DAY – 89

9 December 2025

Below is the **complete, ready-to-paste chatbot_logic.py** with everything you asked for:

- Smart **steps vs paragraph** formatting (matches original PDF style)
- Short / long answer handling
- Follow-up resolution using session history
- Hybrid matching: TF-IDF, ML classifier, rule-based fallback, embeddings, RAG fallback
- RAG supports **multiple PDFs** (you selected option 2)
- Preserves and returns good formatted answers (avoids cuts)
- Compatible with your app.py session-based flow

How to use: replace your existing chatbot_logic.py with the file below (backup original first). Then restart your Flask app. It will auto-load PDFs listed in PDF_PATHS. You can add more PDFs to that list (the file supports multiple documents).

```
# chatbot_logic.py
import os
import re
import unicodedata
import nltk
import difflib
import PyPDF2
import requests
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics.pairwise import cosine_similarity
from dotenv import load_dotenv
from collections import Counter, deque
```

```
import logging

# ===== Config / Env =====
load_dotenv()
GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")
HF_API_KEY = os.getenv("HF_API_KEY")
HF_MODEL = os.getenv("HF_MODEL", "google/flan-t5-large")
GROQ_API_KEY = os.getenv("GROQ_API_KEY")
HF_TOKEN = os.getenv("HF_TOKEN")

DATASET_PATH = "data/dataset.txt"
# Support multiple PDFs: add more paths here if you upload more files later
PDF_PATHS = ["data/PRSC 1 (2).pdf"]
DEBUG = False

# ===== Logging =====
logger = logging.getLogger("chatbot_logic")
if DEBUG:
    logging.basicConfig(level=logging.DEBUG)
else:
    logging.basicConfig(level=logging.INFO)

# ===== NLTK =====
nltk.download('punkt', quiet=True)
nltk.download('stopwords', quiet=True)
STOPWORDS = set(stopwords.words("english"))

# ===== Helpers =====
def remove_stopwords(text):
    return " ".join([t for t in text.split() if t.lower() not in STOPWORDS])

def cheap_stem(w):
    for suf in ("ing", "ed", "es", "s"):
        if w.endswith(suf) and len(w) > len(suf) + 2:
```

```
            return w[:-len(suf)]
    return w

def normalize_text(t, lang="english"):
    t = unicodedata.normalize("NFKC", t).lower()
    # Expand meaning using synonyms for better matching
    synonyms = {
        "proforma": "form document register record",
        "performa": "form document register record",
        "register": "form document proforma record",
        "form": "proforma document register record",
        "record": "proforma form register document",
        "report": "document",
        "explain": "definition details",
        "define": "definition",
        "meaning": "definition",
    }
    for k, v in synonyms.items():
        if k in t:
            t += " " + v
    # Allow punjabi unicode range and bullet symbols
    t = re.sub(r"[^a-z0-9\s\u0A00-\u0A7F•→\-]", " ", t)
    return re.sub(r"\s+", " ", t).strip()

def tokenize_and_stem(t):
    return set(cheap_stem(w) for w in t.split() if w)

# ===== Question detection helpers =====
QUESTION_WORDS = re.compile(

r'^(?:what|how|why|when|where|which|explain|define|describe|steps|procedure|guide|workflo
w|list|features|is|are|can|should|do|does|eligib|qualif|purpose)',
    re.I
)
```

```
def _strip_leading_decorations(s):
    """Remove leading bullets, numbering, and invisible chars so regex matching works."""
    if not s:
        return s
    s = re.sub(r'^[\u200B\u200C\u200D\uFEFF]+', '', s)
    s = re.sub(r'^[\s\-\–\—\•\▪\¬\♣\→\*•\u2022\(\)\[\]\d\.\:\/\\#]+', '', s)
    s = re.sub(r'^(?:Q(?:uestion)?\s*)?\d{1,3}(?:[\.\)\-:])\s*', '', s, flags=re.I)
    return s.strip()

def score_line_question(line, next_line=""):
    if not line or not line.strip():
        return 0.0
    s = line.strip()
    s_stripped = _strip_leading_decorations(s)
    if not s_stripped:
        return 0.0
    score = 0.0
    if s_stripped.endswith('?'):
        score += 2.0
    if QUESTION_WORDS.match(s_stripped):
        score += 1.6
    if re.match(r'^(?:what\s+is|what\s+are|what\s+do|who\s+is|define\s+)', s_stripped,
flags=re.I):
        score += 1.2
    if re.match(r'^(?:Q(?:uestion)?\s*)?\d{1,3}[\.\)\-:]\s*', s, flags=re.I):
        score += 1.0
    # if next line is long and not a question, it's likely answer content
    if next_line and len(next_line.strip()) > 20 and not next_line.strip().endswith('?'):
        score += 0.2
    return score

# ===== PDF extraction =====
```

```
HEADER_FOOTER_PATTERNS = [
    r"^\s*Page\s*\|\s*\d+",
    r".*Punjab Remote Sensing Centre.*",
    r".*e-?Sinchai User Manual.*",
    r".*PISMSUser Manual.*",
    r".*\b2025\b.*",
    r"^\s*Copyright.*",
    r"^\s*All rights reserved.*",
    r"^\s*©.*",
    r'^Page\s*\|\s*\d+',
    r'^PISMSUser Manual',
    r'^Punjab Remote Sensing Centre',
]

def robust_extract_pdf_text(pdf_path):
    text = ""
    if not os.path.exists(pdf_path):
        logger.warning("PDF path not found: %s", pdf_path)
        return ""
    try:
        with open(pdf_path, "rb") as f:
            reader = PyPDF2.PdfReader(f)
            for page in reader.pages:
                ptext = page.extract_text()
                if not ptext:
                    continue
                lines = [l.rstrip() for l in ptext.splitlines() if l.strip()]
                clean_lines = []
                for l in lines:
                    if any(re.match(pat, l, flags=re.I) for pat in HEADER_FOOTER_PATTERNS):
                        continue
                    l = re.sub(r'\(?\s*Fig(?:ure)?s?[\.:]*\s*\d+([–\-–,/&\s\d]*)?\s*\)?', '', l, flags=re.I)
                    l = re.sub(r'\(\s*\d{1,4}\s*\)', '', l)
                    l = re.sub(r'\(\s*(Fig|Figure)[^)]*\)', '', l, flags=re.I)
```

```
                l = re.sub(r'\(\s*\)', '', l)
                l = re.sub(r'\[\s*\]', '', l)
                l = l.replace("¬", "• ").replace("▪", "• ").replace("♣", "• ").replace("→", "→
").replace("—", "- ")
                l = re.sub(r'[\u200B\u200C\u200D\uFEFF\uF000-\uF8FF]', '', l)
                l = re.sub(r'\s{2,}', ' ', l)
                l = l.strip(" ,;:-")
                if len(l.strip()) <= 1:
                    continue
                if re.fullmatch(r'[\(\)\[\]\s]+', l):
                    continue
                clean_lines.append(l.strip())
            if clean_lines:
                text += "\n".join(clean_lines) + "\n\n"
    except Exception as e:
        logger.exception("PyPDF2 error: %s", e)
    return text

# Build section index mapping heading -> section text
def build_section_index(raw_text):
    lines = [l.rstrip() for l in raw_text.splitlines()]
    sections = {}
    cur_head = None
    cur_lines = []
    heading_pattern = re.compile(
        r'^\s*(?:\d+(?:\.\d+){0,3}\s*[-\.\)]\s*)?([A-Z][A-Za-z0-9 &\-]{2,150}|[A-Z0-9 \-
]{3,200})(?:\s*Proforma|\s*Portal|\s*Manual|\s*Report|Proforma)?\s*$'
    )
    numbered_heading = re.compile(r'^\s*\d+(?:\.\d+){1,4}\s*(?:\.)?\s*(.+)$')
    for i, ln in enumerate(lines):
        ln_stripped = ln.strip()
        is_heading = False
        if not ln_stripped:
```

```
            if cur_head and cur_lines and i < len(lines)-1 and lines[i+1].strip():
                cur_lines.append("")
            continue
        mnum = numbered_heading.match(ln_stripped)
        mhead = heading_pattern.match(ln_stripped)
        if mnum:
            if cur_head:
                sections[cur_head.lower()] = "\n".join(cur_lines).strip()
            cur_head = mnum.group(1).strip()
            cur_lines = []
            is_heading = True
        elif mhead and len(ln_stripped.split()) <= 6:
            if cur_head:
                sections[cur_head.lower()] = "\n".join(cur_lines).strip()
            cur_head = ln_stripped
            cur_lines = []
            is_heading = True
        if not is_heading:
            if cur_head:
                cur_lines.append(ln)
            else:
                sections.setdefault("", "")
                sections[""] += ln + "\n"
    if cur_head:
        sections[cur_head.lower()] = "\n".join(cur_lines).strip()
    return sections

# parse pdf for question/answer pairs with fallback to section lookup
def parse_pdf_qa_strict(text):
    faqs = {}
    if not text or not text.strip():
        return faqs
    section_index = build_section_index(text)
    raw_lines = [l.rstrip() for l in text.splitlines()]
```

```
    i, n = 0, len(raw_lines)
    while i < n:
        line = raw_lines[i].strip()
        next_line = raw_lines[i + 1] if i + 1 < n else ""
        sc = score_line_question(line, next_line)
        if sc >= 1.0:
            q = _strip_leading_decorations(line).strip().rstrip(":").strip()
            a_lines, j = [], i + 1
            while j < n:
                ln_raw = raw_lines[j]
                ln = ln_raw.strip()
                if not ln:
                    if a_lines:
                        break
                    j += 1
                    continue
                if score_line_question(ln, raw_lines[j + 1] if j + 1 < n else "") >= 1.0:
                    break
                a_lines.append(_strip_leading_decorations(ln_raw))
                j += 1
            ans = "\n".join(a_lines).strip()
            if not ans:
                key = q.lower()
                key_try = re.sub(r'^(what|who|define|describe|explain)\s+(is|are|the|this|that)\s*', ''
key, flags=re.I).strip()
                candidates = []
                if key in section_index:
                    candidates.append(section_index[key])
                if key_try in section_index:
                    candidates.append(section_index[key_try])
                if not candidates:
                    titles = list(section_index.keys())
                    matches = difflib.get_close_matches(key_try or key, titles, n=3, cutoff=0.5)
                    for m in matches:
```

```
                candidates.append(section_index[m])
            if candidates:
                ans = max(candidates, key=lambda s: len(s)).strip()
        if q and ans:
            faqs[q.lower()] = ans
        i = j
    else:
        i += 1
  return faqs

# ===== Global data =====
faq_data = {"english": {}, "punjabi": {}}
questions, answers = [], []
classifier_pipeline, vectorizer, tfidf_matrix = None, None, None
index = {"english": []}
conversation_history = []  # global fallback memory (not session)

# ===== Load & train =====
def load_and_train():
  global faq_data, questions, answers, classifier_pipeline, vectorizer, tfidf_matrix, index
  faq_data = {"english": {}, "punjabi": {}}
  # Load dataset.txt (optional)
  if os.path.exists(DATASET_PATH):
    with open(DATASET_PATH, "r", encoding="utf-8") as f:
      lang = None
      for line in f:
        line = line.strip()
        if not line:
          continue
        if line.startswith("[") and line.endswith("]"):
          lang = line[1:-1].lower()
          continue
        if "=" in line and lang:
          q, a = line.split("=", 1)
```

```
                faq_data[lang][q.strip().lower()] = a.strip()
    # Load PDFs (multiple)
    parsed_total = 0
    for p in PDF_PATHS:
        if os.path.exists(p):
            pdf_text = robust_extract_pdf_text(p)
            parsed_from_pdf = parse_pdf_qa_strict(pdf_text)
            if parsed_from_pdf:
                faq_data["english"].update(parsed_from_pdf)
                parsed_total += len(parsed_from_pdf)
    if DEBUG:
        logger.debug("Parsed Q/A from PDFs: %s", parsed_total)
    # Build ML + TFIDF structures
    questions = list(faq_data["english"].keys())
    answers = list(faq_data["english"].values())
    if questions:
        norm_qs = [normalize_text(q) for q in questions]
        classifier_pipeline = make_pipeline(TfidfVectorizer(),
LogisticRegression(max_iter=2000))
        classifier_pipeline.fit(norm_qs, list(range(len(questions))))
        vectorizer = TfidfVectorizer()
        tfidf_matrix = vectorizer.fit_transform(norm_qs)
        index["english"] = [{"q": q, "nq": normalize_text(q), "tokens":
tokenize_and_stem(normalize_text(q)), "a": a} for q, a in faq_data["english"].items()]
    logger.info("Loaded %d Q/A from dataset + PDFs", len(questions))

# ===== Matching modules =====
def classify_intent(user_norm):
    if not classifier_pipeline:
        return None
    try:
        pred = classifier_pipeline.predict([user_norm])[0]
        prob = max(classifier_pipeline.predict_proba([user_norm])[0])
        return answers[pred] if prob > 0.45 else None
```

```
    except Exception:
        return None

def ml_best_match(user_norm):
    if tfidf_matrix is None or vectorizer is None:
        return None
    sims = cosine_similarity(vectorizer.transform([user_norm]), tfidf_matrix)[0]
    best_idx = sims.argmax()
    return answers[best_idx] if sims[best_idx] > 0.35 else None

def rule_based_match(user_norm, user_tokens):
    if not index["english"]:
        return None
    best_score, best_ans = 0.0, None
    for e in index["english"]:
        union = e["tokens"] | user_tokens
        if not union:
            continue
        score = len(e["tokens"] & user_tokens) / len(union)
        if score > best_score:
            best_score, best_ans = score, e["a"]
    return best_ans if best_score >= 0.22 else None

def format_answer(ans):
    if not ans:
        return ans
    lines = [l.rstrip() for l in ans.splitlines()]
    cleaned = []
    for l in lines:
        if re.fullmatch(r'[\u200B\u200C\u200D\uFEFF\s]*', l):
            continue
        cleaned.append(l)
    return "\n".join(cleaned)
```

```
def force_short_answer(text):
    if not text:
        return text
    lines = [l for l in text.split("\n") if l.strip()]
    if len(lines) <= 4:
        return text
    max_lines = max(3, min(5, len(lines) // 3))
    short = lines[:max_lines]
    return "\n".join(short)

# ===== Follow-up handling (history-aware) =====
def handle_follow_up(query, history):
    follow_pattern =
r'\b(this|that|it|those|these|above|previous|earlier|mentioned|same|topic|matter|explain
again|brief|short|again)\b'
    if not re.search(follow_pattern, query, re.I):
        return query
    last_topic = None
    if history:
        for entry in reversed(history):
            if not isinstance(entry, (list, tuple)) or len(entry) == 0:
                continue
            past_user = entry[0].strip()
            if not past_user:
                continue
            if len(past_user.split()) < 3:
                continue
            if re.search(follow_pattern, past_user, re.I):
                continue
            last_topic = past_user
            break
    if not last_topic:
        if history:
            entry = history[-1]
```

```
            if isinstance(entry, (list, tuple)) and len(entry) >= 1:
                last_topic = entry[0]
    if not last_topic:
        return query
    rewritten = f"Briefly explain: {last_topic}"
    return rewritten

# ===== Translation helper (very small, fallback) =====
def translate_text(text, target_lang="pa"):
    if not text or target_lang not in ("en", "pa"):
        return text
    try:
        res = requests.get(
            "https://translate.googleapis.com/translate_a/single",
            params={"client": "gtx", "sl": "auto", "tl": target_lang, "dt": "t", "q": text},
            timeout=10,
        )
        if res.status_code == 200:
            return "".join([part[0] for part in res.json()[0]]).strip()
    except Exception as e:
        if DEBUG:
            logger.exception("Translation error: %s", e)
    return text

# ===== RAG (optional) - multi-PDF aware =====
try:
    from langchain_groq import ChatGroq
    from langchain_chroma import Chroma
    from langchain_huggingface import HuggingFaceEmbeddings
    from langchain_text_splitters import RecursiveCharacterTextSplitter
    from langchain_community.document_loaders import PyPDFLoader
    from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
    from langchain.chains import create_history_aware_retriever, create_retrieval_chain
    from langchain.chains.combine_documents import create_stuff_documents_chain
```

```
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vectorstore, retriever = None, None

def init_rag():
    global vectorstore, retriever
    # load all PDFs as documents
    docs = []
    for p in PDF_PATHS:
        if not os.path.exists(p):
            logger.warning("RAG: PDF not found: %s", p)
            continue
        loader = PyPDFLoader(p)
        try:
            docs += loader.load()
        except Exception:
            # fallback: split by pages manually
            pass
    if not docs:
        logger.warning("RAG: No documents loaded.")
        return
    splitter = RecursiveCharacterTextSplitter(chunk_size=4000, chunk_overlap=400)
    chunks = splitter.split_documents(docs)
    vectorstore = Chroma.from_documents(documents=chunks, embedding=embeddings)
    retriever = vectorstore.as_retriever(search_kwargs={"k": 4})
    logger.info("RAG retriever initialized successfully.")

def get_rag_response(query, chat_history=None):
    global retriever
    if retriever is None:
        try:
            init_rag()
```

```
        except Exception as e:
            logger.exception("RAG init error: %s", e)
    if retriever is None:
        return None
    llm = ChatGroq(groq_api_key=GROQ_API_KEY, model_name="Gemma2-9b-It",
temperature=0.3)
    contextualize_q_system_prompt = (
        "Given a chat history and the latest user question which might reference context, "
        "formulate a standalone question that can be understood without the chat history. "
        "Do NOT answer the question, just rewrite if needed."
    )
    contextualize_q_prompt = ChatPromptTemplate.from_messages([
        ("system", contextualize_q_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}")
    ])
    history_aware_retriever = create_history_aware_retriever(llm, retriever,
contextualize_q_prompt)
    system_prompt = (
        "You are a helpful assistant for Punjab Remote Sensing Centre (PRSC) queries. "
        "Use the provided PDF context to answer precisely. If you don't know, politely say
so.\n\n{context}"
    )
    qa_prompt = ChatPromptTemplate.from_messages([
        ("system", system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}")
    ])
    qa_chain = create_stuff_documents_chain(llm, qa_prompt)
    rag_chain = create_retrieval_chain(history_aware_retriever, qa_chain)
    session_history = ChatMessageHistory()
    if chat_history:
        for u, b in chat_history[-6:]:
            session_history.add_user_message(u)
```

```
            session_history.add_ai_message(b)
        conversational_chain = RunnableWithMessageHistory(
            rag_chain,
            lambda _: session_history,
            input_messages_key="input",
            history_messages_key="chat_history",
            output_messages_key="answer"
        )
        try:
            response = conversational_chain.invoke(
                {"input": query},
                config={"configurable": {"session_id": "default_session"}}
            )
            return response["answer"]
        except Exception:
            return None

except Exception:
    if DEBUG:
        logger.exception("□ RAG dependencies missing, fallback to FAQ model.")
    # keep get_rag_response undefined fallback to None

# ===== Smart formatting for steps vs paragraph =====
def extract_steps_and_paragraph(answer_text):
    """
    Returns a tuple (paragraph_summary_or_none, steps_list_or_none)
    - If answer contains clear bullets/numbered lines, return steps_list (list of lines).
    - If answer is paragraph-style, return paragraph string and None.
    - If both exist, return both.
    """
    if not answer_text:
        return None, None
    ans = answer_text.strip()
    lines = [l.strip() for l in ans.splitlines() if l.strip()]
```

```
    # Detect bullets/numbered
    steps = []
    for l in lines:
        if re.match(r'^(?:\d+\.\s+|\d+\)\s+|•\s+|\u2022\s+|\- \s*|•)', l) or re.match(r'^(?:¬|▪|♣)\s*',
l):
            steps.append(re.sub(r'^[\d\.\)\s\-\•\u2022\¬\▪\♣]+', '', l).strip())
        elif re.match(r'^(?:Step\s*\d+[:\.\-\s])', l, flags=re.I):
            steps.append(re.sub(r'^(?:Step\s*\d+[:\.\-\s])', '', l, flags=re.I).strip())
    # If steps found, return them. Else, detect paragraph blocks (one or more sentences)
    if steps:
        # sometimes paragraphs before steps act as intro; capture first paragraph if it's short
        first_par = None
        para_text = "\n\n".join(re.split(r'\n\s*\n', ans))
        # take first paragraph if more than one sentence and not just a header
        paras = [p.strip() for p in re.split(r'\n\s*\n', ans) if p.strip()]
        if paras:
            # if first para length > 30 characters and not just a heading, keep as intro
            if len(paras[0]) > 30 and not re.match(r'^[A-Z\s]{2,40}$', paras[0]):
                first_par = paras[0]
        return first_par, steps
    else:
        # no explicit steps; return paragraph (clean combined)
        # collapse many short lines into paragraph
        combined = " ".join(lines)
        # If combined contains multiple sentences, keep as paragraph.
        return combined, None


def format_smart_answer_for_steps_style(raw_answer, user_wants_steps=False):
    """
    If user wants steps: prefer steps extraction. If steps not found and answer paragraph exists,
return paragraph.
    If user asked general question (not steps), return answer as-is (but cleaned).
    """
```

```
    if not raw_answer:
        return None
    paragraph, steps = extract_steps_and_paragraph(raw_answer)
    if user_wants_steps:
        if steps:
            # prefer numbered if many steps
            if len(steps) >= 2:
                numbered = []
                for i, s in enumerate(steps, 1):
                    numbered.append(f"{i}. {s}")
                if paragraph:
                    return paragraph + "\n\nSteps:\n" + "\n".join(numbered)
                return "Steps:\n" + "\n".join(numbered)
            else:
                # single-line step, return as bullet
                if paragraph:
                    return paragraph + "\n\nSteps:\n• " + "\n• ".join(steps)
                return "Steps:\n• " + "\n• ".join(steps)
        else:
            # no explicit steps found — return short paragraph summary (or whole answer)
            # keep it paragraph style
            return paragraph or raw_answer
    else:
        # user did not specifically ask for steps — return clean formatted answer
        # if both exist return paragraph + steps
        if steps and paragraph:
            numbered = []
            for i, s in enumerate(steps, 1):
                numbered.append(f"{i}. {s}")
            return paragraph + "\n\nSteps:\n" + "\n".join(numbered)
        if steps:
            numbered = []
            for i, s in enumerate(steps, 1):
                numbered.append(f"{i}. {s}")
```

```
            return "\n".join(numbered)
        return paragraph or raw_answer

# ===== Check reload needed for app.py compatibility =====
def check_reload_needed():
    # We don't eagerly reload in this file; app.py can call load_and_train on startup
    return False

# ===== Main response function =====
def get_response(user_input, lang=None, history=None):
    """
    Hybrid QA: uses history to rewrite follow-ups, then TF-IDF / embeddings / RAG to answer.
    Supports short / long answers, steps extraction, and multi-PDF RAG.
    history: list of (user_text, bot_text) tuples (session)
    """
    global conversation_history, vectorizer, tfidf_matrix, questions, answers

    if history is None:
        history = []

    # Ensure data loaded
    if not questions:
        load_and_train()

    if isinstance(user_input, dict):
        user_input = user_input.get("text", "")

    user_input = user_input.strip()
    if not user_input:
        return "□ Please enter a message."

    # STEP 1: follow-up handling
    rewritten = handle_follow_up(user_input, history)
```

```
    query = rewritten

    # STEP 2: language detection / normalization
    lang = lang or ("punjabi" if re.search(r'[\u0A00-\u0A7F]', query) else "english")
    query_en = query if lang == "english" else translate_text(query, "en")
    user_norm = normalize_text(query_en)
    user_tokens = tokenize_and_stem(user_norm)

    # Intent detection: steps/short/long
    intent_steps = False
    intent_short = False
    intent_long = False
    if re.search(r'\b(step|steps|procedure|workflow|process|how to|howdo)\b', query_en, re.I):
        intent_steps = True
    if re.search(r'\b(short|brief|in short|summary|chota|one line|one sentence)\b', query_en,
re.I):
        intent_short = True
    if re.search(r'\b(detail|full|long|explain properly|poora|detailed)\b', query_en, re.I):
        intent_long = True

    best_answer, best_score = None, 0.0

    # TF-IDF / cosine similarity
    try:
        if vectorizer is not None and tfidf_matrix is not None and questions:
            qv = vectorizer.transform([user_norm])
            sims = cosine_similarity(qv, tfidf_matrix)[0]
            idx = int(sims.argmax())
            score = float(sims[idx])
            if score > best_score:
                best_answer, best_score = answers[idx], score
    except Exception:
        logger.exception("TF-IDF match error")
```

```
    # Embedding-based matching (optional)
    try:
        from langchain_community.embeddings import HuggingFaceEmbeddings
        embedder = HuggingFaceEmbeddings(model_name="sentence-transformers/all-
MiniLM-L6-v2")
        question_embeddings = embedder.embed_documents(questions)
        query_emb = embedder.embed_query(user_norm)
        emb_scores = cosine_similarity([query_emb], question_embeddings)[0]
        idx = int(emb_scores.argmax())
        score = float(emb_scores[idx])
        if score > best_score:
            best_answer, best_score = answers[idx], score
    except Exception:
        # no-op fallback
        pass

    # Rule-based fallback
    if not best_answer:
        best_answer = rule_based_match(user_norm, user_tokens)

    # RAG fallback if still nothing or low confidence
    rag_ans = None
    if (not best_answer) or (best_score < 0.4):
        try:
            rag_ans = get_rag_response(query, chat_history=history) if 'get_rag_response' in
globals() else None
            if rag_ans and not rag_ans.lower().startswith(("sorry", "could not", "can't", "cannot",
"i don't")):
                # use RAG only if it returned something reasonable
                best_answer = rag_ans
        except Exception:
            pass

    # Intent-driven short/long handling
```

```
    if intent_short and best_answer:
        best_answer = force_short_answer(best_answer)
    # If user explicitly asked long, keep as-is (no shortening)
    # else default keep best_answer

    # Smart formatting for steps vs paragraph
    if intent_steps:
        formatted = format_smart_answer_for_steps_style(format_answer(best_answer),
user_wants_steps=True)
    else:
        formatted = format_smart_answer_for_steps_style(format_answer(best_answer),
user_wants_steps=False)

    # If still empty, give a fallback help message
    if not formatted:
        # small helpful fallback: try to find a match phrase
        suggestion = None
        if questions:
            # show top 3 similar question titles to help user
            try:
                sims = cosine_similarity(vectorizer.transform([user_norm]), tfidf_matrix)[0]
                topk = sims.argsort()[-3:][::-1]
                suggestion = "I couldn't find an exact answer. Did you mean:\n" + "\n".join(f"-
{questions[i]}" for i in topk)
            except Exception:
                suggestion = "I couldn't find an exact answer in the documents. Try rephrasing or
ask for definition/steps."
        else:
            suggestion = "I couldn't find an answer. Try rephrasing the question."
        formatted = suggestion

    # Save to history (session provided by app.py)
    try:
        # history is passed as list; append (user_input, formatted)
```

```
        if isinstance(history, list):
            history.append((user_input, formatted))
            # keep last 50 messages
            while len(history) > 200:
                history.pop(0)
    except Exception:
        pass

    return formatted

# end of file
```

Quick notes & next steps

- Replace your current chatbot_logic.py with the file above.
- Add more PDFs to PDF_PATHS (e.g., "data/another_doc.pdf") — RAG init will load them.
- If you want numbered steps by default or a different bullet style, tell me and I'll adjust formatting.
- If your server environment lacks langchain/RAG libs, the file gracefully falls back to TF-IDF & rule-based matching (still improved).
- If you want, I can also patch your app.py to add an explicit endpoint to force load_and_train() and init_rag() during startup.