Technical Report: Fine-Tuning and Embedding-Based Model Pipeline

1. Introduction

This report provides an in-depth analysis of the methodologies and decision-making processes involved in the development of a **dataset creation, embedding generation, and fine-tuning pipeline** for NLP applications. The primary objectives of this pipeline include:

1. **Dataset Creation**: Extracting, processing, and structuring data from unstructured sources.
2. **Vector Embedding Generation**: Converting text data into high-dimensional vector representations for efficient similarity retrieval.
3. **Fine-tuning & Inference**: Customizing a base model to enhance its ability to generate responses and performing evaluations to verify performance.

Each phase was designed with scalability and efficiency in mind, ensuring that the pipeline can be easily extended or modified as needed.

---

2. Dataset Creation

2.1 Data Sources and Extraction

Decision and Rationale: Why PDFs and Markdown?

PDFs were chosen because they are a common document format that contains structured textual data. However, not all PDFs allow text extraction directly (e.g., scanned documents), which necessitated the use of **OCR (Optical Character Recognition)**. Markdown files were included because they are widely used in documentation and provide structured yet lightweight text.

Why PyMuPDF over Other Libraries?

- **PyMuPDF (fitz)** was selected over libraries like PDFMiner due to its superior speed and better support for text layout preservation.

- **OCR (Tesseract + pdf2image)** was used as a fallback for scanned PDFs, since PyMuPDF cannot extract text from images.
- **Regular expressions** were used instead of pre-built text cleaners because they provide greater control over unwanted elements in extracted text.

Implementation Details:

```python
import fitz
import pytesseract
from pdf2image import convert_from_path
import markdown
import re
import glob

# Extract text from PDFs
def extract_text_from_pdf(pdf_path):
    doc = fitz.open(pdf_path)
    text = ""
    for page in doc:
        page_text = page.get_text("text")
        if page_text.strip():
            text += page_text + "\n"
        else:
            images = convert_from_path(pdf_path)
            for image in images:
                text += pytesseract.image_to_string(image) + "\n"
    return text.strip()
```

2.2 Text Chunking and Preprocessing

Decision and Rationale: Why Sentence-Based Chunking?

Transformers have a maximum input length constraint (e.g., 512 tokens for BERT, 2048 for Qwen). To optimize text representation while maintaining contextual coherence, **sentence-based chunking** was chosen over fixed-length chunking.

Why NLTK over SpaCy?

- **NLTK's sentence tokenizer** was selected because it is lightweight and does not require an external model.
- **SpaCy** was considered but omitted due to its dependency on large language models for segmentation, which was unnecessary for this case.

Implementation Details:

```python
from nltk.tokenize import sent_tokenize

def split_documents(documents, max_length=256):
    chunks = []
    for doc in documents:
        sentences = sent_tokenize(doc)
        chunk = ""
        for sentence in sentences:
            if len(chunk) + len(sentence) <= max_length:
                chunk += sentence + " "
            else:
                chunks.append(chunk.strip())
                chunk = sentence + " "
        if chunk:
            chunks.append(chunk.strip())
    return chunks
```

2.3 Question-Answer Pair Generation

Decision and Rationale: Why Automated QA Generation?

Instead of manually annotating data, an automated approach was chosen using:

1. **T5-based Question Generator (`valhalla/t5-base-qg-hl`)** – chosen over GPT-3.5 due to its **efficiency in smaller-scale applications**.
2. **DistilBERT (`distilbert-base-cased-distilled-squad`)** – selected over full BERT models for its balance between accuracy and computational efficiency.

Why Not Use GPT-4 for QA Pairing?

While GPT-4 can generate high-quality question-answer pairs, its computational cost and API latency make it unsuitable for batch processing at scale.

Implementation Details:

```python
from transformers import pipeline
question_generator = pipeline("text2text-generation",
model="valhalla/t5-base-qg-hl")
qa_extractor = pipeline("question-answering",
model="distilbert-base-cased-distilled-squad")

qa_pairs = []
for context in chunks:
    question = question_generator(f"Generate questions from:
{context}")[0]["generated_text"]
    answer = qa_extractor(question=question, context=context)
    qa_pairs.append({"context": context, "question": question,
"answer": answer["answer"]})
```

---

3. Vector Embedding Creation

Decision and Rationale: Why Use `all-mpnet-base-v2`?

This embedding model was chosen due to:

- **Better semantic similarity performance** compared to `sentence-BERT`.
- **Lower computational cost** than models like `text-embedding-ada-002` (used in OpenAI APIs).

Why Not Use FAISS Indexing?

While FAISS provides efficient nearest neighbor search, this pipeline focuses on embedding generation. FAISS integration is planned for downstream retrieval tasks.

Implementation Details:

```python
from sentence_transformers import SentenceTransformer
import numpy as np

embedding_model = SentenceTransformer("all-mpnet-base-v2")
embeddings = embedding_model.encode([chunk.page_content for chunk
in document_chunks])

# Normalize and store
embeddings_array = np.array(embeddings).astype('float32')
embeddings_array /= np.linalg.norm(embeddings_array, axis=1,
keepdims=True)
np.save("vector_database.npy", embeddings_array)
```

---

4. Model Fine-Tuning & Inference

4.1 Model Selection & Configuration

Decision and Rationale: Why Qwen2.5-3B Instead of Llama 2?

- **Qwen2.5-3B** was chosen for its superior performance in instruction-following tasks.
- **Llama 2** was considered but required more computational resources for fine-tuning.

Why LoRA (Low-Rank Adaptation)?

- **LoRA reduces the number of trainable parameters** while achieving similar fine-tuning quality.
- **Full fine-tuning would require significant VRAM**, making it infeasible for most setups.

Implementation Details:

```python
from unsloth import FastLanguageModel
model, tokenizer = FastLanguageModel.from_pretrained(
```

```
    "Qwen/Qwen2.5-3B-Instruct", max_seq_length=2048,
load_in_4bit=True
)
```

---

5. Conclusion

This report has documented the pipeline from dataset creation to fine-tuning a model using LoRA. The **key decision points** included:

- **Using PDFs & Markdown over raw text files** due to their structured format.
- **Using PyMuPDF over PDFMiner** for faster and more accurate extraction.
- **Using NLTK over SpaCy** for efficient sentence segmentation.
- **Using T5 & DistilBERT for QA generation** instead of GPT-4 for cost-efficiency.
- **Using `all-mpnet-base-v2` for embeddings** due to its superior performance over SBERT.
- **Using LoRA for fine-tuning** instead of full model training to reduce VRAM usage.

This pipeline is designed to be **modular, scalable, and optimized** for future NLP applications.