TASK – 2

Optimizing RAG Model

Technique 1: Fine-tuning the Retriever Model

Fine-tuning the retriever model can significantly enhance the relevance of the documents retrieved in response to a query. Here's how to implement this:

1. **Collect Domain-Specific Data**: Gather a large corpus of domain-specific documents related to the business. This could include FAQs, manuals, emails, and other relevant documents.
2. **Preprocess Data**: Clean and preprocess the data to remove noise. Tokenize the documents and convert them into embeddings using a pre-trained model like sentence-transformers.
3. **Fine-tune the Model**: Fine-tune a pre-trained model on the domain-specific data. This involves training the model to better understand the context and semantics of the specific business domain.

Code:

```
from transformers import AutoModel, AutoTokenizer, Trainer, TrainingArguments
import datasets

model_name = 'sentence-transformers/all-MiniLM-L6-v2'
model = AutoModel.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

data = datasets.Dataset.from_pandas(your_domain_specific_data)
tokenized_data = data.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)

training_args = TrainingArguments(output_dir='./results', num_train_epochs=3, per_device_train_batch_size=8)
trainer = Trainer(model=model, args=training_args, train_dataset=tokenized_data)
trainer.train()
```

4. **Update Pinecone Embeddings**: Re-index the documents in Pinecone with the new embeddings from the fine-tuned model.

Code:

```
for doc in documents:
```

```
embedding = model.encode(doc['text'])
index.upsert([(doc['id'], embedding, doc)])
```

Technique 2: Dynamic Context Window

Implementing a dynamic context window allows the RAG model to adjust the amount of context provided to the OpenAI API based on the complexity of the query. This helps in balancing between providing enough context for accurate responses and avoiding unnecessary information overload.

1. **Query Analysis**: Analyze the query to determine its complexity. This can be done using NLP techniques to assess the length, number of entities, and specific keywords.

Code:

```
def analyze_query(query):
tokens = tokenizer.tokenize(query)
num_entities = len(set(tokens))
return num_entities
```

2. **Dynamic Context Generation**: Based on the complexity of the query, adjust the number of top-k documents retrieved and included in the context.

```
def dynamic_context(query):
num_entities = analyze_query(query)
top_k = min(10, num_entities) # Example heuristic
documents = retrieve_documents(query, top_k=top_k)
context = "\n\n".join(documents)
return context
```

3. **Optimized Answer Generation**: Use the dynamically generated context for generating the final answer.

```
def generate_answer(query):
context = dynamic_context(query)
```

```
prompt = f"Question: {query}\n\nContext:\n{context}\n\nAnswer:"
response = openai.Completion.create(
engine="text-davinci-003",
prompt=prompt,
max_tokens=150,
n=1,
stop=None,
temperature=0.7
)
return response.choices[0].text.strip()
```