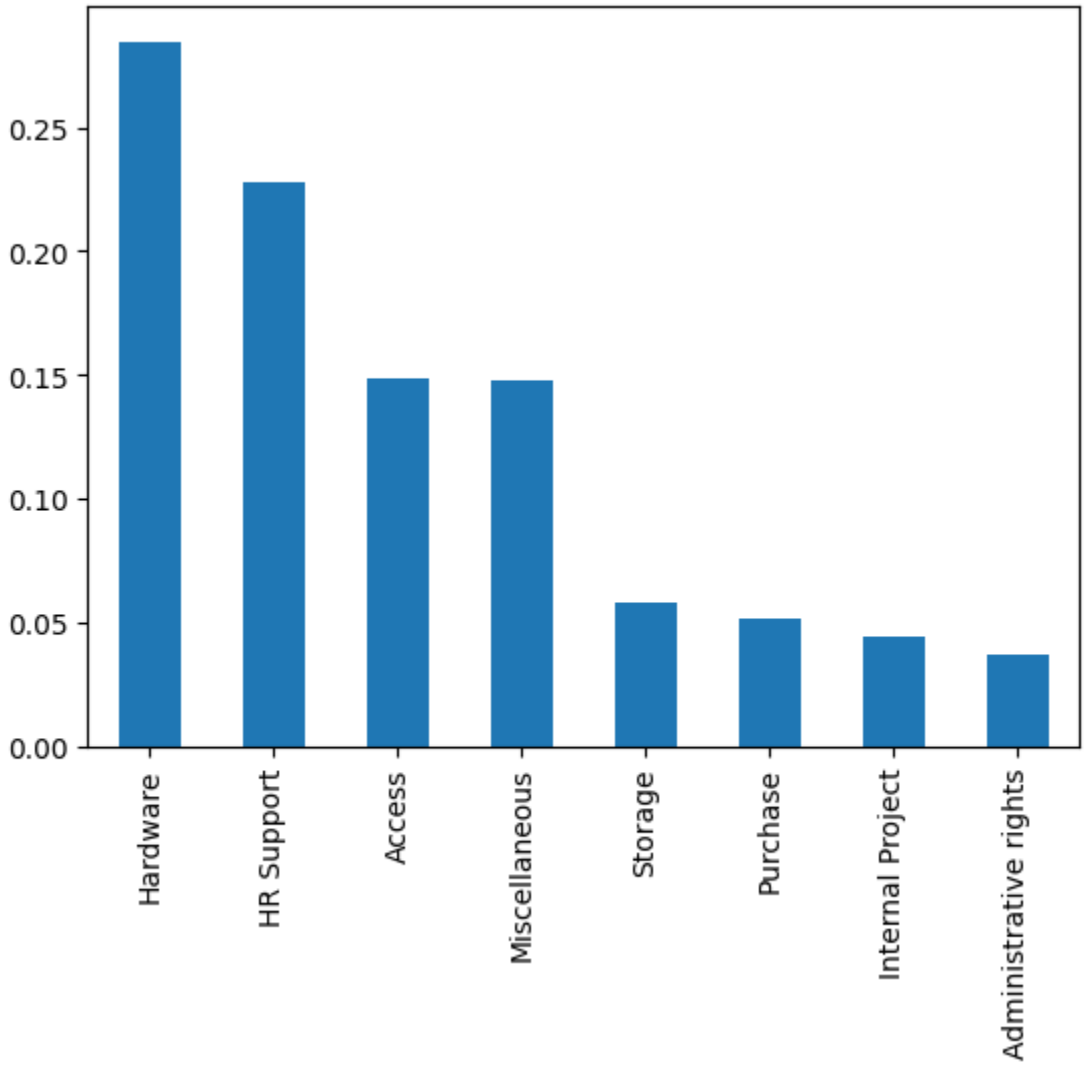
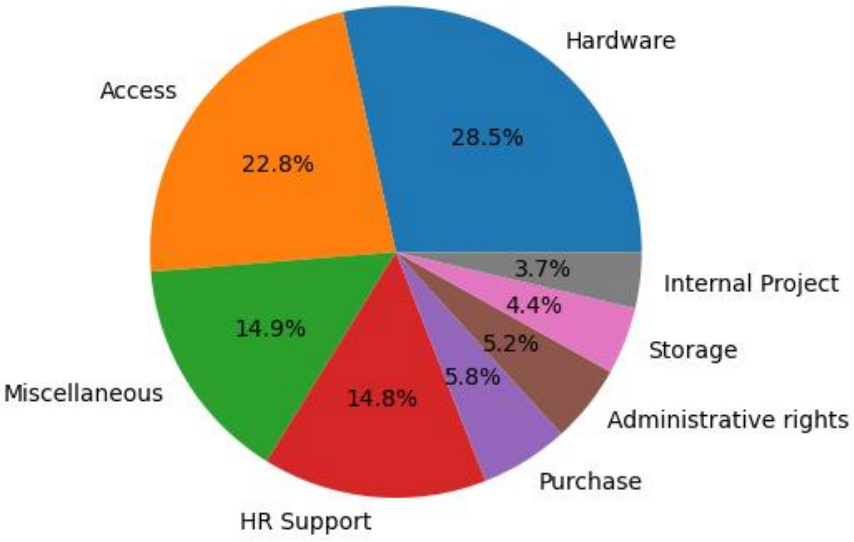
1. Exploratory Data Analysis (EDA)

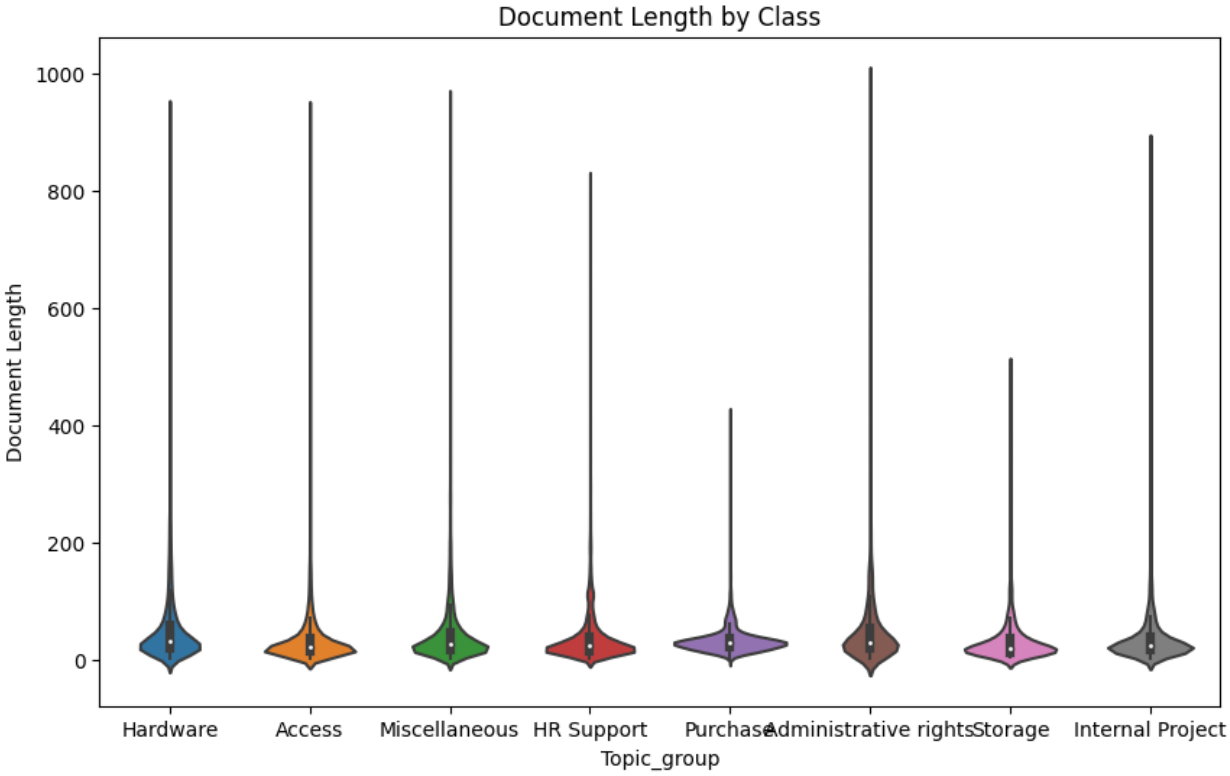
- **Class Frequency Visualization**
    i. Visualized the frequency of each class using bar plots. This helped identify that the dataset is imbalanced.



ii. Also this is a pie chart for classes distribution in the dataset



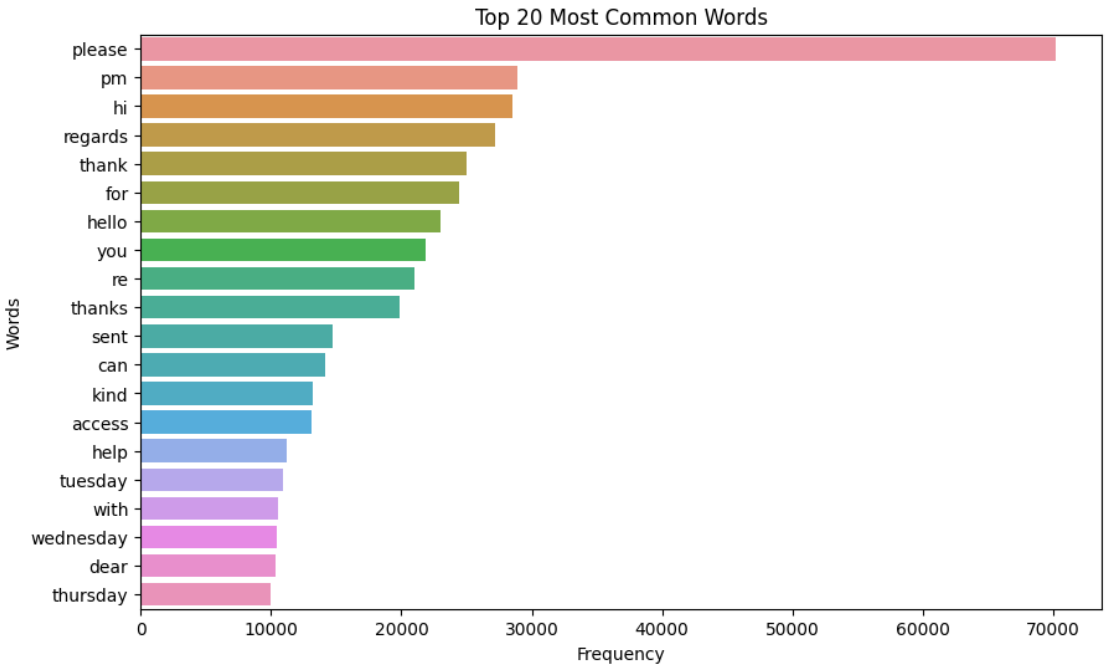
iii. Distribution of document lengths across different classes.



Word Cloud Generation

- To visualize the most frequent words in the dataset. This provided a quick visual insight into the prominent terms used in different ticket categories.

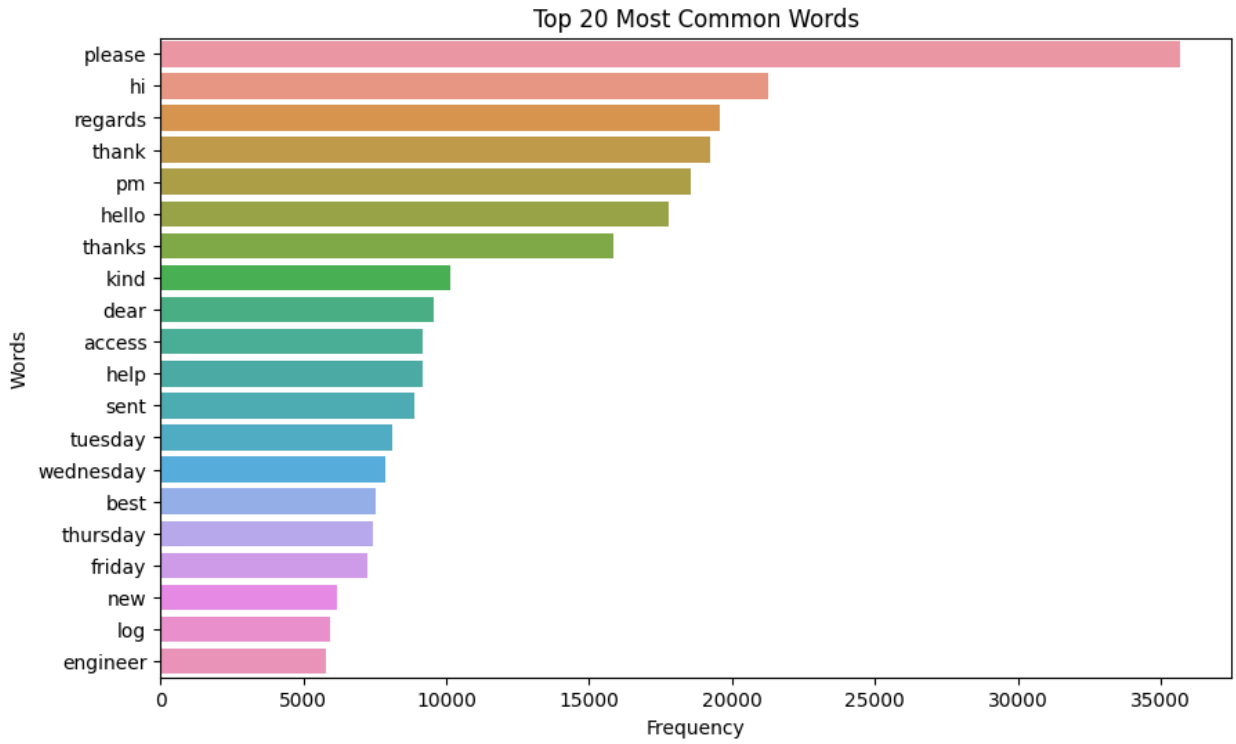
- Also the top 20 words in the dataset.



Preprocessing steps

1. Convert Text to Lowercase
2. Tokenization
3. Stop Words Removal
4. Stemming (Optional)
5. Removing Duplicates
6. Rejoining Tokens

This is the top 20 words after preprocessing



Feature Extraction Using TF-IDF

The TF-IDF (term frequency-inverse document frequency) method was employed to convert the preprocessed text data into numerical features suitable for machine learning models.

- **Term Frequency (TF)**: The number of times a term appears in a document.
- **Inverse Document Frequency (IDF)**: A measure of how much information the word provides, based on its frequency across all documents.

- **Model Selection**:
    - **Support Vector Machine (SVM)**: Chosen for its effectiveness in high-dimensional spaces and its robustness to overfitting.
    - **Logistic Regression**: Selected for its simplicity and efficiency, especially for large datasets.
    - **Random Forest**: Used for its ability to handle large datasets with higher dimensionality and its robustness to overfitting.

- **F1 Score (Macro)**: Selected as the evaluation metric due to the imbalanced nature of the dataset. The macro F1 score calculates the F1 score for each class independently and then takes the average, ensuring that all classes are equally weighted.
    - **Value**: 0.85

Deployment

1. Save the trained model using a serialization library such as joblib or pickle
2. Create app that serve as api endpoint (streamlit)
3. Use a platform like Heroku, AWS, or Streamlit Sharing to deploy the app.
4. Allow users to interact with the model and get predictions.

LLM Challenges

1. Difficulty in handling multi-class classifications

- They often struggle with multi-class classification, especially when there are many possible categories since they weren't trained specifically to map text inputs directly to a set of predefined categories.
- To solve this problem we fine-tune a smaller language model on the specific classes or use few-shot examples that differentiate between categories.

2. Prompt Length Limitations

- Prompting large models with many few-shot examples to improve classification accuracy can lead to excessive prompt lengths. Most LLMs have token limits, and too-long prompts may be truncated.
- Break prompts into smaller parts or reduce the number of examples per prompt. Consider prompt engineering to prioritize concise, informative examples.

3. Variability in Responses

- LLM responses can be inconsistent across runs, especially when prompted with ambiguous or similar categories.
- Use deterministic settings (e.g., low temperature) to reduce variability in the output. Another approach is to prompt the model multiple times and aggregate the results (e.g., using a majority voting mechanism).

4. Overfitting on Few-Shot Examples

- Relying heavily on few-shot examples may cause the model to overfit to those specific examples rather than generalizing across all classes in the ticket data.
- Rotate few-shot examples to cover diverse cases and avoid redundancy. Testing the LLM on a separate validation set can help ensure that it's generalizing well.

5. Difficulty in Scaling with Large Datasets

- Prompting LLMs is compute-intensive, and when handling a large volume of tickets, processing costs can quickly add up.
- leverage the LLM as a classifier for only ambiguous cases, while a smaller and faster classifier handles most cases.