# **RAG Model for QA Bot Documentation**

#### **Table of Contents**

- 1. Project Overview
- 2. Technologies Used
- 3. <u>Dataset Preparation</u>
- 4. System Architecture
- 5. Implementation Details
  - Environment Setup
  - o Code Structure
  - Pinecone Setup
  - o Flask API Implementation
- 6. Optimization Techniques
  - Enhanced Query Expansion
  - o Fine-Tuning with User Feedback
- 7. Comparison of Fine-Tuning Approaches
- 8. Testing the Model
- 9. Conclusion
- **10.References**

## **Project Overview**

The Retrieval-Augmented Generation (RAG) model is designed to improve the performance of question-answering (QA) systems by combining retrieval of relevant documents with generative language capabilities. This project leverages the OpenAI API for natural language processing and Pinecone DB for efficient vector storage and retrieval.

## **Technologies Used**

- OpenAl API: For generating text responses and embeddings.
- Pinecone DB: A vector database for storing and retrieving document embeddings.
- Flask: A lightweight web framework for creating the API.
- **Python**: The primary programming language for implementation.
- dotenv: For managing environment variables.

## **Dataset Preparation**

#### **Techniques for Developing and Refining Datasets**

- 1. **Data Collection**: Gather diverse and relevant data from multiple sources, such as articles, forums, and domain-specific datasets.
- 2. **Data Cleaning**: Remove noise, duplicates, and inconsistencies. Normalize data formats.
- 3. **Data Annotation**: Annotate data with labels indicating the correct outputs or context. Engage experts for quality review.
- 4. Data Augmentation: Generate synthetic examples to increase dataset variability.
- 5. **Dataset Splitting**: Split the dataset into training, validation, and test sets using stratified sampling.
- 6. **Continuous Refinement**: Implement feedback loops for iterative dataset improvement.

## **System Architecture**

The system architecture consists of the following components:

- **User Interface**: Where users input their questions.
- **Flask API**: Handles incoming requests, queries the vector database, and interacts with the OpenAI API.
- **Pinecone DB**: Stores document embeddings for efficient retrieval.
- OpenAl API: Generates answers based on retrieved context.

# **Implementation Details**

### **Environment Setup**

### 1.Install Required Packages:

pip install openai pinecone-client flask python-dotenv

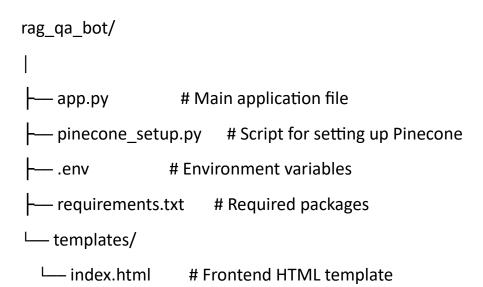
**2.Environment Variables**: Create a .env file with the following content:

```
OPENAI_API_KEY=your_openai_api_key

PINECONE_API_KEY=your_pinecone_api_key

PINECONE_ENVIRONMENT=your_pinecone_environment

Code Structure
```



#### **Pinecone Setup**

In pinecone\_setup.py, set up the Pinecone index and upload document embeddings:

import pinecone

import os

import openai

from dotenv import load dotenv

```
# Load environment variables
load dotenv()
pinecone.init(api_key=os.getenv("PINECONE_API_KEY"),
environment=os.getenv("PINECONE ENVIRONMENT"))
index name = "qa-bot"
pinecone.create index(index name, dimension=1536) # Change dimension
according to the model used
index = pinecone.Index(index_name)
# Upload document embeddings
documents = ["doc1 text", "doc2 text", "doc3 text"]
for doc in documents:
  embedding = openai.Embedding.create(input=doc, model="text-embedding-
ada-002")['data'][0]['embedding']
  index.upsert([(str(documents.index(doc)), embedding, {"text": doc})])
Flask API Implementation
In app.py, implement the Flask API that handles user queries:
from flask import Flask, request, jsonify, render_template
import openai
import pinecone
import os
from dotenv import load dotenv
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")
```

```
pinecone.init(api key=os.getenv("PINECONE API KEY"),
environment=os.getenv("PINECONE_ENVIRONMENT"))
index = pinecone.Index("qa-bot")
app = Flask(__name__)
@app.route('/')
def home():
  return render_template('index.html')
@app.route('/ask', methods=['POST'])
def ask():
  data = request.json
  question = data.get("question", "")
  question embedding = openai. Embedding.create(input=question,
model="text-embedding-ada- 002")['data'][0]['embedding']
  query results = index.query(queries=[question embedding], top k=3,
include_metadata=True)
  context = "\n\n".join([match["metadata"]["text"] for match in
query results["matches"]])
  response = openai.Completion.create(
    model="text-davinci-003",
    prompt=f"Answer the following question using the context
below:\n\nContext: {context}\n\nQuestion: {question}\nAnswer:",
    temperature=0.7,
    max_tokens=150
  )
  answer = response["choices"][0]["text"].strip()
```

```
return jsonify({"answer": answer})
if __name__ == "__main__":
    app.run(debug=True)
```

# **Optimization Techniques**

## **Enhanced Query Expansion**

• Utilize synonym replacement and contextual embeddings to expand user queries, improving retrieval quality.

## **Fine-Tuning with User Feedback**

• Implement a feedback mechanism to refine the model based on user interactions, enhancing relevance and accuracy.

Approach	Description	Advantages	Disadvantages
Full Fine- Tuning	Update all model parameters	High performance, task-specific adaptation	Requires more resources, risk of overfitting
Feature Extraction	Use model as a fixed feature extractor	d Faster training, less overfitting risk	Limited performance
Adapters	Introduce task- specific modules	Parameter-efficient, multi-task capabilities	Requires careful design
Prompt Tuning	Craft specific prompts without changing weights	Minimal resource requirements	Relies heavily on prompt engineering skills

## **Comparison of Fine-Tuning Approaches**

### **Preferred Method: Full Fine-Tuning**

Full fine-tuning is preferred for its ability to maximize performance by fully adapting the model to specific tasks.

## **Testing the Model**

1.Start the Flask app:

python app.py

- 1. Open your web browser and navigate to http://127.0.0.1:5000.
- 2. Enter a question in the input field and submit to receive an answer.

#### Conclusion

The RAG model for a QA bot combines retrieval and generation techniques to provide accurate and contextually relevant answers. By leveraging optimization techniques and thorough dataset preparation, the model can be fine-tuned for superior performance.

#### References

OpenAl API Documentation: OpenAl API

• Pinecone Documentation: Pinecone

• Flask Documentation: Flask