```python
import numpy as np
import pandas as pd
from collections import Counter

import streamlit as st

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    classification_report,
    accuracy_score,
    confusion_matrix,
)
from sklearn.metrics.pairwise import cosine_similarity

from sentence_transformers import SentenceTransformer

import matplotlib.pyplot as plt

from mundartchat_data import (
    RANDOM_STATE,
    DATA_CSV_BASE,
    DATA_CSV_CHATPAIRS,
    LABEL_ORDER,
    TOKEN_PATTERN,
    preprocess_text_chat,
    build_base_dataset,
    build_chatpairs_dataset,
)

# ============================================================
# Globale Config
# ============================================================

SBERT_MODEL_NAME = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
BATCH_SIZE = 32
```

```python
# ============================================================
# Daten laden / erstellen
# ============================================================

@st.cache_data
def load_datasets():
    """Basis- und Chatpair-Datensätze laden, bei Bedarf neu erstellen."""
    try:
        base_df = pd.read_csv(DATA_CSV_BASE)
    except FileNotFoundError:
        base_df = build_base_dataset()

    if "text_clean" not in base_df.columns:
        base_df["text_clean"] = base_df["text"].astype(str).apply(preprocess_text_chat)

    try:
        resp_df = pd.read_csv(DATA_CSV_CHATPAIRS)
    except FileNotFoundError:
        resp_df = build_chatpairs_dataset()

    return base_df, resp_df


# ============================================================
# N-Gramm Language Model
# ============================================================

def tokenize_for_lm(text: str):
    clean = preprocess_text_chat(text)
    if not clean:
        return []
    return clean.split()


def train_ngram_lm(texts, n_max: int = 3):
    ngram_counts = {n: Counter() for n in range(1, n_max + 1)}
    for t in texts:
        toks = tokenize_for_lm(t)
        if not toks:
            continue
```

```
        toks = ["<s>"] + toks + ["</s>"]
        for n in range(1, n_max + 1):
            if len(toks) < n:
                continue
            for i in range(len(toks) - n + 1):
                ngram = tuple(toks[i:i + n])
                ngram_counts[n][ngram] += 1

    def lm_analyzer(text: str):
        return tokenize_for_lm(text)

    return ngram_counts, lm_analyzer


def _is_good_token(tok: str) -> bool:
    if tok in ("<s>", "</s>"):
        return False
    if tok.startswith("<") and tok.endswith(">"):
        return False
    if len(tok) < 2:
        return False
    return True


def get_top_ngrams(ngram_counts, n: int, topk: int = 20) -> pd.DataFrame:
    if n not in ngram_counts:
        return pd.DataFrame(columns=["ngram", "count", "rel_freq"])

    counter = ngram_counts[n]
    if not counter:
        return pd.DataFrame(columns=["ngram", "count", "rel_freq"])

    total = sum(counter.values())
    rows = []
    for ng, cnt in counter.most_common(topk * 3):
        text = " ".join(ng)

        if "<s>" in ng or "</s>" in ng:
            continue
        if text.strip() in ("<s>", "</s>"):
```

```
            continue

        rows.append({
            "ngram": text,
            "count": cnt,
            "rel_freq": round(cnt / total, 3),
        })

    return pd.DataFrame(rows[:topk])


def next_word_candidates(prefix, ngram_counts, analyzer, n_max=3, topk=5):
    toks = analyzer(preprocess_text_chat(prefix))
    backoff_level = None

    for n in range(n_max, 0, -1):
        if n == 1:
            total = sum(
                cnt
                for (tok,), cnt in ngram_counts[1].items()
                if _is_good_token(tok)
            )
            if total == 0:
                continue
            candidates = [
                (tok, cnt)
                for (tok,), cnt in ngram_counts[1].most_common()
                if _is_good_token(tok)
            ][:topk]
            backoff_level = 1
            probs = [(w, c / float(total)) for w, c in candidates]
            return probs, backoff_level

        if len(toks) < n - 1:
            continue

        context = tuple(toks[-(n - 1):])
        candidates = []
        for ng, cnt in ngram_counts[n].items():
            if ng[:-1] == context:
```

```
                w = ng[-1]
                if _is_good_token(w):
                    candidates.append((w, cnt))

        if candidates:
            candidates.sort(key=lambda x: x[1], reverse=True)
            candidates = candidates[:topk]
            total_cnt = sum(c for _, c in candidates)
            backoff_level = n
            probs = [(w, c / float(total_cnt)) for w, c in candidates]
            return probs, backoff_level

    return [], None


# ============================================================
# Modelle trainieren
# ============================================================

@st.cache_resource
def train_all_models(base_df: pd.DataFrame, resp_df: pd.DataFrame):
    """Trainiert Klassifikationsmodelle, LM und Retrieval-Komponenten."""
    # Split
    X_tr_clean, X_te_clean, y_train, y_test = train_test_split(
        base_df["text_clean"],
        base_df["label"],
        test_size=0.25,
        random_state=RANDOM_STATE,
        stratify=base_df["label"],
    )

    X_tr_raw = base_df.loc[X_tr_clean.index, "text"]
    X_te_raw = base_df.loc[X_te_clean.index, "text"]

    # BoW / TF-IDF
    bow = Pipeline([
        ("vec", CountVectorizer(token_pattern=TOKEN_PATTERN, min_df=2)),
        ("clf", LogisticRegression(max_iter=1000, random_state=RANDOM_STATE)),
    ]).fit(X_tr_clean, y_train)
```

```
tfidf = Pipeline([
    ("vec", TfidfVectorizer(ngram_range=(1, 2), token_pattern=TOKEN_PATTERN, min_df=2)),
    ("clf", LogisticRegression(max_iter=1000, random_state=RANDOM_STATE)),
]).fit(X_tr_clean, y_train)

# SBERT + LR
sbert_model = SentenceTransformer(SBERT_MODEL_NAME)
X_list_tr = pd.Series(X_tr_raw).astype(str).tolist()
emb_train = sbert_model.encode(
    X_list_tr,
    convert_to_numpy=True,
    batch_size=BATCH_SIZE,
)
sbert_clf = LogisticRegression(
    max_iter=1000,
    random_state=RANDOM_STATE,
).fit(emb_train, y_train)

# Evaluation
X_list_te = pd.Series(X_te_raw).astype(str).tolist()
emb_test = sbert_model.encode(
    X_list_te,
    convert_to_numpy=True,
    batch_size=BATCH_SIZE,
)
y_pred_sbert = sbert_clf.predict(emb_test)

eval_info = {}

# BoW
y_pred_bow = bow.predict(X_te_clean)
report_bow = classification_report(
    y_test, y_pred_bow, digits=3, output_dict=True
)
eval_info["bow"] = {
    "report": report_bow,
    "accuracy": accuracy_score(y_test, y_pred_bow),
}

# TF-IDF
```

```
y_pred_tfidf = tfidf.predict(X_te_clean)
report_tfidf = classification_report(
    y_test, y_pred_tfidf, digits=3, output_dict=True
)
eval_info["tfidf"] = {
    "report": report_tfidf,
    "accuracy": accuracy_score(y_test, y_pred_tfidf),
}

# SBERT
report_sbert = classification_report(
    y_test, y_pred_sbert, digits=3, output_dict=True
)
cm_sbert = confusion_matrix(y_test, y_pred_sbert, labels=LABEL_ORDER)
cm_df = pd.DataFrame(
    cm_sbert,
    index=[f"true_{l}" for l in LABEL_ORDER],
    columns=[f"pred_{l}" for l in LABEL_ORDER],
)
eval_info["sbert"] = {
    "report": report_sbert,
    "accuracy": accuracy_score(y_test, y_pred_sbert),
    "confusion_matrix": cm_df,
}

# N-Gramm LM
ngram_counts, lm_analyzer = train_ngram_lm(base_df["text_clean"], n_max=3)

# Antwort-Retrieval
resp_df = resp_df[
    resp_df["answer_mundart"].astype(str).str.len() > 0
].reset_index(drop=True)

resp_emb = sbert_model.encode(
    resp_df["user_text"].astype(str).tolist(),
    convert_to_numpy=True,
    batch_size=BATCH_SIZE,
)

models = {
```

```
        "bow": bow,
        "tfidf": tfidf,
        "sbert_model": sbert_model,
        "sbert_clf": sbert_clf,
        "ngram_counts": ngram_counts,
        "lm_analyzer": lm_analyzer,
        "resp_df": resp_df,
        "resp_emb": resp_emb,
        "eval_info": eval_info,
    }
    return models


# ============================================================
# Inferenz-Helper (Klassifikation & Retrieval)
# ============================================================

def probs_pipeline(model, texts):
    vec = model.named_steps["vec"]
    clf = model.named_steps["clf"]
    X = vec.transform(texts)
    P = clf.predict_proba(X)
    cls = clf.classes_
    out = []
    for p in P:
        out.append({c: float(p[i]) for i, c in enumerate(cls)})
    return out


def sbert_predict(models, texts):
    sbert_model = models["sbert_model"]
    sbert_clf = models["sbert_clf"]
    X = pd.Series(texts).astype(str).tolist()
    emb = sbert_model.encode(
        X,
        convert_to_numpy=True,
        batch_size=BATCH_SIZE,
    )
    return sbert_clf.predict(emb)
```

```python
def sbert_predict_proba(models, texts):
    sbert_model = models["sbert_model"]
    sbert_clf = models["sbert_clf"]
    X = pd.Series(texts).astype(str).tolist()
    emb = sbert_model.encode(
        X,
        convert_to_numpy=True,
        batch_size=BATCH_SIZE,
    )
    P = sbert_clf.predict_proba(emb)
    cls = sbert_clf.classes_
    out = []
    for p in P:
        out.append({c: float(p[i]) for i, c in enumerate(cls)})
    return out


def classify_text(models, raw_inp: str):
    clean_inp = preprocess_text_chat(raw_inp)

    bow = models["bow"]
    tfidf = models["tfidf"]

    bow_pred = bow.predict([clean_inp])[0]
    tfidf_pred = tfidf.predict([clean_inp])[0]
    sbert_pred = sbert_predict(models, [raw_inp])[0]

    bow_probs = probs_pipeline(bow, [clean_inp])[0]
    tfidf_probs = probs_pipeline(tfidf, [clean_inp])[0]
    sbert_probs = sbert_predict_proba(models, [raw_inp])[0]

    return {
        "bow_pred": bow_pred,
        "tfidf_pred": tfidf_pred,
        "sbert_pred": sbert_pred,
        "bow_probs": bow_probs,
        "tfidf_probs": tfidf_probs,
        "sbert_probs": sbert_probs,
    }
```

```python
def generate_answer(
    models,
    user_text: str,
    predicted_label: str | None = None,
    topk: int = 5,
    min_sim: float = 0.2,
):
    resp_df = models["resp_df"]
    resp_emb = models["resp_emb"]
    sbert_model = models["sbert_model"]

    if len(resp_df) == 0:
        return None, None

    q_emb = sbert_model.encode([user_text], convert_to_numpy=True)
    sims = cosine_similarity(q_emb, resp_emb)[0]

    candidate_idx = np.arange(len(resp_df))
    if predicted_label is not None and "label" in resp_df.columns:
        mask = (resp_df["label"].astype(str) == str(predicted_label))
        if mask.any():
            candidate_idx = np.where(mask)[0]

    if len(candidate_idx) == 0:
        candidate_idx = np.arange(len(resp_df))

    sims_sub = sims[candidate_idx]
    top_local = np.argsort(-sims_sub)[:min(topk, len(sims_sub))]

    best_local_idx = top_local[0]
    best_idx = candidate_idx[best_local_idx]
    best_sim = float(sims_sub[best_local_idx])
    best_answer = resp_df.iloc[best_idx]["answer_mundart"]

    if best_sim < min_sim:
        return None, best_sim

    return best_answer, best_sim
```

```python
def debug_neighbors(
    models,
    raw_inp: str,
    topn: int = 5,
    filter_by_label: bool = True,
):
    resp_df = models["resp_df"]
    resp_emb = models["resp_emb"]
    sbert_model = models["sbert_model"]

    sbert_label = sbert_predict(models, [raw_inp])[0]

    q_emb = sbert_model.encode([raw_inp], convert_to_numpy=True)
    sims = cosine_similarity(q_emb, resp_emb)[0]

    candidate_idx = np.arange(len(resp_df))
    if filter_by_label and "label" in resp_df.columns:
        mask = (resp_df["label"].astype(str) == str(sbert_label))
        if mask.any():
            candidate_idx = np.where(mask)[0]

    if len(candidate_idx) == 0:
        return sbert_label, []

    sims_sub = sims[candidate_idx]
    order = np.argsort(-sims_sub)[:min(topn, len(sims_sub))]

    neighbors = []
    for local_idx in order:
        idx = candidate_idx[local_idx]
        row = resp_df.iloc[idx]
        sim = sims_sub[local_idx]

        neighbors.append({
            "similarity": float(sim),
            "user_text": str(row["user_text"]),
            "answer_mundart": str(row["answer_mundart"]),
            "label": row.get("label", "?"),
```

```
            "intent": row.get("intent", "?"),
            "is_seed": bool(row.get("is_seed", False)),
        })

    return sbert_label, neighbors


# ============================================================
# Token-/Zipf-Analyse
# ============================================================

def get_token_freqs(texts):
    freqs = Counter()
    for t in texts:
        clean = preprocess_text_chat(str(t))
        if not clean:
            continue
        for tok in clean.split():
            freqs[tok] += 1
    return freqs


def plot_zipf_with_fit(
    df: pd.DataFrame,
    text_col: str = "text_clean",
    min_freq: int = 1,
    fit_range: tuple[int, int] | None = (5, 100),
    title_suffix: str = "",
):
    texts = df[text_col].astype(str)
    freqs = get_token_freqs(texts)

    counts = np.array(sorted(
        [c for c in freqs.values() if c >= min_freq],
        reverse=True
    ))
    ranks = np.arange(1, len(counts) + 1)

    log_ranks = np.log(ranks)
    log_counts = np.log(counts)
```

```
hapax_mask = counts == 1
non_hapax_mask = ~hapax_mask

fig, ax = plt.subplots(figsize=(7, 5))

if non_hapax_mask.any():
    ax.loglog(
        ranks[non_hapax_mask],
        counts[non_hapax_mask],
        "o",
        markersize=4,
        alpha=0.7,
        label="Tokens (freq > 1)",
    )

if hapax_mask.any():
    ax.loglog(
        ranks[hapax_mask],
        counts[hapax_mask],
        "o",
        markersize=4,
        alpha=0.7,
        color="red",
        linestyle="none",
        label="Hapax (freq = 1)",
    )

if fit_range is not None:
    r_start, r_end = fit_range
    mask = (ranks >= r_start) & (ranks <= r_end)
else:
    mask = np.ones_like(ranks, dtype=bool)

X = log_ranks[mask].reshape(-1, 1)
y = log_counts[mask]

reg = LinearRegression().fit(X, y)
slope = reg.coef_[0]
```

```
    y_fit = reg.predict(log_ranks.reshape(-1, 1))
    ax.loglog(
        ranks,
        np.exp(y_fit),
        "--",
        label=f"Fit: slope={slope:.2f}",
    )

    zipf_slope = -1.0
    zipf_intercept = np.log(counts[0]) - zipf_slope * np.log(ranks[0])
    zipf_line = np.exp(zipf_intercept + zipf_slope * log_ranks)
    ax.loglog(
        ranks,
        zipf_line,
        ":",
        label="Referenz: slope = -1",
    )

    ax.set_xlabel("Rang")
    ax.set_ylabel("Häufigkeit")
    title = "Zipf-Plot mit Regressionsgeraden"
    if title_suffix:
        title += f" – {title_suffix}"
    ax.set_title(title)
    ax.legend()
    fig.tight_layout()

    alpha = -slope
    return fig, alpha, slope


# ============================================================
# Streamlit UI
# ============================================================

def main():
    st.set_page_config(
        page_title="Mundart-Chat Demo",
        page_icon="💬",
        layout="wide",
```

```
    )

    st.title("Mundart-Chat Demo")
    st.caption("Sentiment, Next-Word, Antwort-Retrieval für Schweizerdeutsch-Chat")

    # ---- Daten & Modelle EINMAL laden/trainieren ----
    base_df, resp_df = load_datasets()
    with st.spinner("Modelle werden geladen / trainiert ..."):
        models = train_all_models(base_df, resp_df)
    eval_info = models["eval_info"]

    # ---------------- Sidebar: Daten & Modelle ----------------
    with st.sidebar:
        st.header("📊 Modelle & Datengrundlage")
        st.write(f"💬 Anzahl Chatnachrichten: {len(resp_df)}")

        with st.expander("😊 Sentiment (3) & Intents (18)"):
            st.markdown(
                """
**😔 negativ**
- 😵 Stress & Überforderung (50)
- ⚡ Konflikte & Spannungen (50)
- 😟 Selbstzweifel & Unsicherheit (50)
- 🥀 Traurigkeit & Einsamkeit (50)
- 🩺 Gesundheit & Sorgen (50)
- 💭 Kurznachricht (negativ) (50)

**😐 neutral**
- 💬 Smalltalk / Allgemeines (50)
- 🗓️ Organisation & Abmachungen (50)
- ❓ Info-Fragen & Erklärungen (50)
- 🎮 Hobbys & Interessen (50)
- 💻 Tech-Support & Sachprobleme (50)
- 🔹 Kurznachricht (neutral) (50)

**😊 positiv**
- 🙏 Dankbarkeit (50)
```

```
- 😄 Freude & Gute Laune (50)
- 🏆 Erfolg & Stolz (50)
- 🤝 Verbundenheit & Nähe (50)
- 🚀 Motivation & Vorfreude (50)
- ✨ Kurznachricht (positiv) (50)
                """
            )

        with st.expander("🧠 Standardantworten (Defaults)"):
            st.markdown(
                """
- Jede Nachricht erhält automatisch eine passende Antwort
  basierend auf **Sentiment** (negativ/neutral/positiv)
  und **Intent** (18 Kategorien).
- Pro Intent gibt es mehrere Varianten inkl. **Kurznachrichten**.
- Falls keine Intent-spezifische Antwort existiert, greift ein
  allgemeiner Fallback je Sentiment.
                """
            )

        with st.expander("💻 Verwendete Modelle"):
            st.markdown(
                """
**1. Klassifikation (Sentiment & Intents)**
- Bag-of-Words + Logistic Regression
- TF-IDF (1–2-Gramme) + Logistic Regression
- SBERT-Embeddings + Logistic Regression

**2. Sprachmodell (Next-Word)**
- Einfaches N-Gramm-Modell (1–3-Gramme, Backoff)

**3. Antwort-Retrieval**
- SBERT-Embeddings + Kosinus-Ähnlichkeit
- Sucht die ähnlichsten Trainingsbeispiele und deren Antworten
                """
            )

        #1️⃣ Modell-Performance
        with st.expander("🔍 Modell-Performance (Testset)", expanded=False):
```

```
    for name, info in eval_info.items():
        st.subheader(name.upper())
        st.metric("Accuracy", f"{info['accuracy']:.3f}")

        report_dict = info["report"]
        df_report = pd.DataFrame(report_dict).T
        if "accuracy" in df_report.index:
            df_report = df_report.drop(index="accuracy")
        df_report = df_report[["precision", "recall", "f1-score", "support"]]
        st.dataframe(df_report, use_container_width=True)

        if "confusion_matrix" in info:
            st.caption(f"Confusion Matrix ({name.upper()})")
            cm_df = info["confusion_matrix"]
            cm = cm_df.to_numpy()
            fig, ax = plt.subplots()
            ax.imshow(cm, cmap="Blues")
            ax.set_xticks(np.arange(len(LABEL_ORDER)))
            ax.set_yticks(np.arange(len(LABEL_ORDER)))
            ax.set_xticklabels(LABEL_ORDER)
            ax.set_yticklabels(LABEL_ORDER)
            ax.set_xlabel("Predicted label")
            ax.set_ylabel("True label")
            for i in range(cm.shape[0]):
                for j in range(cm.shape[1]):
                    ax.text(j, i, int(cm[i, j]), ha="center", va="center")
            fig.tight_layout()
            st.pyplot(fig)

#2️⃣ Label-Verteilung
with st.expander("📈 Label-Verteilung", expanded=False):
    label_counts = base_df["label"].value_counts().reindex(LABEL_ORDER, fill_value=0)
    st.bar_chart(label_counts)
    st.write(label_counts)

#3️⃣ Textlängen
with st.expander("📏 Textlängen (Tokens)", expanded=False):
    lengths = base_df["text_clean"].astype(str).apply(
        lambda t: len(t.split()) if t.strip() else 0
```

```
    )
    st.write(f"Ø Länge: {lengths.mean():.1f} Tokens")
    st.write(f"Median: {lengths.median():.0f} Tokens")
    st.write(f"Max: {lengths.max():.0f} Tokens")
    st.bar_chart(lengths.value_counts().sort_index())

#④ Token-Statistik
with st.expander("🔤 Token-Statistik", expanded=False):
    unigram_counter = models["ngram_counts"][1]
    total_types = len(unigram_counter)
    total_tokens = sum(unigram_counter.values())
    hapax = [
        tok for (tok,), cnt in unigram_counter.items()
        if cnt == 1 and _is_good_token(tok)
    ]
    st.metric("Token-Typen (Vokab)", total_types)
    st.metric("Token-Instanzen (laufende Wörter)", total_tokens)
    st.metric("Hapax-Typen", len(hapax))
    st.metric("Hapax-Anteil", f"{len(hapax) / total_types:.2f}")
    st.write("Beispiele (Hapax):")
    st.write(", ".join(hapax[:10]))

#⑤ Zipf-Analyse
with st.expander("📐 Zipf-Analyse (Token-Verteilung)", expanded=False):
    try:
        fig_zipf, alpha, slope = plot_zipf_with_fit(
            base_df,
            text_col="text_clean",
            min_freq=1,
            fit_range=(5, 100),
            title_suffix="Basisdaten",
        )
        st.pyplot(fig_zipf)
        st.caption(f"Zipf-Exponent α ≈ {alpha:.3f}, Steigung ≈ {slope:.3f}")
    except Exception as e:
        st.warning(f"Zipf-Plot konnte nicht berechnet werden: {e}")

#⑥ N-Gramm-Statistik
with st.expander("🧩 N-Gramm-Statistik (LM)", expanded=False):
```

```
        ngram_counts = models["ngram_counts"]
        st.subheader("Unigramme (1-Gramme)")
        df_uni = get_top_ngrams(ngram_counts, n=1, topk=20)
        st.dataframe(df_uni, use_container_width=True)
        st.subheader("Bigramme (2-Gramme)")
        df_bi = get_top_ngrams(ngram_counts, n=2, topk=20)
        st.dataframe(df_bi, use_container_width=True)

    #7️⃣ Projekt-PDF
    with st.expander("📄 Projektpräsentation (PDF)", expanded=False):
        pdf_path = "Schlusspräsentation.pdf"
        try:
            with open(pdf_path, "rb") as f:
                pdf_bytes = f.read()
            st.download_button(
                label="📥 Präsentation als PDF herunterladen",
                data=pdf_bytes,
                file_name="Schlusspräsentation.pdf",
                mime="application/pdf",
            )
        except FileNotFoundError:
            st.warning(f"PDF nicht gefunden unter: {pdf_path}")

# ---------------- Eingabe ----------------
user_text = st.text_area(
    "Mundart-Nachricht eingeben",
    height=120,
    placeholder="z.B. «ich ha kei bock meh uf dä stress»",
)

# ---------------- Tabs ----------------
tab1, tab2, tab3, tab4 = st.tabs([
    "Sentiment-Klassifikation",
    "Next-Word Vorschlag",
    "Antwortvorschlag",
    "Debug Nachbarn",
])

# --- Tab 1: Klassifikation ---
```

```
with tab1:
    if st.button("Klassifizieren", key="btn_classify"):
        if not user_text.strip():
            st.warning("Bitte oben zuerst eine Nachricht eingeben.")
        else:
            with st.spinner("Klassifiziere ..."):
                cls = classify_text(models, user_text)

            col1, col2, col3 = st.columns(3)
            col1.metric("BoW", cls["bow_pred"])
            col2.metric("TF-IDF", cls["tfidf_pred"])
            col3.metric("SBERT", cls["sbert_pred"])

            st.subheader("Wahrscheinlichkeiten")
            probs_df = pd.DataFrame([
                {**{"modell": "BoW"}, **cls["bow_probs"]},
                {**{"modell": "TF-IDF"}, **cls["tfidf_probs"]},
                {**{"modell": "SBERT"}, **cls["sbert_probs"]},
            ])
            st.dataframe(probs_df, use_container_width=True)

# --- Tab 2: Next-Word ---
with tab2:
    if st.button("Next-Word Vorschläge berechnen", key="btn_nextword"):
        if not user_text.strip():
            st.warning("Bitte oben zuerst eine Nachricht eingeben.")
        else:
            with st.spinner("Berechne Next-Word-Vorschläge ..."):
                cands, backoff = next_word_candidates(
                    user_text,
                    models["ngram_counts"],
                    models["lm_analyzer"],
                    n_max=3,
                    topk=5,
                )
            if not cands:
                st.warning("Keine brauchbaren Vorschläge gefunden.")
            else:
                if backoff == 3:
                    st.info("N-Gramm-Level: 3-Gramm (voller Kontext benutzt)")
```

```
                elif backoff == 2:
                    st.info("N-Gramm-Level: 2-Gramm (Backoff – nur letztes Wort)")
                elif backoff == 1:
                    st.info("N-Gramm-Level: 1-Gramm (Unigram-Fallback)")
                else:
                    st.info("N-Gramm-Level: unbekannt / kein Treffer")

                rows = []
                for w, p in cands:
                    rows.append({
                        "Token": w,
                        "p (relativ)": round(p, 3),
                        "Vorschlag": (user_text + " " + w).strip(),
                    })
                st.table(pd.DataFrame(rows))

# --- Tab 3: Antwortvorschlag ---
with tab3:
    if st.button("Antwort generieren", key="btn_answer"):
        if not user_text.strip():
            st.warning("Bitte oben zuerst eine Nachricht eingeben.")
        else:
            with st.spinner("Klassifiziere & suche passende Antwort ..."):
                cls = classify_text(models, user_text)
                sbert_label = cls["sbert_pred"]
                answer, sim = generate_answer(
                    models,
                    user_text,
                    predicted_label=sbert_label,
                    topk=5,
                    min_sim=0.2,
                )
            st.write(f"**SBERT-Label:** {sbert_label}")
            if answer is None:
                st.warning(
                    f"Keine passende Antwort im Datensatz gefunden "
                    f"(beste Ähnlichkeit: {sim:.2f})."
                )
            else:
                st.subheader("Antwortvorschlag (Mundart)")
```

```
                    st.success(answer)
                    st.caption(f"Ähnlichkeit zu Trainingsbeispielen: {sim:.2f}")

    # --- Tab 4: Debug Nachbarn ---
    with tab4:
        topn = st.slider("Anzahl Nachbarn", min_value=3, max_value=15, value=5)
        if st.button("Ähnlichste Beispiele anzeigen", key="btn_debug"):
            if not user_text.strip():
                st.warning("Bitte oben zuerst eine Nachricht eingeben.")
            else:
                with st.spinner("Suche ähnliche Beispiele ..."):
                    sbert_label, neighbors = debug_neighbors(
                        models,
                        user_text,
                        topn=topn,
                        filter_by_label=True,
                    )
                st.write(f"**SBERT-Label:** {sbert_label}")
                if not neighbors:
                    st.warning("Keine passenden Nachbarn gefunden.")
                else:
                    df_neighbors = pd.DataFrame(neighbors)
                    df_neighbors = df_neighbors.drop(columns=["is_seed"], errors="ignore")
                    st.dataframe(df_neighbors, use_container_width=True)


if __name__ == "__main__":
    main()
```