DAY – 74, 75

17, 18 November 2025

Quick summary of the problems

1. **Loss of signal from over-aggressive normalization** — you remove stopwords *before* TF-IDF and classifier training. For short questions that rely on function words ("is", "how", "can"), that kills the signal and makes matching poor.
2. **Not using classifier / ml_best_match in get_response** — you defined classify_intent() and ml_best_match() but then rely mainly on TF-IDF/tf-idf thresholds and RAG fallback. The classifier/ML fallback should be integrated early.
3. **Answer-length not controlled** — no logic to choose a short/medium/long reply based on question brevity or user preference.
4. **Question detection regex/thresholds** — can be improved (short conversational queries like "eligible?" or "how to apply" may not match).
5. **Thresholds tuned a bit strict/loose** — that causes incorrect fallbacks to "Sorry I don't know".
6. **Small robustness issues** — e.g. the pipeline and separate vectorizer sometimes produce mismatched vocabularies or insufficient n-gram features.

Drop-in patch

Replace (or add/overwrite) the functions below in your module. I kept changes minimal and explicit so you can drop them into your existing file.

```
# ---------- PATCH START ----------

# Replace normalize_text with two modes: keep_stopwords flag
def normalize_text(t, lang="english", keep_stopwords=False):
    t = unicodedata.normalize("NFKC", t)
    if lang == "english":
        t = t.lower()
        # Option: keep stopwords for short queries / vectorizer training
```

```
        if not keep_stopwords:
            t = remove_stopwords(t)
        t = re.sub(r'[^a-z0-9\s\?]', ' ', t)  # keep question mark for detection
    elif lang == "punjabi":
        t = re.sub(r'[^\u0A00-\u0A7F\s;|]', ' ', t)
    return re.sub(r"\s+", " ", t).strip()

# Improve question detection (broader)
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|eligible|eligib|qualif|purpose|tell|show|help)\b',
    re.I
)

def score_line_question(line, next_line=""):
    if not line or not line.strip():
        return 0.0
    s = line.strip()
    score = 0.0
    if s.endswith('?'):
        score += 2.0
    if QUESTION_WORDS.match(s):
        score += 1.6
    # short imperative like "Eligibility" or "Procedure" often indicate Q
    if re.match(r'^\s*(?:Eligibility|Procedure|Steps|How to|How do|Requirements)\b', s,
flags=re.I):
        score += 1.4
    # numbered question lines
    if re.match(r'^\s*(?:Q(?:uestion)?\s*)?\d{1,3}(?:[\.\)\-:])\s*', s, flags=re.I):
        score += 1.2
    return score
```

```
# Improved load_and_train: keep stopwords for vectorizer, use ngrams, store vectorizer used for classifier
def load_and_train():
    global faq_data, questions, answers, classifier_pipeline, vectorizer, tfidf_matrix, index
    faq_data = {"english": {}, "punjabi": {}}

    # Load dataset.txt (unchanged)
    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 PDF Q/A
    if os.path.exists(PDF_PATH):
        pdf_text = robust_extract_pdf_text(PDF_PATH)
        parsed_from_pdf = parse_pdf_qa_strict(pdf_text)
        if parsed_from_pdf:
            faq_data["english"].update(parsed_from_pdf)

    # Prepare questions/answers lists
    questions = list(faq_data["english"].keys())
    answers = list(faq_data["english"].values())

    if questions:
```

```
        # Normalize for pipeline training but keep stopwords for vectorizer (so short queries retain signal)
        norm_qs_for_pipe = [normalize_text(q, keep_stopwords=True) for q in questions]  # keep stopwords
        # classifier pipeline with its own TfidfVectorizer tuned for short queries (use unigrams+bigrams)
        classifier_pipeline = make_pipeline(TfidfVectorizer(ngram_range=(1,2), min_df=1), LogisticRegression(max_iter=2000))
        # labels are indexes
        classifier_pipeline.fit(norm_qs_for_pipe, list(range(len(questions))))

        # separate vectorizer + tfidf matrix for cosine similarity search (same preprocessing)
        vectorizer = TfidfVectorizer(ngram_range=(1,2))
        tfidf_matrix = vectorizer.fit_transform(norm_qs_for_pipe)

        # index for rule-based token overlap (tokenize normalized text but keep useful words)
        index["english"] = [{"q": q, "nq": normalize_text(q, keep_stopwords=True), "tokens": tokenize_and_stem(normalize_text(q, keep_stopwords=True)), "a": a} for q, a in faq_data["english"].items()]

    print(f"Loaded {len(questions)} Q/A from dataset + PDF")

# Answer length control helper
def adjust_answer_length(ans, user_input):
    if not ans:
        return ans
    # decide desired verbosity by question length and explicit cues
    q_words = len(user_input.split())
    want_short = q_words <= 6 or re.search(r'\b(brief|short|tl;dr|one-liner)\b', user_input, re.I)
    want_long = q_words > 25 or re.search(r'\b(detailed|in detail|explain|elaborate|describe|full)\b', user_input, re.I)

    # split into sentences (naive)
    sents = re.split(r'(?<=[.!?])\s+', ans.strip())
```

```
    if want_short:
        # return first sentence or first 25 words
        first = sents[0] if sents else ans
        if len(first.split()) > 40:
            return " ".join(first.split()[:40]) + "..."
        return first
    if want_long:
        return ans  # full answer
    # medium default: first 2 sentences or up to 80 words
    if len(sents) >= 2:
        medium = " ".join(sents[:2])
    else:
        medium = ans
    if len(medium.split()) > 80:
        return " ".join(medium.split()[:80]) + "..."
    return medium

# Integrate classifier + ml_best_match into get_response and use adjust_answer_length
def get_response(user_input, lang=None, history=None):
    global conversation_history

    if history is None:
        history = []

    # ensure data loaded
    if not questions:
        load_and_train()

    lang = lang or ("punjabi" if re.search(r'[\u0A00-\u0A7F]', user_input) else "english")
    query = user_input if lang == "english" else translate_text(user_input, "en")

    # Normalize for matching: use keep_stopwords=True here to preserve short-question cues
    user_norm_for_search = normalize_text(query, keep_stopwords=True)
    user_tokens = tokenize_and_stem(user_norm_for_search)
```

```
# Try classifier intent first (if available)
best_answer = None
best_score = 0.0
try:
    if classifier_pipeline:
        pred_ans = classify_intent(user_norm_for_search)
        if pred_ans:
            best_answer = pred_ans
            best_score = 0.95
except Exception as e:
    if DEBUG:
        print("Classifier error:", e)

# TF-IDF cosine fallback (use vectorizer trained above)
if not best_answer or best_score < 0.6:
    try:
        if vectorizer is not None and tfidf_matrix is not None and questions:
            qv = vectorizer.transform([user_norm_for_search])
            sims = cosine_similarity(qv, tfidf_matrix)[0]
            idx = int(sims.argmax())
            score = float(sims[idx])
            if score > best_score and score > 0.35:
                best_answer = answers[idx]
                best_score = score
    except Exception as e:
        if DEBUG:
            print("TF-IDF error:", e)

# ml_best_match (cosine on same tfidf_matrix) as additional fallback
if not best_answer or best_score < 0.45:
    try:
        ml_ans = ml_best_match(user_norm_for_search)
        if ml_ans:
```

```
                best_answer = ml_ans
                best_score = max(best_score, 0.5)
        except Exception as e:
            if DEBUG:
                print("ml_best_match error:", e)

    # Rule-based fuzzy match
    if not best_answer or best_score < 0.5:
        rb = rule_based_match(user_norm_for_search, user_tokens)
        if rb:
            best_answer = rb
            best_score = max(best_score, 0.45)

    # RAG fallback last
    if not best_answer or best_score < 0.6:
        try:
            rag_input = f"{query}\nContext: {' '.join([u for u, _ in history[-3:]])}"
            rag_ans = get_rag_response(rag_input)
            if rag_ans and not rag_ans.lower().startswith(("sorry", "couldn't", "i cannot")):
                best_answer = rag_ans
                best_score = 0.9
        except Exception as e:
            if DEBUG:
                print("RAG error:", e)

    # Default fallback
    if not best_answer:
        best_answer = "Sorry, I don’t have an answer for that yet. Could you specify the topic again?"

    # Adjust verbosity to match question length & user cues
    best_answer = adjust_answer_length(best_answer, user_input)

    # Translate back if Punjabi
```

```
    if lang == "punjabi":
        best_answer = translate_text(best_answer, "pa")

    # update history and return formatted
    history.append((user_input, best_answer))
    if len(history) > 50:
        history[:] = history[-50:]

    return format_answer(best_answer)

# ---------- PATCH END ----------
```

Why these changes help

- **Keep stopwords for vectorizer/classifier**: short questions rely on words like *is*, *how*, *can*. If you strip them before building TF-IDF, the model has less to match against. Using keep_stopwords=True preserves that signal for the vectorizer and classifier, while you can still remove them for other normalization tasks if needed.
- **Use classifier first**: if you have labeled Q/A pairs, a classifier often gives very precise matches for known questions. Using it early avoids incorrect cosine matches.
- **Use n-grams (1,2)** in TF-IDF: many useful cues are bigrams (e.g., “how to”, “apply online”), which improves cosine similarity on short queries.
- **Answer length control**: adjust_answer_length() inspects question length and explicit user words like “brief” or “detailed” to return short/medium/long answers.
- **Better thresholds and fallbacks**: making classifier and RAG stronger fallbacks reduces random “I don’t know” replies.

Additional tips & next steps

1. **Small dataset → increase examples**: if your dataset.txt only has a few Q/A pairs, ML models will be brittle. Add paraphrases for common questions (e.g., “How to apply?”, “How can I apply?”, “Application procedure” → same Q).

2. **Enable DEBUG during testing**: set DEBUG=True to log classifier/TF-IDF scores so you can tune thresholds.
3. **User-specified verbosity**: consider adding a UI toggle or parse explicit tokens like [short] or [detailed] in the query to give exact control.
4. **Unit tests**: create a small test set of user queries + expected short/long answers and run locally to measure correctness.
5. **If RAG is available, prefer RAG for novel queries** — your RAG code block is good; keep it initialized and monitor for errors (missing PDF or langchain deps).