**Project: Ticket Classification and Analysis**

**Objective**: To process customer support ticket data, perform feature engineering, and train models to classify ticket issue types and urgency levels.

**Section: Data loading**

* **Markdown Cell**: Introduces the subtask: loading data from an Excel file into a pandas DataFrame.
* **Code Cell**:
  + Imports pandas as pd.
  + Attempts to read ai\_dev\_assignment\_tickets\_complex\_1000.xls into a DataFrame df.
  + Displays the first 5 rows of df using df.head().
  + Includes try-except blocks to handle FileNotFoundError (prints an error message and sets df to None) and any other Exception (prints the error and sets df to None).
  + **Output**: Shows a table with columns: ticket\_id, ticket\_text, issue\_type, urgency\_level, product.

**Section: Data cleaning**

* **Markdown Cell**: Introduces the subtask: cleaning the ticket\_text column.
* **Code Cell**:
  + Imports re (for regular expressions).
  + df['ticket\_text'] = df['ticket\_text'].str.lower(): Converts all text in ticket\_text to lowercase for consistency.
  + df['ticket\_text'] = df['ticket\_text'].apply(lambda x: re.sub(r'[^a-zA-Z0-9\\s]', '', str(x))): Removes special characters (anything not alphanumeric or whitespace) from ticket\_text.
  + df['ticket\_text'] = df['ticket\_text'].fillna(''): Fills any missing values (NaN) in ticket\_text with an empty string. Justification: to preserve other data in rows with missing text.
  + df.drop\_duplicates(subset=['ticket\_text'], keep='first', inplace=True): Removes rows that have duplicate ticket\_text, keeping the first occurrence. inplace=True modifies the DataFrame directly. Justification: avoids data redundancy, assumes the first record is most reliable.
  + df['urgency\_level'] = df['urgency\_level'].str.lower(): Converts urgency\_level values to lowercase.
  + df['urgency\_level'] = df['urgency\_level'].fillna('unknown'): Fills missing urgency\_level values with the string 'unknown'. Justification: 'unknown' is more informative than dropping or guessing.
  + display(df.head()): Shows the first 5 rows of the cleaned DataFrame.
  + print(df.info()): Prints a summary of the DataFrame, including data types and non-null counts for each column.
  + **Output**: Shows the cleaned DataFrame head and info. issue\_type still has some nulls.

**Section: Data preparation**

* **Markdown Cell**: Introduces the subtask: preparing data for feature engineering and modeling.
* **Code Cell**:
  + import nltk
  + nltk.download('punkt'): Downloads the 'punkt' tokenizer models from NLTK if not already present.
  + **Output**: Shows download status.
* **Markdown Cell**: Provides reasoning for re-executing text preprocessing and label encoding after ensuring 'punkt' is available.
* **Code Cell**:
  + import nltk
  + nltk.download('punkt\_tab'): Downloads the 'punkt\_tab' resource. (Note: This seems to be an alternative or specific version of punkt. The primary 'punkt' download should usually suffice for word\_tokenize.)
  + import re
  + from nltk.corpus import stopwords
  + from nltk.stem import WordNetLemmatizer
  + from sklearn.preprocessing import LabelEncoder
  + stop\_words = set(stopwords.words('english')): Loads the set of English stop words.
  + lemmatizer = WordNetLemmatizer(): Initializes the lemmatizer.
  + def preprocess\_text(text):
    - tokens = nltk.word\_tokenize(text): Tokenizes the input text.
    - processed\_tokens = [...]: Creates a list of tokens that are lemmatized, not in stop\_words (case-insensitive check), and are purely alphabetic (using re.match(r'[a-zA-Z]+', token)).
    - return " ".join(processed\_tokens): Joins the processed tokens back into a single string.
  + df['processed\_text'] = df['ticket\_text'].apply(preprocess\_text): Applies this preprocessing function to create a new processed\_text column.
  + le\_issue\_type = LabelEncoder() and le\_urgency\_level = LabelEncoder(): Initializes label encoders for the target variables.
  + df['encoded\_issue\_type'] = le\_issue\_type.fit\_transform(df['issue\_type']): Encodes issue\_type into numerical labels. fit\_transform learns the mapping and applies it.
  + df['encoded\_urgency\_level'] = le\_urgency\_level.fit\_transform(df['urgency\_level']): Encodes urgency\_level into numerical labels.
  + display(df.head()): Shows the DataFrame head with new columns.
  + print(df.info()): Shows updated DataFrame info.
  + **Output**: Shows download status for 'punkt\_tab', the DataFrame head with processed\_text, encoded\_issue\_type, and encoded\_urgency\_level columns, and the DataFrame info.

**Section: Feature engineering**

* **Markdown Cell**: Introduces the subtask: creating features from preprocessed text data.
* **Code Cell**:
  + import nltk
  + nltk.download('vader\_lexicon'): Downloads the VADER lexicon for sentiment analysis.
  + **Output**: Shows download status.
* **Code Cell**:
  + from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer
  + from nltk.sentiment.vader import SentimentIntensityAnalyzer
  + import numpy as np
  + bow\_vectorizer = CountVectorizer(max\_features=1000, ngram\_range=(1, 2)): Initializes CountVectorizer to create Bag-of-Words features. It will consider the 1000 most frequent words/n-grams (both unigrams and bigrams).
  + bow\_features = bow\_vectorizer.fit\_transform(df['processed\_text']): Creates the BoW feature matrix.
  + tfidf\_vectorizer = TfidfVectorizer(max\_features=1000, ngram\_range=(1, 2)): Initializes TfidfVectorizer. Parameters are similar to CountVectorizer.
  + tfidf\_features = tfidf\_vectorizer.fit\_transform(df['processed\_text']): Creates the TF-IDF feature matrix.
  + df['ticket\_length'] = df['processed\_text'].apply(lambda x: len(x.split())): Calculates the number of words in each processed ticket text and stores it in ticket\_length.
  + analyzer = SentimentIntensityAnalyzer(): Initializes the VADER sentiment analyzer.
  + df['sentiment\_score'] = df['processed\_text'].apply(lambda x: analyzer.polarity\_scores(x)['compound']): Calculates the compound sentiment score (a normalized score between -1 and 1) for each processed ticket text.
  + feature\_matrix = np.hstack([...]): Horizontally stacks the BoW features (converted to an array), TF-IDF features (converted to an array), and the ticket\_length and sentiment\_score columns to create the final feature\_matrix.
  + print(f"Shape of the feature matrix: {feature\_matrix.shape}"): Prints the dimensions of the feature matrix.
  + **Output**: Shows the shape of the feature matrix (e.g., (710, 1130) meaning 710 samples and 1130 features).

**Section: Data splitting**

* **Markdown Cell**: Introduces the subtask: splitting data into training and testing sets.
* **Code Cell**:
  + from sklearn.model\_selection import train\_test\_split
  + X = feature\_matrix: Assigns the engineered features to X.
  + y\_issue = df['encoded\_issue\_type']: Assigns the encoded issue types to y\_issue.
  + y\_urgency = df['encoded\_urgency\_level']: Assigns the encoded urgency levels to y\_urgency.
  + X\_train, X\_test, y\_train\_issue, y\_test\_issue, y\_train\_urgency, y\_test\_urgency = train\_test\_split(...): Splits the data.
    - test\_size=0.2: 20% of the data for testing.
    - random\_state=42: For reproducible splits.
    - stratify=y\_issue: Ensures that the proportion of each issue type is similar in both training and testing sets.
  + Prints the shapes of X\_train, X\_test, y\_train\_issue, y\_test\_issue, y\_train\_urgency, y\_test\_urgency.
  + **Output**: Shows the shapes of the created datasets.

**Section: Model training**

* **Markdown Cell**: Introduces the subtask: training two separate classification models.
* **Markdown Cell**: Repeats the reasoning for training two separate models.
* **Code Cell**:
  + issue\_type\_model = LogisticRegression(random\_state=42, max\_iter=1000): Initializes a Logistic Regression model for issue type classification. max\_iter is increased to 1000 to help with convergence.
  + urgency\_level\_model = SVC(random\_state=42): Initializes a Support Vector Classifier model for urgency level classification.
  + issue\_type\_model.fit(X\_train, y\_train\_issue): Trains the issue type model.
  + urgency\_level\_model.fit(X\_train, y\_train\_urgency): Trains the urgency level model.
  + **Output**: Shows the SVC model object after training.

**Section: (Model Evaluation)**

* **Markdown Cell**: "Reasoning: Evaluate the performance of the trained models using the test set and calculate the evaluation metrics."
* **Code Cell**:
  + from sklearn.metrics import classification\_report, accuracy\_score
  + from sklearn.linear\_model import LogisticRegression
  + from sklearn.svm import SVC
  + Re-initializes issue\_type\_model and urgency\_level\_model (as in the training cell).
  + Re-trains both models using fit (as in the training cell).
  + y\_pred\_issue = issue\_type\_model.predict(X\_test): Predicts issue types on the test set.
  + y\_pred\_urgency = urgency\_level\_model.predict(X\_test): Predicts urgency levels on the test set.
  + print("Issue Type Classifier Evaluation:"):
  + print(classification\_report(y\_test\_issue, y\_pred\_issue)): Prints precision, recall, F1-score, and support for each class of issue type.
  + print("Accuracy:", accuracy\_score(y\_test\_issue, y\_pred\_issue)): Prints the overall accuracy for issue type classification.
  + print("\\nUrgency Level Classifier Evaluation:"):
  + print(classification\_report(y\_test\_urgency, y\_pred\_urgency)): Prints evaluation metrics for urgency level.
  + print("Accuracy:", accuracy\_score(y\_test\_urgency, y\_pred\_urgency)): Prints overall accuracy for urgency level.
  + **Output**:
    - For Issue Type: Accuracy around 87.3%. Classification report shows varying performance across different issue types. Class 7 (likely corresponding to one of the issue types) has 0 precision, recall, and F1-score, indicating the model failed to predict it correctly or it wasn't present in predictions.
    - For Urgency Level: Accuracy around 31.7%. The classification report shows poor performance, particularly for class 3. An UndefinedMetricWarning is present, indicating issues with precision calculation likely due to no predicted samples for some labels.

**Section: Data exploration**

* **Markdown Cell**: Introduces the subtask: exploring the 'product' column, developing a strategy for product name extraction, and identifying potential complaint keywords.
* **Markdown Cell**: "Reasoning: Analyze the 'product' column to understand its unique values and their frequencies. Then, visualize this distribution using a bar chart. This will inform the strategy for product name extraction."
* **Code Cell**:
  + import matplotlib.pyplot as plt
  + product\_counts = df['product'].value\_counts(): Counts the occurrences of each unique product name.
  + print(product\_counts): Prints the product counts.
  + plt.figure(figsize=(12, 6)): Sets the figure size for the plot.
  + product\_counts.plot(kind='bar'): Creates a bar chart of product counts.
  + plt.title(...), plt.xlabel(...), plt.ylabel(...): Sets plot labels.
  + plt.xticks(rotation=45, ha='right'): Rotates x-axis labels for better readability.
  + plt.tight\_layout(): Adjusts plot to prevent labels from overlapping.
  + plt.show(): Displays the plot.
  + complaint\_keywords = [...]: Manually defines a list of potential complaint keywords. The comment suggests this is an example and might need a more advanced approach.
  + print("\\nPotential complaint keywords:", complaint\_keywords)
  + **Output**: Prints the list of products and their counts (e.g., "Vision LED TV": 83, "RoboChef Blender": 75). Displays a bar chart visualizing these counts. Prints the example list of complaint keywords.