CUSTOMER QUERY CATEGORIZATION USING DISTILBERT

Project Report

by

Nikhil Gupta

Github - Query Categorization using DistilBERT

1. INTRODUCTION

1.1 Objective

The objective of this project is to automate the categorization of customer support queries by leveraging Natural Language Processing (NLP) techniques. Specifically, the aim is to develop a classification model capable of accurately predicting the appropriate departmental queue for each customer query based on its content. Automating this process helps improve the efficiency of customer service operations by ensuring that queries are routed to the correct team without manual intervention.

2. DATASET

2.1 Dataset Description

The dataset used in this project consists of 11,923 customer support query records. Each record contains:

- Subject: A brief summary of the customer's issue.
- Body: A more detailed description of the query.
- Queue: The target variable, representing the department to which the query should be routed.

There are five unique queue categories:

- Billing Support
- Customer Service
- Product Support
- Sales & HR
- Technical Support

The dataset also contains additional metadata fields such as language, tags, and answer status, which are not directly relevant to the queue classification task.

2.2 Data Loading and Initial Inspection

The dataset was loaded using pandas, and initial inspection was conducted using .info() to check the structure, types, and null values. A check was also performed to identify entries where both the Subject and Body were missing.

```
df.info()
```

```
missing_both = df[(df['subject'].isnull()) &
(df['body'].isnull())].shape[0]
print(f"Rows with both 'subject' and 'body' missing:
{missing_both}")
```

No records had both fields missing, so no rows needed to be removed due to missing critical information.

2.3 Data Cleaning

The following steps were undertaken for data cleaning:

- **Irrelevant columns removed:** Non-informative or unused columns such as language, type, and tag-related fields were dropped.
- **Missing values handled:** Missing values in the Subject and Body columns were filled with empty strings.
- **Text cleaned:** HTML tags, URLs, emails, phone numbers, special characters, and non-ASCII characters were removed.
- **Text normalized:** All text converted to lowercase, and excessive whitespace was removed.

Unnecessary metadata was removed to streamline the dataset for NLP modeling.

2.4 Duplicate Handling

Duplicate entries in the dataset were checked and removed.

No duplicate entries were found; no rows were dropped at this stage.

2.5 Feature Engineering

To assist in exploratory analysis and future modeling, the following features were created:

- subject_length: Number of characters in the subject line.

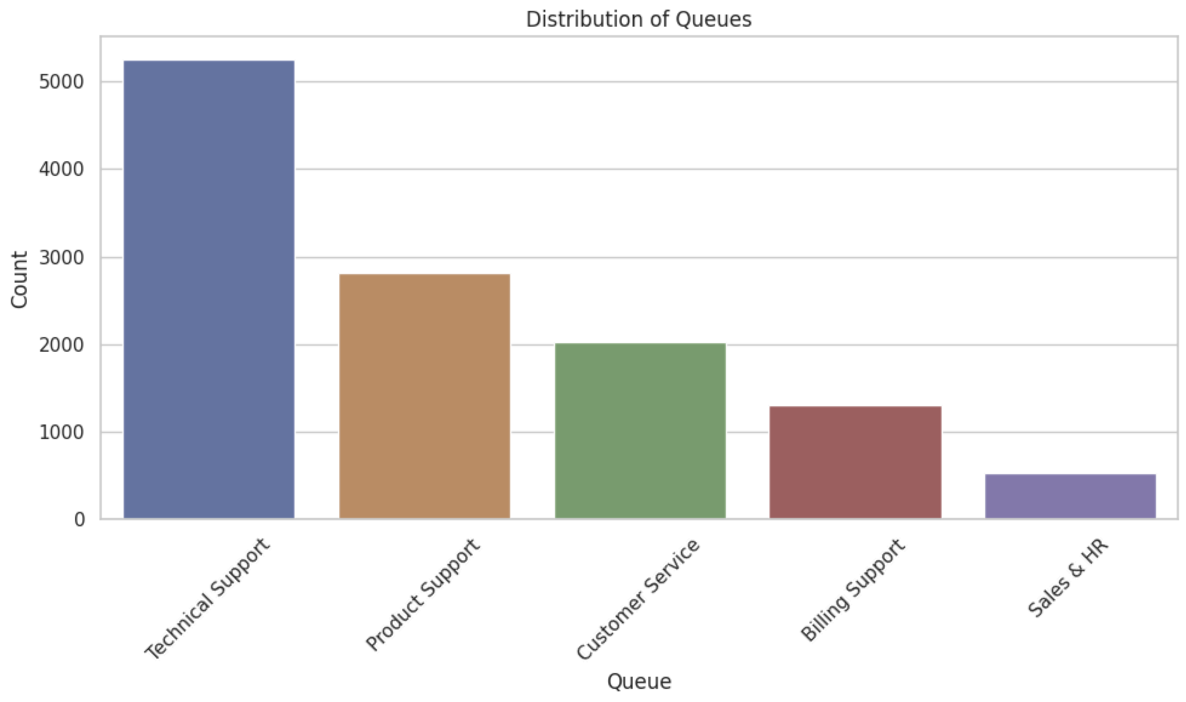
- text_length: Number of characters in the combined subject and body text.

2.6 Exploratory Data Analysis (EDA)

EDA was conducted to understand the distribution and structure of the data using Seaborn, Matplotlib, and WordCloud.

2.6.1 Queue Distribution

A bar plot was used to visualize the distribution of queries across the five queue categories.



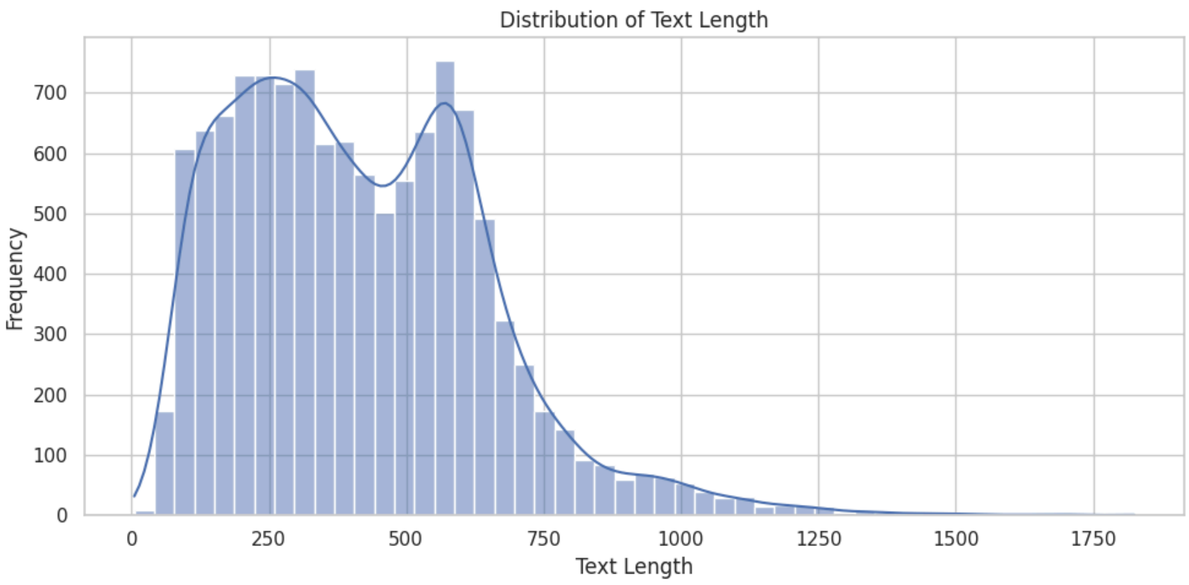
The distribution of queues is imbalanced, with Technical Support and Product Support departments receiving a significantly higher number of queries. Billing Support and Sales & HR departments have less data compared to other departments.

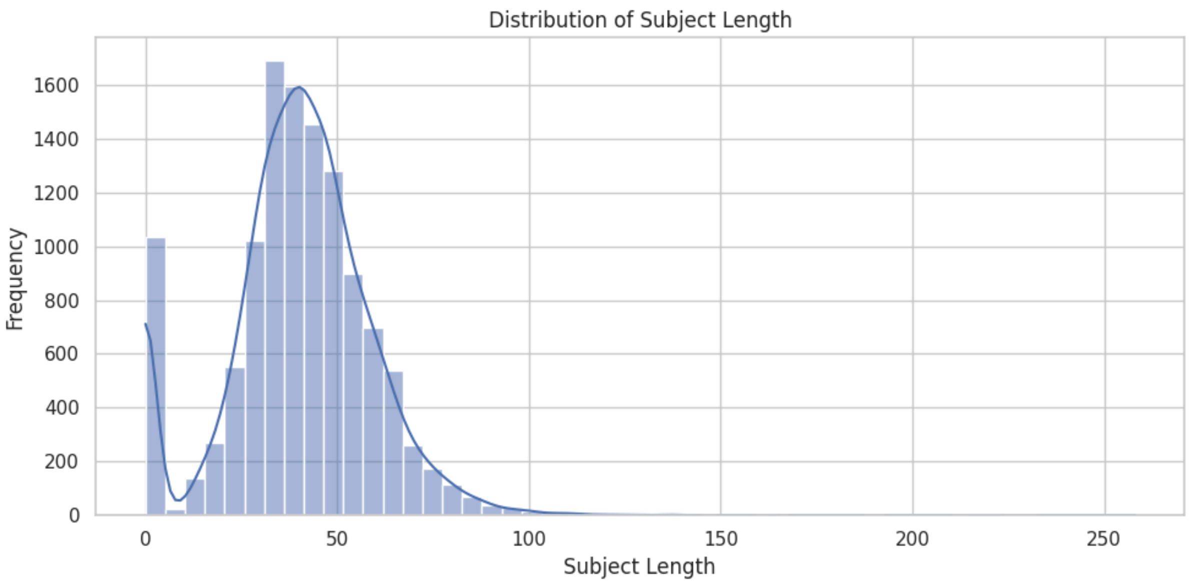
2.6.2 Text Length Distribution

Histograms were plotted for:

- *text_length* (subject + body)

- *subject_length*

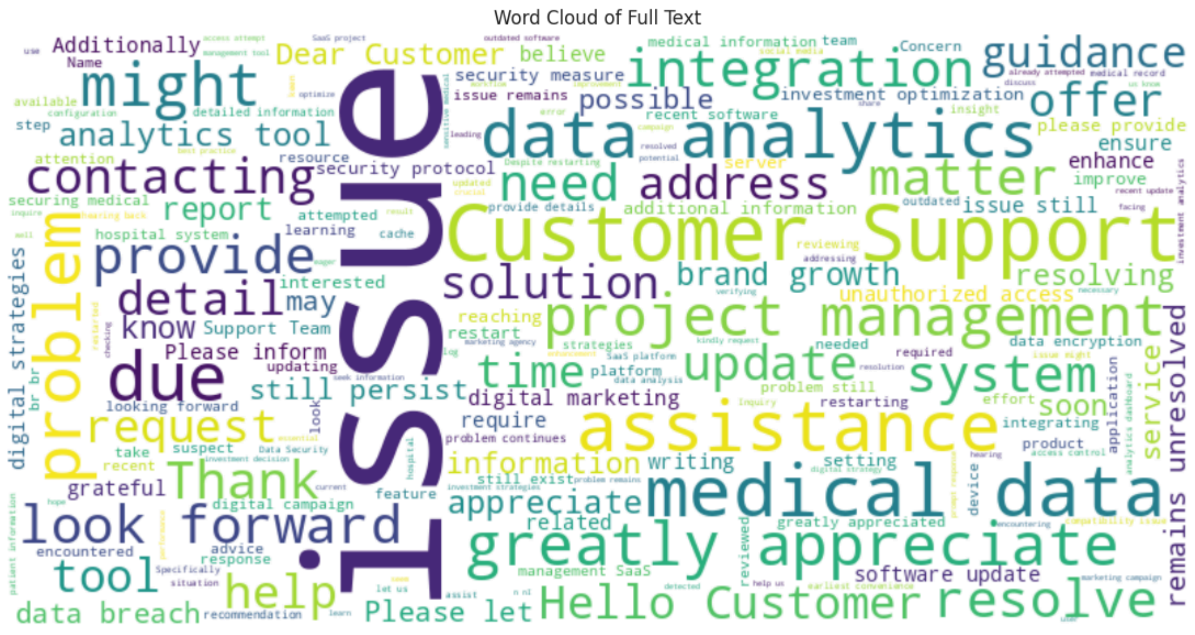




Most of the text falls under the length of 1000 characters and most of the subjects are under 100 character length. Around 1000 subjects have length 0 which suggest there was not subject provided for such datapoints.

2.6.3 Word Cloud of Full Text

To visualize commonly used terms in customer queries, a word cloud was generated from the full_text column.



2.7 Exporting Processed Data

After cleaning and feature engineering, the processed dataset was saved for downstream modeling.

3. MODEL BUILDING AND EVALUATION

3.1 Overview

This section details the training, evaluation, and comparison of three transformer-based models for customer query classification. The models evaluated are:

- BERT-base-uncased
- DistilBERT-base-uncased
- RoBERTa-base

All models were fine-tuned on the same processed dataset under consistent training conditions. Key metrics such as training time, inference time, and classification accuracy were used for comparison.

3.2 Data Preparation

Before training, the dataset was prepared as follows:

- Target Encoding: Queues were converted into numeric labels using LabelEncoder.
- Data Splitting: The data was split into 80% training and 20% validation using train_test_split with stratification.

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['queue'])
train_texts, val_texts, train_labels, val_labels =
train_test_split(
    df['full_text'], df['label'], test_size=0.2,
stratify=df['label'], random_state=42
)
```

3.3 Model Architecture and Tokenization

Each model was loaded using the Hugging Face transformers library, along with the corresponding tokenizer. The models used were:

- bert-base-uncased
- distilbert-base-uncased
- roberta-base

A custom PyTorch Dataset class was used to tokenize the data:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)
train_dataset = TicketDataset(train_texts, train_labels,
tokenizer)
val_dataset = TicketDataset(val_texts, val_labels, tokenizer)
```

3.4 Training Setup

Models were trained for 3 epochs using the Hugging Face Trainer API with custom class weights to handle class imbalance.

TrainingArguments and Custom Trainer:

```
from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir=f"./results/{model_name.replace('/', '_')}",
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    num_train_epochs=3,
    eval_strategy="epoch",
    logging_strategy="epoch",
    save_strategy="no",
    load_best_model_at_end=False,
    report_to="none",
    dataloader_num_workers=0,
    remove_unused_columns=False
)
```

Trainer with Class Weights:

```
trainer = CustomTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    data_collator=DataCollatorWithPadding(tokenizer),
    class_weights=class_weights
)
trainer.train()
```

3.5 Evaluation Metrics

Each model was evaluated on:

- Accuracy on the validation set
- Training Time in seconds
- Inference Time on 100 samples

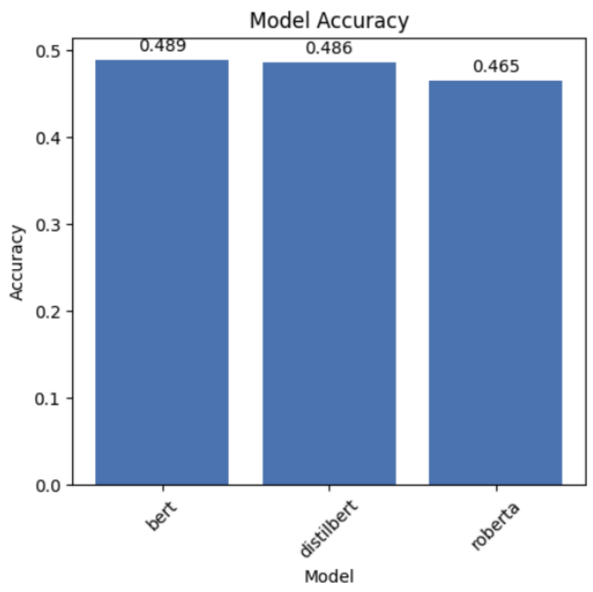
The results were recorded and compiled into a DataFrame for visualization.

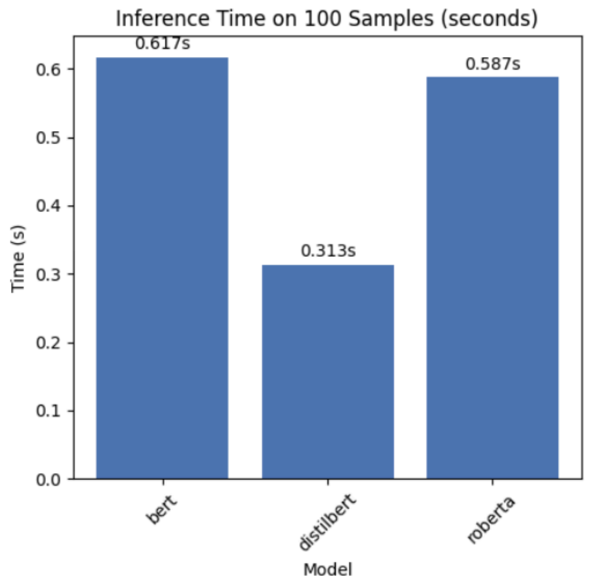
3.6 Results Summary

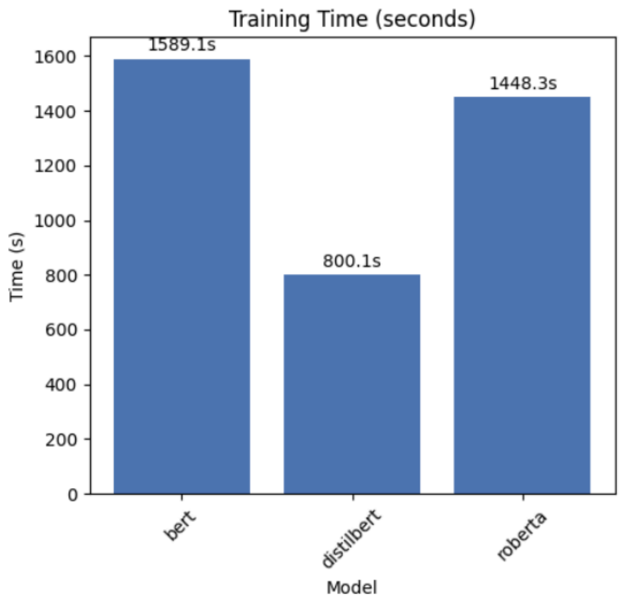
Model	Accuracy	Training Time (s)	Inference Time (s)
BERT-base-uncased	0.489	1589.12	0.617
DistilBERT-base-uncased	0.486	800.14	0.313
RoBERTa-base	0.465	1448.29	0.587

3.7 Visual Comparison

Bar plots were generated to visually compare model performance across the three key metrics.







3.8 Observations

- BERT-base-uncased achieved the highest accuracy (~48.9%) but required the longest training time.

- DistilBERT-base-uncased demonstrated nearly identical performance with significantly reduced training and inference time, making it a more efficient choice for practical deployment.
- RoBERTa-base showed slightly lower accuracy while still incurring a high computational cost.

4. TEXT SUMMARIZATION

4.1 Purpose

In natural language processing tasks involving large-scale customer queries, it is common to encounter lengthy textual inputs. To ensure model efficiency and maintain input within token limits of transformer models, summarization techniques can be applied to compress longer queries into concise, informative representations without losing key contextual information.

This section explores the implementation of an automated summarization pipeline as a pre-processing step to handle long customer queries.

4.2 Summarization Workflow

The summarization was implemented using a transformer-based model (e.g., BART) from Hugging Face's transformers library. The workflow included:

1. Word Count Analysis: Compute the number of words in each query.
2. Threshold Check: Apply summarization only to texts exceeding a defined word count threshold (e.g., 300 words).
3. Batch Processing: Summarize texts in batches for efficiency.
4. Fallback Handling: For edge cases (e.g., model error), fallback to truncated original text.

Summarization Model Example:

```
from transformers import pipeline
summarizer = pipeline("summarization",
model="facebook/bart-large-cnn")
```

4.3 Dataset Analysis for Summarization Need

To determine if summarization was required, the word length of all queries was analyzed.

Output:

```
Mean words: 62.7
Median words: 58.0
Max words: 276
Texts longer than 300 words: 0
```

All queries were well below the summarization threshold. As a result, no records required summarization. The model was not applied further, and the full_text field was retained without modification.

4.4 Summary and Recommendations

Although summarization was not required for the current dataset, this module adds flexibility for handling future datasets with longer queries. It is particularly useful when:

- Queries exceed 300–500 words.
- Input length breaches token limits (e.g., 512 tokens for BERT).
- Efficient memory usage and model throughput are critical.

This modular approach ensures that summarization can be seamlessly activated with minimal changes if the dataset characteristics evolve.

5. FINE-TUNING DISTILBERT WITH DOWNSAMPLING

This section documents the second experiment, where the pre-trained DistilBERT model was fine-tuned on a class-balanced dataset obtained through downsampling. The experiment did not use any weighted loss function. The primary goal was to evaluate model performance when trained on an equal number of samples per class, without applying additional class weight corrections.

5.1 Objective

The objective of this experiment was to investigate whether class balancing via downsampling could improve classification performance on an originally imbalanced dataset. Unlike the previous setup, no class weights or sampling strategies were applied during model training. This experiment serves as a performance baseline using purely balanced data.

5.2 Dataset Balancing via Downsampling

5.2.1 Original Class Distribution

The dataset originally had the following distribution of query labels (queues):

Queue	**Number of Samples**
Technical Support	5245
Product Support	2814

Customer Service	2027
Billing Support	1302
Sales & HR	535

5.2.2 Downsampled Dataset

To create a balanced training set, each class was downsampled to 535 samples, matching the count of the minority class (Sales & HR). This resulted in a total of:

- Total Samples: 535 × 5 = 2675
- Training Set (80%): 2140 samples
- Validation Set (20%): 535 samples

Stratified sampling was used to ensure equal class representation in both splits.

5.3 Model Configuration

The model used was distilbert-base-uncased from the HuggingFace Transformers library. The training was configured as follows:

Tokenizer: **DistilBertTokenizerFast**

Max Token Length: **512** (with truncation and padding)

Batch Size: **64**

Number of Epochs: **30**

Learning Rate: **Default**

Loss Function: **CrossEntropyLoss** (no weights applied)

Evaluation Metric: **Accuracy** (per epoch)

5.4 Training and Validation Results

The training completed successfully across all 30 epochs. However, the model showed limited classification performance on the validation set:

Final Validation Accuracy: 0.50

Observation: The accuracy remained significantly low despite class balancing.

5.5 Conclusion and Observations

Although the class imbalance was corrected via downsampling, the model failed to achieve high accuracy. This can be attributed to the following factors:

- Loss of Information: Downsampling led to the removal of a substantial amount of data from the majority classes (e.g., over 4700 samples were removed from Technical Support). This reduced the model's exposure to valuable linguistic patterns and query diversity.
- Limited Dataset Size: With only 535 samples per class, the model may not have had sufficient data to generalize well.

As a result, while downsampling addressed class imbalance in theory, it degraded overall model performance due to data reduction. This experiment highlights the limitations of naive downsampling, especially for large NLP tasks involving complex sentence structures and nuanced semantics.

6. FINAL MODEL SELECTION

This section outlines the final selected model architecture, its implementation details, training configuration, evaluation metrics, and the rationale behind its selection. The final model is based on DistilBERT which provides a strong trade-off between performance and computational efficiency.

6.1 Rationale for Model Selection

After iterative experimentation with summarization, downsampling, and multiple transformer architectures, the following conclusions were drawn:

- Summarization was not required in our case as the texts were less than 300 words.
- Downsampling led to less accuracy since dataset size was reduced.
- DistilBERT (distilbert-base-uncased) was selected due to:
    - It gave comparable accuracy to BERT and RoBERTa.
    - It took almost half the training time compared to the other two models.
    - Inference time was also almost half of the other two models.

6.2 Dataset and Preprocessing

The dataset consisted of 11,923 support tickets, each with a full_text and a queue label. After preprocessing:

- Null entries were dropped.

- The queue labels were encoded using LabelEncoder
- The dataset was split into 80% training and 20% validation using stratified sampling.

```
train_texts, val_texts, train_labels, val_labels =
train_test_split(
    df['full_text'], df['label'], test_size=0.2,
stratify=df['label'], random_state=42
)
```

Split	Number of Samples
Training	9,538
Validation	2,385

6.3 Model Architecture

The final model used was DistilBERT with a classification head consisting of a linear layer over the [CLS] token representation. The model architecture includes:

- Base: distilbert-base-uncased
- Tokenizer: HuggingFace AutoTokenizer
- Sequence Classification Head: Linear layer with softmax over 5 classes

Tokenizer and Model Initialization:

```
tokenizer =
AutoTokenizer.from_pretrained("distilbert-base-uncased")
model =
AutoModelForSequenceClassification.from_pretrained("distilbert-bas
e-uncased", num_labels=5)
```

If the tokenizer does not have a padding token:

```
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

6.4 Dataset Preparation

A custom PyTorch Dataset (TicketDataset) was implemented to tokenize the text and prepare model inputs with attention masks and labels.

```python
class TicketDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=512):
        self.encodings = tokenizer(texts.tolist(),
truncation=True, padding=True, max_length=max_length)
        self.labels = labels.tolist()

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in
self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx],
dtype=torch.long)
        return item
```

6.5 Handling Class Imbalance

To address class imbalance, weighted cross-entropy loss was used. We computed class weights using the formula:

$$w_c = \frac{n}{|C| \cdot n_c}$$

where:

- n: total number of samples
- |C|: number of classes

- nc : number of samples in class

These weights were passed into a custom Trainer class that overrides the default loss function with CrossEntropyLoss(weight=...).

Weights for each class were calculated and used in a custom trainer:

```
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(train_labels),
    y=train_labels
)
class_weights = torch.tensor(class_weights, dtype=torch.float)
```

Custom Trainer with Weighted Loss:

```
class CustomTrainer(Trainer):
    def __init__(self, *args, class_weights=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.class_weights = class_weights

    def compute_loss(self, model, inputs, return_outputs=False):
        labels = inputs.pop("labels")
        outputs = model(**inputs)
        logits = outputs.logits
        loss_fn =
torch.nn.CrossEntropyLoss(weight=self.class_weights)
        loss = loss_fn(logits, labels)
        return (loss, outputs) if return_outputs else loss
```

6.6 Training Configuration

The model was trained using the HuggingFace Trainer API, with the following hyperparameters:

Parameter	**Value**
Epochs	30
Batch Size	64
Learning Rate	2e-5
Optimizer	AdamW (default)
Scheduler	Linear decay
Warm-up Steps	500
Evaluation Strategy	Per Epoch
Device GPU (cuda)	
Mixed Precision (FP16)	Enabled

Training was performed for 30 epochs, after evaluating convergence over 50 runs. The training process was automatically evaluated after every epoch.

The model was trained using the following configuration:

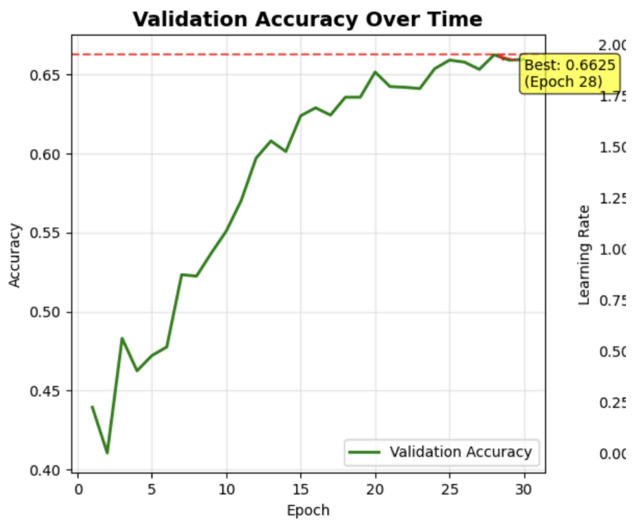
```
training_args = TrainingArguments(
    output_dir="./results/distilbert_30_epochs_original",
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    num_train_epochs=30,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="eval_accuracy",
    learning_rate=2e-5,
    weight_decay=0.01,
    warmup_steps=500,
    lr_scheduler_type="linear",
    fp16=torch.cuda.is_available()
)
```

Model was trained using:

```
trainer = CustomTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    data_collator=DataCollatorWithPadding(tokenizer),
    class_weights=class_weights.to(device),
    compute_metrics=compute_metrics
)
```

6.7 Training Performance

- Total Training Time: 6025 seconds (~1.67 hours)
- Final Accuracy: **66.25%**
- Inference Time (100 samples): 0.3029 seconds
- Parameters: 66,957,317



6.8 Evaluation

This is the classification report for this training output.

Class	Precision	Recall	F1-Score	Support
Billing Support	0.8559	0.7769	0.8145	260
Customer Service	0.4954	0.5345	0.5142	406
Product Support	0.5747	0.5737	0.5742	563
Sales & HR	0.5854	0.4486	0.5079	107
Technical Support	0.7404	0.7531	0.7467	1049
Overall			**0.6625**	2385

6.9 Conclusion

The final DistilBERT-based model with class-weighted training and 30 epochs achieved:

- Good generalization without needing data reduction
- Efficient training time and inference speed
- Balanced performance across five support ticket queues

This configuration is suitable for real-time ticket routing systems and forms the basis for future production deployment.

7. DEPLOYMENT

This section outlines the deployment process of the application, including frontend, backend, and database services. The system was containerized using Docker and orchestrated with Docker Compose to ensure isolated, portable, and reproducible environments.

7.1 Deployment Overview

The application architecture comprises three main services:

- Frontend (React via Vite)
- Backend (FastAPI with ML model processing)
- Database (MySQL)

Each component is packaged into a separate container and managed using Docker Compose. This approach ensures ease of deployment, consistency across environments, and better maintainability.

7.2 Container Structure

Each service is encapsulated in a dedicated Docker container:

	ServiceTechnology	Port Mapping
Backend	FastAPI	8000:8000
Frontend	React + Nginx	1573:80
Database	MySQL	3307:3306

- The backend container waits for the database to be fully initialized before starting. This is implemented to prevent errors like Segmentation Fault due to early DB queries.

- Persistent storage for MySQL is ensured using a Docker volume named mysql_data.

7.3 Docker Compose Configuration

Below is a code snippet from the docker-compose.yml file illustrating the setup:

```yaml
version: '3.8'
services:
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: ticket_system
    ports:
      - "3307:3306"
    volumes:
      - mysql_data:/var/lib/mysql

  backend:
    build: ./backend
    depends_on:
      - db
    ports:
      - "8000:8000"
    environment:
      DB_HOST: db
      DB_PORT: 3306
    command: >
      sh -c "while ! nc -z db 3306; do sleep 1; done &&
             uvicorn main:app --host 0.0.0.0 --port 8000"

  frontend:
    build: ./frontend
    ports:
      - "1573:80"
    depends_on:
      - backend
    restart: always

volumes:
  mysql_data:
```

7.4 Backend Deployment

The backend is implemented using FastAPI with the following stack:

- SQLAlchemy for ORM
- Pydantic for request/response models
- asyncio for handling asynchronous background tasks

Key Components:

- ModelService: Responsible for loading and serving the machine learning model.
- TicketProcessor: A background job that runs every 10 seconds to check and process tickets. It updates ticket statuses:

```
Pending → Processing → Completed / Failed
```

- Models are organized inside the /backend/models directory.

7.5 Frontend Deployment

The frontend is built with React using Vite as the bundler. It is served using Nginx in the production Docker container.

Pages:

- User Page: Allows users to submit a ticket with Subject and Body, simulating an email-like interface.
- Support Dashboard: Displays classified tickets for support agents and allows queue

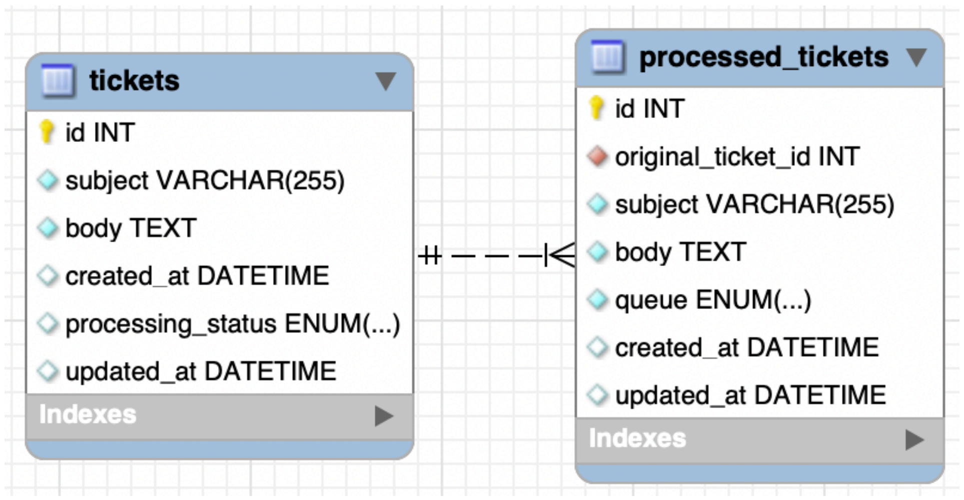
reassignment in case of misclassification.

API calls are handled using the axios library.

7.6 Database Setup

Database: ticket_system

Entity Relationship Diagram:

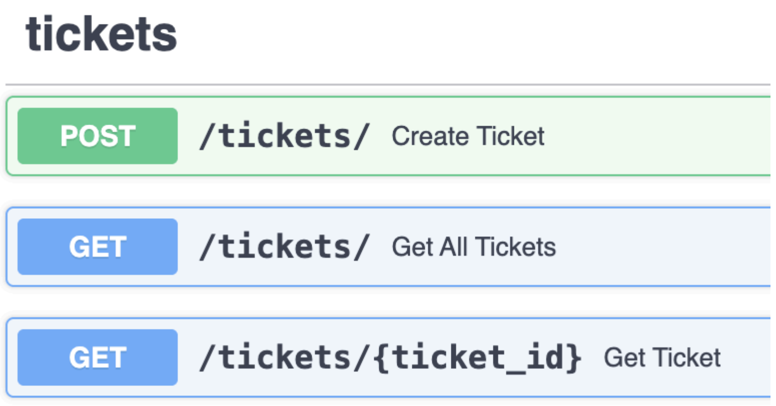


Tables:

- tickets – Stores raw ticket information.
- processed_tickets – Stores tickets post ML classification.

7.7 Backend API Overview

All APIs are documented and structured for modular access:

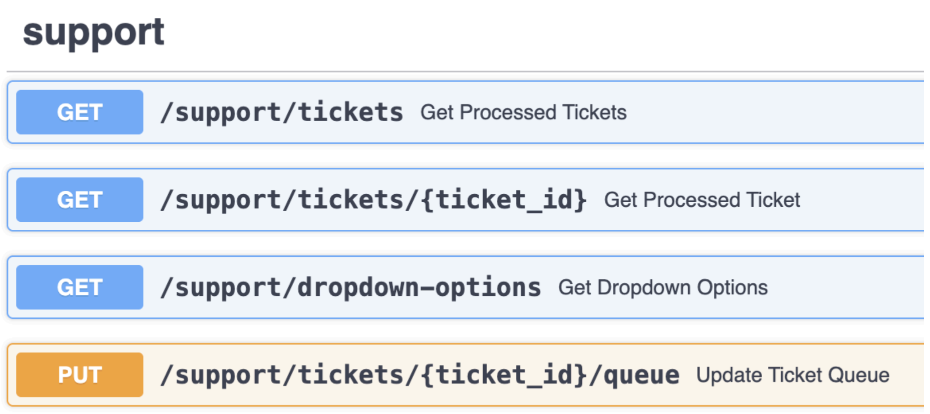


tickets

POST /tickets/ Create Ticket

GET /tickets/ Get All Tickets

GET /tickets/{ticket_id} Get Ticket



support

GET /support/tickets Get Processed Tickets

GET /support/tickets/{ticket_id} Get Processed Ticket

GET /support/dropdown-options Get Dropdown Options

PUT /support/tickets/{ticket_id}/queue Update Ticket Queue