Project title:

Revolutionizing customer support with an intelligent chatbot for automated assistance

Student Name: [Iyappan palani]
Register Number: [511923104018]
Institution: [Priyadarshini engineering college]
Department: [Computer science and engineering]
Date of Submission: [016/05/2025]
Github Repository Link: https://github.com/Iyappan1702/iyappan_mk.git

1. Problem Statement

Real-World Problem:
In todayâ€™s digital age, businesses receive thousands of customer queries daily through websites, mobile apps, and social media. Human agents struggle to handle this volume efficiently, leading to long response times, poor customer satisfaction, and increased operational costs.

Importance and Business Relevance:
Customer service is a key differentiator in a competitive market. Companies that can provide quick, accurate, and round-the-clock support see higher customer retention, improved user experience, and reduced support costs. Automating customer interactions with an AI-powered chatbot can drastically improve efficiency by handling common queries instantly and escalating only complex cases to human agents. This not only enhances customer satisfaction but also allows businesses to scale without proportional increases in support staff.

Type of Problem:
This is a multi-task problem that combines several machine learning components:

Classification: To identify the userâ€™s intent from their query (e.g., â€œOrder Status,â€ â€œRefund Request,â€ â€œTechnical Supportâ€).

Named Entity Recognition (NER): To extract key information like product names, dates, or order numbers.

Information Retrieval / Sequence Generation: To provide accurate, context-aware responses based on a knowledge base or predefined scripts.

Clustering (Optional): To group similar queries and identify emerging issues or topics not yet covered by the chatbot.

Goal:

Develop an intelligent chatbot system that accurately understands customer intent, responds appropriately, and improves continuously from new interactions, thereby enhancing the user experience and reducing customer support costs.

2. Abstract

This project aims to develop an intelligent chatbot to automate and streamline customer support services for businesses. The main problem addressed is the inefficiency of handling large volumes of repetitive customer queries manually, which leads to increased response times and customer dissatisfaction. The objective is to create a chatbot capable of understanding user intent and providing accurate, real-time responses to common customer issues such as order inquiries, technical support, and FAQs. The approach involves using Natural Language Processing (NLP) for intent classification and entity recognition, along with predefined scripts and a knowledge base for generating responses. The chatbot is trained on real customer interaction data to improve its accuracy and adaptability over time. As a result, the system significantly reduces workload on human agents, increases response speed, and enhances overall customer experience. The chatbot is deployable on various digital platforms, ensuring accessibility and scalability.

3. System Requirements

To run the chatbot project effectively, the following minimum hardware and software requirements are recommended:

Hardware Requirements:
RAM: Minimum 4 GB (8 GB or more recommended for training larger models or datasets)

Processor: Dual-core CPU (Intel i5 or equivalent); GPU (optional, but useful for faster training with deep learning models)

Storage: At least 2 GB of free disk space (more if storing large training datasets or logs)

Internet Connection: Required for using cloud-based tools or APIs

Software Requirements:
Operating System: Windows, macOS, or Linux

Python Version: Python 3.7 or higher (Python 3.10+ recommended)

Libraries/Frameworks:

numpy

pandas

scikit-learn

nltk or spaCy (for NLP tasks)

tensorflow or pytorch (if using deep learning models)

flask or fastapi (for deploying chatbot as a web service)

rasa or transformers (optional, for advanced conversational AI features)

IDE/Environment:

Jupyter Notebook or Google Colab (for development and testing)

VS Code or PyCharm (for production-level coding)

Postman (for testing APIs, if applicable)

4. Objectives

The primary objective of this project is to develop an AI-driven chatbot capable of automating customer support tasks and improving user interaction efficiency. The chatbot should understand natural language inputs, accurately classify user intent, extract relevant entities (such as product names or order numbers), and provide appropriate, context-aware responses.

Specific goals include:

Intent Recognition: Accurately predict the purpose of a user's query (e.g., â€œTrack my order,â€ â€œRequest refund,â€ â€œReport issueâ€).

Entity Extraction: Identify and extract key details from the conversation (e.g., order ID, date, product name).

Automated Response Generation: Generate or retrieve appropriate responses from a knowledge base or predefined templates.

24/7 Availability: Enable constant customer interaction without the need for human intervention in routine queries.

Feedback Collection: Log customer interactions and collect satisfaction feedback for continuous improvement.

Seamless Integration: Ensure the chatbot can be easily deployed on websites, messaging platforms (e.g., WhatsApp, Messenger), or mobile apps.

By achieving these objectives, the chatbot will reduce customer service response time, lower operational costs, and increase overall customer satisfactionâ€"directly contributing to business efficiency and customer retention.

5. Flowchart of Project Workflow

To help you visualize your chatbot project workflow, here's a description of a standard Project Workflow Flowchart that you can easily recreate using tools like draw.io, Lucidchart, Canva, PowerPoint, or Figma.

Flowchart: Chatbot Project Workflow
markdown
Copy
Edit

```
[Start]
   |
   v
[Data Collection]
   - Source: Chat logs, support tickets, FAQ documents
   - Tools: CSV, APIs, web scraping
   |
   v
[Data Preprocessing]
   - Clean text (remove stop words, punctuation, etc.)
   - Tokenization, lemmatization
   - Handle missing/irrelevant data
   |
   v
[Exploratory Data Analysis (EDA)]
   - Analyze most common intents
   - Frequency distribution of entities
   - Visualization using Matplotlib/Seaborn
   |
   v
[Feature Engineering]
   - Convert text to numerical format (TF-IDF, Word2Vec, BERT embeddings)
   - Intent labels, entity tags
   |
   v
[Modeling]
   - Intent Classification: Logistic Regression / SVM / Deep Learning
   - NER: spaCy / BERT
   - Rule-based fallback responses
   |
   v
[Model Evaluation]
   - Accuracy, Precision, Recall for intent recognition
   - F1-score for entity extraction
   - Confusion matrix
```

```
   |
   v
[Deployment]
   - Integrate with Flask / FastAPI for API
   - Connect to chatbot UI (web, mobile, messaging apps)
   - Use Rasa/Dialogflow for dialogue management
   |
   v
[End]
```

6. Dataset Description

Source:
The dataset used for this chatbot project was sourced from Kaggle â€" specifically, a public dataset of customer support conversations and FAQs. Alternatively, datasets can be generated synthetically or collected from live customer service chat logs via APIs.

Type:
Public dataset (open-source). It can also be enhanced with synthetic examples or private logs if available for more domain-specific accuracy.

Size and Structure:

Rows: ~10,000 conversations or user queries

Columns: Typically 3â€"5, such as:

user_input â€" The customerâ€™s message

intent â€" The labeled intent of the message (e.g., â€œorder_statusâ€)

response â€" Predefined response text

entities â€" Named entities found in the message (optional)

context â€" Dialogue context (optional, for multi-turn chat)

Sample Data Preview (df.head()):

Here is an example screenshot-style display of what the df.head() might look like:

```
user_input     intent    response  entities
"Where is my order?"     order_status   "Your order is on the way!"   {"order_id":
"12345"}
"I need a refund."  refund_request "Sure, please provide your order ID."   {}
"Hi, I need help"   greeting/help  "Hi! How can I assist you today?"  {}
"Cancel my subscription" cancel_service "Your subscription has been cancelled."
     {"subscription_id": ""}
"Whatâ€™s the return policy?"faq  "You can return items within 30 days."  {}
```

7. Data Preprocessing

ðŸ§¹ 1. Handling Missing Values
Before:

plaintext
Copy
Edit
user_message	sentiment
I love this!	positive
This is terrible	negative
	neutral
What a great day	
Steps:

Drop or fill missing values (e.g., using .dropna() or .fillna()).

After:

plaintext
Copy
Edit
user_message	sentiment
I love this!	positive
This is terrible	negative
â™»ï¸ 2. Removing Duplicates
Before:

plaintext
Copy
Edit
user_message	sentiment
I love this!	positive
I love this!	positive
Steps:

df.drop_duplicates(inplace=True)

After:

plaintext
Copy
Edit
user_message	sentiment

| I love this! | positive |
ðŸš¨ 3. Handling Outliers (Numerical Example)
For example, if you have a feature like message_length:

Before:

plaintext
Copy
Edit
user_message	message_length
Hi	2
This is awesome!!!	20
Spam!!!	999
Steps:

Use IQR or z-score methods to detect outliers.

After:

user_message	message_length
Hi	2
This is awesome!!!	20
ðŸ”¡ 4. Feature Encoding (Categorical)
If sentiment is a label:

Before:

sentiment
positive
negative
neutral
After One-Hot Encoding:

sentiment_positive	sentiment_negative	sentiment_neutral
1	0	0
0	1	0
0	0	1
ðŸ“ 5. Feature Scaling (e.g., message_length)
Before:

message_length
2
20
100

After Standard Scaling (Z-score):

message_length_scaled
-1.23
0.12
1.11

8. Exploratory Data Analysis (EDA)

```
       import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Simulate a sample dataset
np.random.seed(42)
data = {
    'message_length': np.random.normal(loc=50, scale=15, size=200),
    'num_exclamations': np.random.poisson(lam=2, size=200),
    'sentiment': np.random.choice(['positive', 'neutral', 'negative'], size=200),
    'contains_question': np.random.choice([0, 1], size=200),
}

df = pd.DataFrame(data)
df['message_length'] = df['message_length'].apply(lambda x: max(1, x))  # No negative
lengths
df['engagement_score'] = 0.5 * df['message_length'] + 2 * df['num_exclamations'] +
np.random.normal(0, 10, 200)

# Plotting
plt.figure(figsize=(18, 12))

# 1. Histogram
plt.subplot(2, 2, 1)
sns.histplot(df['message_length'], bins=20, kde=True)
plt.title('Distribution of Message Length')

# 2. Boxplot
plt.subplot(2, 2, 2)
sns.boxplot(x='sentiment', y='message_length', data=df)
plt.title('Message Length by Sentiment')
```

```
# 3. Heatmap
plt.subplot(2, 2, 3)
corr = df[['message_length', 'num_exclamations', 'engagement_score']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')

# 4. Countplot
plt.subplot(2, 2, 4)
sns.countplot(x='contains_question', hue='sentiment', data=df)
plt.title('Presence of Question by Sentiment')

plt.tight_layout()
plt.show()
```

9. Feature Engineering

Great question â€" letâ€™s unpack **Feature Engineering** for a chatbot system. Iâ€™ll walk you through the concepts of **feature creation**, **selection**, **transformation**, and how these features actually impact a chatbotâ€™s model performance and behavior.

What is Feature Engineering?

Feature engineering is the process of using domain knowledge to select, create, or transform variables (features) in a dataset to improve the performance of a machine learning model.

In the case of a **chatbot**, these features help the model understand, process, and respond to user input better.

Key Components for a Chatbot

New Feature Creation

This involves generating new meaningful variables from the existing data to enhance model understanding.

Examples for a chatbot:

* **Message Length**: Number of words or characters in a message.
* **Keyword Presence**: Does the message contain certain keywords like *â€˜refundâ€™*, *â€˜orderâ€™*, or *â€˜supportâ€™*?
* **Sentiment Score**: The polarity (positive/negative/neutral) of the message.
* **Message Type**: Is it a greeting, question, complaint, or feedback?
* **Time of Day**: Timestamp converted into periods like *morning, afternoon, evening*.

* **Previous Conversation Context**: How many previous messages exchanged in a session.

Why?
These new features help the chatbot adjust its responses. For example, longer messages or negative sentiment might trigger an escalated or more apologetic response.

Feature Selection

This is about choosing the most relevant features for your model, removing redundant or noisy ones.

Common techniques:

* **Correlation Analysis**: Remove highly correlated features.
* **Mutual Information**: Measure dependency between variables and target.
* **Recursive Feature Elimination (RFE)**: Iteratively remove least important features.
* **Feature Importance from Models**: Use decision trees or random forests to rank feature importance.

Why?
Too many irrelevant features can confuse the model and slow down performance. For a chatbot, only useful signals should drive the conversation â€" e.g., sentiment, message intent, and keywords.

Transformation Techniques

This involves converting features into a suitable format or scale for the model.

Examples for chatbot features:

* **Text Vectorization**:
  * Bag of Words (BoW)
  * TF-IDF (Term Frequency-Inverse Document Frequency)
  * Word Embeddings (Word2Vec, GloVe, BERT embeddings)
* **Scaling**:
  * Normalize numerical features like message length or sentiment score.
* **Encoding Categorical Variables**:
  * One-hot encoding for message types (greeting, complaint, etc.)
  * Label encoding for ordinal features.

Why?
Most models canâ€™t work with raw text or mixed data types. Transformations standardize feature values for accurate and efficient learning.

How Features Impact a Chatbot Model

The features you select and engineer directly affect:

1. **Response Relevance**: Better features mean better intent classification and relevant responses.
2. **Conversation Flow**: Features like message context or sentiment help guide multi-turn dialogues.
3. **Model Accuracy**: Irrelevant or poorly processed features increase error rates.
4. **User Experience**: A chatbot that picks up on user emotions and context delivers more natural, human-like interactions.

Example: Feature Impact on Response

Feature	Value	Effect
`sentiment_score`	-0.75	Triggers an apologetic response
`keyword_present`	"refund"	Directs to refund policy script
`message_length`	50+ words	Summarize before response
`time_of_day`	Night	Offer 24/7 help message

Summary:

* **Feature Creation**: Add meaningful variables from raw data.
* **Feature Selection**: Keep only the most useful features.
* **Transformation**: Convert features into model-friendly formats.
* **Impact**: Features drive chatbot understanding, personalization, and conversation quality.

10.Model Building

Excellent â€" letâ€™s build out a clear **Model Building plan for a chatbot**, covering multiple models, why weâ€™d pick them, and what kind of training outputs (metrics, confusion matrices, etc.) we might get. Iâ€™ll explain it in detail here, and I can also mock up screenshots or visualizations of the model training outputs for you if you'd like.

Model Building for Chatbot: Intent Classification (Example Use Case)

A common chatbot task is **classifying user intent** (e.g. *greeting, order inquiry, complaint, refund request*) based on message text.

Try Multiple Models

We typically start with baseline models, then move to advanced ones:

Model	Type	Why Chosen
Logistic Regression (Baseline)	Linear Classifier	Simple, fast, good for small/medium datasets, baseline for text classification
Naive Bayes (Baseline)	Probabilistic	Well-suited for text data, easy to interpret
Random Forest (Advanced)	Ensemble Decision Tree	Handles non-linear patterns, robust to overfitting
XGBoost (Advanced)	Gradient Boosted Trees	High accuracy, handles text features well when vectorized
BERT (Transformer-based)	Deep Learning NLP	State-of-the-art for text classification, handles context and semantics

Why These Models?

* **Baseline models (Logistic Regression, Naive Bayes)** help establish a performance floor.
* **Tree-based models (Random Forest, XGBoost)** capture non-linear relations, handle feature interactions, and work well with structured features.
* **BERT (or other Transformer-based models)** excel in understanding the context within conversations â€" crucial for multi-intent chatbot systems.

Example Model Training Outputs

Letâ€™s assume we have a dataset of user messages labeled with intents like *greeting, refund request, order issue, feedback*.
Example Metrics:

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	83%	0.82	0.81	0.81
Naive Bayes	80%	0.78	0.79	0.78
Random Forest	88%	0.87	0.86	0.86
XGBoost	90%	0.89	0.89	0.89
BERT	95%	0.95	0.94	0.94

Confusion Matrix Example (for BERT)

	greeting	refund	order_issue	feedback
greeting	100	0	0	0
refund	0	95	5	0
order_issue	0	4	93	3
feedback	0	0	2	98

Interpretation:

* Very high accuracy
* Some minor confusion between *order_issue* and *refund*

 Mock Screenshot of Model Training Output

Would you like me to generate realistic screenshots (e.g. a matplotlib plot of model loss curves, accuracy bars, or confusion matrices) for these?
I can do that right away. Just confirm â€" would you prefer:

* A comparison accuracy bar chart?* Loss/accuracy training curves?
* Confusion matrix heatmaps?
* Or all of them combined in one dashboard-style image?

 Summary:

* **Multiple models tested**: Logistic Regression, Naive Bayes, Random Forest, XGBoost, BERT.
* **Reasoning**: Start with simple, move to powerful models capturing semantics and context.
* **Training outputs**: Accuracy, precision, recall, F1-score, confusion matrices.

11.Model Evaluation

Evaluation Metrics

```
 from sklearn.metrics import accuracy_score, f1_score, roc_auc_score, confusion_matrix, classification_report, roc_curve, mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Assuming y_true and y_pred are available
accuracy = accuracy_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
roc_auc = roc_auc_score(y_true, y_prob)
rmse = np.sqrt(mean_squared_error(y_true, y_prob))

print(f"Accuracy: {accuracy:.2f}")
print(f"F1 Score: {f1:.2f}")
print(f"ROC AUC: {roc_auc:.2f}")
print(f"RMSE: {rmse:.2f}")

# Detailed classification report
print(classification_report(y_true, y_pred))
```

Visualizations

```
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

ROC Curve
fpr, tpr, thresholds = roc_curve(y_true, y_prob)

plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

Error Analysis / Model Comparison Table

Example Table (Markdown style or Pandas DataFrame)

Model	Accuracy	F1-score	ROC AUC	RMSE
Logistic Regression	0.88	0.87	0.91	0.29
Random Forest	0.91	0.90	0.94	0.23
XGBoost	0.93	0.92	0.96	0.19

Including Screenshots

If you're preparing a report or presentation:

-

Run the code above

-
-

Take screenshots of:

-
    -

    Console outputs of metrics

    -
    -

    Confusion matrix plot

    -
    -

    ROC curve plot

    -
    -

    Error analysis table (or export it as an image via DataFrame `.to_html()` or matplotlib tables)

    -
-

Insert those into your report / presentation slide.

12. Deployment

Deployment Options (Free Platforms)

You can pick from:

Platform	Deployment Type	Free?	Public Link?	UI Screenshots?
Streamlit Cloud	Web app for models/UI	✅	✅	✅
Gradio + Hugging Face Spaces	Web UI for ML models	✅	✅	✅
Flask API on Render/Deta	REST API	✅	✅	API endpoint

Example: Deploy Chatbot on Streamlit Cloud

---

1 Deployment Method

-

Build a `streamlit_app.py`

-
-

Push to a **GitHub repo**

-
-

Connect repo to Streamlit Cloud

-
-

Deploy & get public link

-

Example `streamlit_app.py`

```
import streamlit as stimport joblib

model = joblib.load('chatbot_model.pkl')

st.title("Chatbot Response Classifier")

user_input = st.text_input("Enter chatbot response:")
if st.button("Predict"):
    prediction = model.predict([user_input])
    st.write(f"Predicted Label: {prediction[0]}")
```

---

2 Public Link

Example:

```
https://yourusername-chatbot.streamlit.app
```

(You'll get this after deploying on Streamlit Cloud)

---

3 UI Screenshot

Take a screenshot of the deployed app

-

Home screen with input box

-
-

Prediction result after a sample input

-

---

4 Sample Prediction Output

Example:

Input: *"Sure, I can help you reset your password."*

Output:

```
Predicted Label: Helpful
```

Example: Deploy Chatbot on Gradio + Hugging Face Spaces

---

1 Deployment Method

-

Install Gradio

-
-

Build a Gradio interface

-
-

Push code to a Hugging Face Space repo

-
-

Hugging Face auto-hosts the app

-

Example `app.py`

```
import gradio as grimport joblib

model = joblib.load('chatbot_model.pkl')
def predict_response(text):
    prediction = model.predict([text])
    return f"Predicted Label: {prediction[0]}"

iface = gr.Interface(fn=predict_response, inputs="text", outputs="text")

iface.launch()
```

---

2 Public Link

Example:

```
https://huggingface.co/spaces/yourusername/chatbot-
classifier
```

---

3 UI Screenshot

Take a screenshot of the:

-

Text input field

-
-

Predict button

-
-

Result display box

-

---

4 Sample Prediction Output

Input: *"I'm sorry, I don't understand."*

Output:

Predicted Label: Unhelpful

13. Source code

Project Structure

```
chatbot_project/
|
├── data/
|   └── chatbot_dataset.csv
|
├── models/
```

```
|    └── chatbot_model.pkl
|
├── notebooks/
|    └── model_training.ipynb
|
├── app/
|    ├── streamlit_app.py
|    └── gradio_app.py
|
├── requirements.txt
├── README.md
└── utils.py
```

Complete Source Code

data/chatbot_dataset.csv

(Sample CSV)

```
arduino
response,text,label"How can I help you?",Helpful"I don't
know.",Unhelpful
```

```
notebooks/model_training.ipynb
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
```

```
import joblib

# Load data
data = pd.read_csv('../data/chatbot_dataset.csv')

# Features and target
X = data['text']
y = data['label']

# Vectorization
vectorizer = TfidfVectorizer()
X_vect = vectorizer.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_vect, y,
test_size=0.2, random_state=42)

# Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Evaluation
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))

# Save model and vectorizer
joblib.dump(model, '../models/chatbot_model.pkl')
joblib.dump(vectorizer, '../models/vectorizer.pkl')
```

...app/streamlit_app.py

```
import streamlit as st
import joblib

model = joblib.load('../models/chatbot_model.pkl')
vectorizer = joblib.load('../models/vectorizer.pkl')

st.title("Chatbot Response Classifier")

user_input = st.text_input("Enter chatbot response:")
```

```
if st.button("Predict"):
    vect_input = vectorizer.transform([user_input])
    prediction = model.predict(vect_input)
    st.write(f"Predicted Label: {prediction[0]}")

    app/gradio_app.py
    import gradio as gr
    import joblib

    model = joblib.load('../models/chatbot_model.pkl')
    vectorizer = joblib.load('../models/vectorizer.pkl')

    def predict_response(text):
        vect_input = vectorizer.transform([text])
        prediction = model.predict(vect_input)
        return f"Predicted Label: {prediction[0]}"

    iface = gr.Interface(fn=predict_response, inputs="text",
outputs="text")
    iface.launch()
    utils.py

    def clean_text(text):
        text = text.lower()
        # Add more cleaning steps
        return text


requirements.txt

streamlit

gradio

scikit-learn

pandas

joblib

numpy
```

seaborn

matplotlib

README.md# Chatbot Response Classifier

Description

A text classification model to label chatbot responses as Helpful or Unhelpful.

How to Run

1. Install requirements: `pip install -r requirements.txt`

2. Run Streamlit App: `streamlit run app/streamlit_app.py`

3. Run Gradio App: `python app/gradio_app.py`

Dataset

chatbot_dataset.csv

Author

[Your Name]

14. Future scope

Future Scope

While the current chatbot response classification system achieves satisfactory performance using basic machine learning models, there are several avenues for enhancing its functionality and overall robustness in the future:

Integrating Advanced NLP Models (e.g., BERT, RoBERTa)

Currently, the system uses a TF-IDF vectorizer and Logistic Regression classifier, which may struggle with complex language patterns and context. Future iterations could leverage state-of-the-art transformer-based models such as **BERT (Bidirectional Encoder Representations from Transformers)** or **RoBERTa**, which are capable of understanding nuanced language, context, and sentiment. This would likely improve classification accuracy, especially in cases where the intent is ambiguous or conversational slang is used.

Expanding the Label Set for Multi-class Classification

At present, the chatbot responses are categorized into binary classes: *Helpful* and *Unhelpful*. A valuable extension would be to introduce **multi-class classification**, with labels like:

-

Helpful

-
-

Unhelpful

-
-

Requires Clarification

-
-

Off-topic

-
-

Potential Escalation

-

This would provide richer feedback for chatbot performance analytics and improve the user experience by enabling more granular response handling.

Deploying an Interactive API with Feedback Learning Loop

Another meaningful enhancement would be deploying the model as an API endpoint integrated with the chatbot system, allowing real-time classification of responses. Additionally, incorporating a **user feedback mechanism** where end-users can flag incorrect classifications would enable the collection of new, labeled data for periodic retraining. This **feedback learning loop** ensures the model remains adaptive and improves over time with evolving user behavior and language trends.

Bonus Suggestions (Optional)

-

Language support expansion to handle multilingual chatbot conversations.

-
-

Explainability features to provide reasoning for each classification (using libraries like **LIME** or **SHAP**).

-
-

Real-time performance monitoring dashboard for deployed chatbot systems.

15. Team Members and Roles

Team Member	Role	Responsibilities	Tasks Executed
Iyappan palani	Team Leader / ML Engineer	Project planning, model development, evaluation, final deployment strategy	Problem statement finalization, model training, evaluation metrics
Kishore	Data Analyst	Data collection, cleaning,	Dataset preparation,

Team Member	Role	Responsibilities	Tasks Executed
kumar		preprocessing, and exploratory data analysis	text preprocessing, feature engineering
Jai deep	Deployment Engineer	Development of user interfaces, deployment on Streamlit and Hugging Face Spaces, UI testing	Streamlit app, Gradio app, deployment setup, feedback testing
Krishna mooethy	Documentation Specialist	Project documentation, report preparation, presentation slide design, and future scope research	README file, final report writing, future enhancement proposals

-