selected Numeric Features')
plt.show()
```

Why Correlation Analysis?

- Helps identify redundant features (highly correlated variables) which might be dropped during feature selection.
- Detects multicollinearity issues that could affect regression models.
- Reveals potential linear associations useful for predictive modeling.

Interpretation of Correlation Matrix:

- **Price & Log Price Norm (0.737):** There is a strong positive correlation between `price` and its normalized log-transformed version (`log_price_norm`). This is expected since `log_price_norm` is a transformed representation of `price` designed to reduce skewness.
- **Price & Desc Length Norm (0.058):** The correlation between price and description length (normalized) is very weak and positive. This indicates that longer product descriptions have a very slight tendency to be associated with higher prices, but the effect is minimal.
- **Price & Item Condition (0.005):** Price shows almost no linear relationship with item condition. This suggests item condition might not directly influence price in a linear manner, or the effect is negligible in this dataset.
- **Price & Shipping (-0.111):** There is a weak negative correlation between price and shipping. This suggests items with shipping fees might tend to have slightly lower prices, or cheaper items might have shipping included, but the effect is not very strong.
- **Item Condition & Desc Length (-0.120):** There is a weak negative correlation between item condition and description length, implying that better condition items might have slightly shorter descriptions, but this is a subtle relationship.
- **Shipping & Log Price Norm (-0.233):** There is a mild negative correlation between shipping and the log price, stronger than with raw price, which again suggests that items requiring shipping fees tend to be lower priced on a log scale.

Summary:

- **Strongest relationships:** Between price and log_price_norm (as expected, since one is a transformation of the other).
- **Weak to negligible relationships:** Between price and other features like description length, item condition, and shipping.
- These weak correlations suggest that features other than the ones analyzed here may be important predictors of price, or relationships may be non-linear and require more advanced modeling techniques.

Model Building and Prediction (4M)

Fit a linear regression model using the most important features identified(1M).Plot the visuals(1M).Briefly explain the regression model,equation (1M) and perform one prediction using the same(1M).

Prepare Data for Modeling

- **Objective:** Extract the selected features from the standardized dataframe (df_std) and set the target variable (y). Standardize numeric features to have zero mean and unit variance for better model convergence and performance, while preserving categorical features.
- **Steps:**
  1. **Identify numeric columns** (e.g., IDs, condition, shipping, description length, encoded brand).
  2. **Extract categorical columns** as those not in the numeric list.
  3. Convert categorical columns to string and then to numeric (if one-hot encoded), replacing non-convertible values with zero.
  4. Apply `StandardScaler` to numeric columns to standardize their scale.
  5. Concatenate scaled numeric features and processed categorical features into one final dataframe ready for modeling.
- **Outcome:** A combined, standardized dataset (`df_std`) suitable for feeding into regression or other machine learning models, ensuring all features are numeric and properly scaled.

In [ ]:
```
# 1. Define numeric columns to scale
numeric_cols_to_scale = ['train_id', 'item_condition_id', 'shipping', 'desc_len', 'desc_len_norm', 'brand_name_le']

# 2. Identify categorical columns as all other columns not in numeric_cols_to_scale
cat_cols = df_norm_encoded.columns.difference(numeric_cols_to_scale)

# 3. Convert categorical columns to string type to avoid categorical dtype issues
df_cat = df_norm_encoded[cat_cols].astype(str).fillna('0')

# If these are one-hot encoded columns, convert them to numeric
# Try to convert to numeric, non-convertible become NaN
df_cat = df_cat.apply(pd.to_numeric, errors='coerce').fillna(0)

# 4. Scale numeric features using StandardScaler
```

```
scaler = StandardScaler()
df_num_scaled = pd.DataFrame(
    scaler.fit_transform(df_norm_encoded[numeric_cols_to_scale]),
    columns=numeric_cols_to_scale,
    index=df_norm_encoded.index
)

# 5. Combine scaled numeric and categorical features into one dataframe
df_std = pd.concat([df_num_scaled, df_cat], axis=1)

# 6. Display shape and preview
print(f"Combined standardized dataframe shape: {df_std.shape}")
print(df_std.head())
```

Selection of Most Important Features

- Based on feature selection techniques (Chi-square and Mutual Information), the following key features were identified as most relevant for predicting price:
  - `desc_len` (Description length)
  - `shipping` (Shipping option)
  - `brand_name_le` (Encoded brand name)
  - `item_condition_id` (Product condition)
  - `log_price_norm` (Normalized log-transformed price — target or reference)
- These features capture important aspects such as product description detail, shipping method, brand influence, item condition, and target scaling.
- Using this subset simplifies the model, improves interpretability, and reduces overfitting risk while maintaining predictive power.

In [ ]:
```
# Select the Most Important Features
selected_features = [
    'desc_len',
    'shipping',
    'brand_name_le',
    'item_condition_id',
    'log_price_norm'
]
```

Preparing Data for Regression

- **Target Variable (`y`)**: The actual product price from the standardized dataframe.
- **Feature Matrix (`X`)**: The selected important features extracted from the standardized dataframe.
- **Train-Test Split**: The dataset is split into training (80%) and testing (20%) sets to evaluate model performance on unseen data, ensuring generalization.
- **Random State**: Set for reproducibility of the split.

In [ ]:
```
# Target variable
y = df_std['price']

# Feature matrix
X = df_std[selected_features]

# Split data: 80% training, 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Training and Evaluating the Linear Regression Model

- **Model Initialization and Training:** A `LinearRegression` model is instantiated and trained on the training data (`X_train`, `y_train`), learning the linear relationships between selected features and the target price.
- **Prediction:** The trained model predicts prices on the test dataset (`X_test`), generating `y_pred`.
- **Evaluation Metrics:**
  - **Mean Squared Error (MSE):** Measures the average squared difference between actual and predicted prices; lower values indicate better fit.
  - **R^2 Score:** Indicates the proportion of variance in the target variable explained by the model; values closer to 1 imply better predictive power.

In [ ]:
```
# Initialize and train the model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
```

```
# Predict on test set
y_pred = lr_model.predict(X_test)

# Evaluation metrics
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R² Score: {r2:.2f}")
```

Actual vs Predicted Prices Plot

- **Scatter Plot:** Visualizes how well predicted prices (`y_pred`) align with the actual prices (`y_test`). Each point represents one test sample.
- **Ideal Fit Line (y = x):** The dashed red line shows perfect predictions where predicted price equals actual price. Points close to this line indicate accurate predictions.
- **Interpretation:**
  - Points tightly clustered around the red line suggest the model predicts prices well.
  - Deviations highlight prediction errors or areas where the model struggles.
- **Purpose:** This visualization helps quickly assess the model's accuracy and spot any systematic over- or under-predictions.

In [ ]:
```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred, alpha=0.5)

# Plot ideal prediction line (y = x)
lims = [min(y_test.min(), y_pred.min()), max(y_test.max(), y_pred.max())]
plt.plot(lims, lims, '--r', label='Ideal Fit (y = x)')

plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices (Linear Regression)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Explanation of Actual vs Predicted Plot

- The **red dashed line (y = x)** represents the ideal scenario where predicted prices exactly match the actual prices.
- **Points above the line** indicate **underestimation** — the model predicted a lower price than actual.
- **Points below the line** indicate **overestimation** — the model predicted a higher price than actual.
- This plot provides a straightforward visual to assess prediction accuracy and bias.
- Adding this reference line greatly enhances interpretability in regression evaluation.

Explanation of Linear Regression and Model Equation

Linear Regression is a fundamental supervised learning algorithm used for predicting a continuous target variable—in this case, the product price. The goal of linear regression is to model the relationship between one or more explanatory variables (features) and the target by fitting a linear equation to observed data. The general form of the linear regression model is:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \varepsilon$$

where:

- $(\hat{y})$

  is the **predicted value** (in this case, the price).
- (β_0)

  is the **intercept** — the expected value of y when all input features are zero.
- $(\beta_1, \beta_2, \ldots, \beta_n)$

  are the **coefficients** associated with each feature

  $(x_1, x_2, \ldots, x_n)$
- (ε)

  is the **error term**, representing the difference between the observed and predicted values.

For our case, the model can be written as:

$$\text{Price} = \beta_0 + \beta_1 \cdot \text{desc\_len} + \beta_2 \cdot \text{shipping} + \beta_3 \cdot \text{brand\_name\_le} + \beta_4 \cdot \text{item\_condition\_id} + \beta_5 \cdot \text{log\_price\_norm} + \varepsilon$$

This linear model assumes that the relationship between each feature and the target variable is additive and linear. The coefficients (β) are learned from the training data to minimize the prediction error.

Sample Prediction Demonstration

This code selects a single random sample from the test dataset and uses the trained linear regression model to predict its price. It then prints both the input feature values for this sample and compares the predicted price with the actual price. This helps illustrate how the model performs on individual data points and provides a tangible example of its predictive ability.

In [ ]:
```
# Pick a random sample from test set
sample = X_test.iloc[[0]]
predicted_price = lr_model.predict(sample)

print("Sample input:")
print(sample)
print(f"\nPredicted Price: ${predicted_price[0]:.2f}")
print(f"Actual Price: ${y_test.iloc[0]:.2f}")
```

Observations and Conclusions(1M)

Observations:

1. Model Fit:

- The regression model was trained on standardized features, ensuring comparability across all input variables.
- Evaluation using an **Actual vs Predicted scatter plot** shows most points lie close to the ideal line (`y = x`), suggesting a reasonably good fit.

2. Error Trend:

- Some scatter is visible, especially at higher price values, indicating the model has **slightly more error for expensive products**.
- Lower-priced items are predicted with higher accuracy.

3. Feature Influence:

- Key features like `log_price_norm` , `shipping` , `desc_len` , and `brand_name_le` had high MI scores, validating their strong influence on the target variable.

Conclusions:

- **Linear Regression** provides a **straightforward and interpretable model**, which worked fairly well on the selected features.
- The presence of minor deviations suggests that the model might benefit from more complex algorithms (e.g., decision trees or ensemble methods) for further accuracy.
- Overall, for the scope of this analysis, linear regression serves as an effective baseline model, highlighting the key factors that impact product price on the platform.

Solution (1M)

What is the solution that is proposed to solve the business problem discussed in the beginning. Also share your learnings while working through solving the problem in terms of challenges, observations, decisions made etc.

Proposed Solution to the Business Problem

The business objective was to **predict product prices** based on listing features such as product descriptions, item condition, category hierarchy, shipping information, and brand metadata. Accurate price prediction can improve user experience, support dynamic pricing strategies, and assist in inventory decisions.

To address this, a **Linear Regression model** was developed using a **sample of 10,000 rows** from the Mercari dataset to ensure processing efficiency and memory feasibility. The following steps were undertaken:

Data Preprocessing

- Cleaned and filtered the data (no duplicates were present).
- Filled missing values with `"Unknown"` for categorical variables and median for numerics.
- Standardized string formats (lowercased, stripped).
- Cleaned text in `item_description` and engineered new features such as `desc_len` .
- Normalized and log-transformed `price` to reduce skewness.

- Split hierarchical categories (`category_name`) into three levels: `cat_1` , `cat_2` , and `cat_3` .

Feature Engineering & Selection

- Applied **equal-frequency binning** to discretize `price` and `desc_len` .
- One-hot encoded selected categorical variables (e.g., binned categories, category splits).
- Used **Chi-Square (χ^2)** for categorical feature selection and **Mutual Information (MI)** for both categorical and numeric variables.
- To avoid crashes from memory exhaustion, MI was computed on the **10,000-sample subset** only.

Data Transformation

- Standardized selected numeric features using `StandardScaler` .
- Combined scaled numerical features with processed categorical features to create a model-ready dataframe.

Modeling

- Built a **Linear Regression model** using the top features selected (e.g., `desc_len` , `shipping` , `brand_name_le` , `item_condition_id` , `log_price_norm`).
- Split data into 80% training and 20% testing for validation.
- Trained the model and evaluated performance using **MSE** and **R^2 Score**.
- Visualized **actual vs. predicted prices** with scatter plots and an ideal prediction line (`y = x`).

Evaluation & Prediction

- Achieved a baseline regression model that predicts price with acceptable error and interpretability.
- Ran **individual sample predictions** to demonstrate real-time usability of the model.

Learnings and Reflections

Challenges Faced

1. **High Dimensionality & Memory Constraints**
   - One-hot encoding and large data volume led to **MemoryErrors**.
   - Scaling operations converted sparse data to dense `float64` , overwhelming system memory.
2. **Mutual Information Crash**
   - MI regression failed on the full dataset due to internal transformations.
   - **Workaround**: Reduced data to a 10,000-row subset to complete MI analysis.
3. **Data Cleaning Complexity**
   - Required type-casting of categories and handling placeholder text.
   - Addressed inconsistencies in price, shipping, and categorical splits.
   - Applied transformations carefully to avoid dtype conflicts during encoding or scaling.
4. **Skewed Target Variable**
   - Raw prices were heavily right-skewed; applied **log transformation** to stabilize distribution for regression.

Key Observations

- **Shipping**, **item condition**, and **brand_name** were influential in predicting price.
- **Log-transformed prices** improved regression performance and interpretability.
- **Chi-Square** was effective for categorical filtering, while **Mutual Information** captured non-linear dependencies.
- Visualizations like **boxplots**, **scatter plots**, and **correlation heatmaps** revealed insightful data relationships.

Decisions Made

- Performed **sampling** (10,000 rows) to avoid crashes while preserving feature diversity.
- Used **standardization** post feature selection to ensure consistent scaling.
- Excluded high-cardinality or uninformative fields (e.g., `train_id` , `item_description`).
- Chose **Linear Regression** due to project constraints and the requirement for interpretability, while leaving scope for future modeling enhancements.

Final Outcome

A lightweight yet interpretable **Linear Regression model** was developed and tested successfully. It allows the business to:

- Offer **data-driven price suggestions** to sellers.
- **Detect and flag pricing anomalies**.
- **Automate price recommendations** for new product listings.

This model sets a strong foundation for future enhancements such as advanced tree-based models (e.g., Random Forest, XGBoost) or deep learning pipelines, should computational resources allow.

In [ ]: