```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, accuracy_score
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans


#%% ----------- Config ----------------
#DATA_CSV = "mundart_chat.csv"  # CSV mit Spalten: text,label
DATA_CSV ="mundart_augmented.csv"
SBERT_MODEL_NAME = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
RANDOM_STATE = 42
HIGHLIGHT_TEXT = "mega blöd"   # initial im Plot markiert
BATCH_SIZE = 32
np.random.seed(RANDOM_STATE)
df=pd.read_csv(DATA_CSV)

EMOJI_RE_RANGE = r"[\U0001F300-\U0001FAFF\u2600-\u27BF]"
def preprocess_text(t: str) -> str:
    if t is None: return ""
    t = str(t).lower()
    t = re.sub(r"\d+", "<NUM>", t)
    t = re.sub(r"(.)\1{2,}", r"\1\1", t)
    t = re.sub(r"[’']", " ", t)
    # Emojis einzeln separieren (keine Buckets!)
    t = re.sub(EMOJI_RE_RANGE + "+", lambda m: " " + " ".join(list(m.group(0))) + " ", t)
    t = t.replace("-", " ").replace("/", " ")
    t = (t.replace("ä","ae").replace("ö","oe").replace("ü","ue").replace("ß","ss"))
    # Dialekt-Mapping
    for k,v in {"nöd":"nicht","nid":"nicht","ned":"nicht","isch":"ist","bisch":"bist","chunsch":"kommst","huere":"sehr"}.items():
        t = re.sub(rf"\b{k}\b", v, t)
    t = re.sub(rf"[^\wäöüÄÖÜß<>{EMOJI_RE_RANGE}]+", " ", t)
    t = re.sub(r"\s{2,}", " ", t).strip()
    return t
```

```python
# Vectorizer mit Emoji-fähigem token_pattern
TOKEN_PATTERN = rf"(?u)(?:\b[\wäöüÄÖÜß]+\b|{EMOJI_RE_RANGE})"

#%% ---------- Deskriptives: Top-N-Grams, PMI, Emojis ----------
import re
from math import log2

def describe(df: pd.DataFrame) -> None:
    print("\n– Klassenverteilung –")
    print(df["label"].value_counts())
    print("\n– Länge (Zeichen) –")
    print(df["text"].str.len().describe()[["mean", "50%", "min", "max"]])

def _make_vectorizer(ngram_range=(1,1), min_df=2):
    return CountVectorizer(
        lowercase=True,
        ngram_range=ngram_range,
        min_df=min_df,
        token_pattern=TOKEN_PATTERN)


# gewünschte Label-Reihenfolge
LABEL_ORDER = ["negativ", "neutral", "positiv"]

def top_ngrams(df, label=None, ngram_range=(1,1), topk=5, min_df=2):
    """Top-N n-grams (nur für Teilmenge wenn label gesetzt)."""
    texts = df["text"] if label is None else df.loc[df["label"] == label, "text"]
    vec = _make_vectorizer(ngram_range=ngram_range, min_df=min_df)
    X = vec.fit_transform(texts.astype(str))
    vocab = np.array(vec.get_feature_names_out())
    counts = np.asarray(X.sum(axis=0)).ravel()
    rows = [(tok, int(cnt)) for tok, cnt in zip(vocab, counts)]
    rows.sort(key=lambda x: x[1], reverse=True)
    return pd.DataFrame(rows[:topk], columns=["ngram", "count"])

def top_ngrams_by_label(df, ngram_range=(1,1), topk=5, min_df=2):
    """Top-N je Klasse, sortiert nach LABEL_ORDER (wo vorhanden)."""
    out = {}
    labels_sorted = [l for l in LABEL_ORDER if l in set(df["label"])] + \
```

```python
                    [l for l in sorted(df["label"].unique()) if l not in LABEL_ORDER]
    for lab in labels_sorted:
        out[lab] = top_ngrams(df, label=lab, ngram_range=ngram_range, topk=topk, min_df=min_df)
    return out

def show_dict_of_dfs(d, title_prefix):
    """Schöne Konsolen-Ausgabe der DataFrames je Label."""
    for k in d:
        print(f"\n– {title_prefix}: {k} –")
        print(d[k].to_string(index=False))

# --- PMI je Label  ---
def pmi_bigrams_subset(texts, topk=5, min_df=3):
    """PMI nur auf einer Text-Teilmenge."""
    v1 = _make_vectorizer((1,1), min_df=1)
    X1 = v1.fit_transform(texts)
    vocab1 = np.array(v1.get_feature_names_out())
    uni_counts = np.asarray(X1.sum(axis=0)).ravel()
    uni = dict(zip(vocab1, uni_counts))

    v2 = _make_vectorizer((2,2), min_df=min_df)
    X2 = v2.fit_transform(texts)
    vocab2 = np.array(v2.get_feature_names_out())
    bi_counts = np.asarray(X2.sum(axis=0)).ravel()

    N = uni_counts.sum()
    rows = []
    for bg, c_xy in zip(vocab2, bi_counts):
        w1, w2 = bg.split()
        c_x = uni.get(w1, 0); c_y = uni.get(w2, 0)
        pmi = log2(((c_xy + 1) * N) / ((c_x + 1) * (c_y + 1)))
        rows.append((bg, int(c_xy), pmi))
    rows.sort(key=lambda x: (x[2], x[1]), reverse=True)
    return pd.DataFrame(rows[:topk], columns=["bigram", "count", "PMI"])

def pmi_bigrams_by_label(df, topk=5, min_df=3):
    out = {}
    labels_sorted = [l for l in LABEL_ORDER if l in set(df["label"])] + \
                    [l for l in sorted(df["label"].unique()) if l not in LABEL_ORDER]
    for lab in labels_sorted:
```

```python
        texts = df.loc[df["label"] == lab, "text"].astype(str)
        out[lab] = pmi_bigrams_subset(texts, topk=topk, min_df=min_df)
    return out

_EMOJI_RE = re.compile(r"[\U0001F300-\U0001FAFF\u2600-\u27BF]+")


describe(df)

print("\n== UNIGRAMS je Klasse ==")
uni_by = top_ngrams_by_label(df, ngram_range=(1,1), topk=5, min_df=2)
show_dict_of_dfs(uni_by, "Top-1g")

print("\n== BIGRAMS je Klasse ==")
bi_by = top_ngrams_by_label(df, ngram_range=(2,2), topk=5, min_df=2)
show_dict_of_dfs(bi_by, "Top-2g")

print("\n== PMI-BIGRAMS je Klasse ==")
pmi_by = pmi_bigrams_by_label(df, topk=5, min_df=3)
show_dict_of_dfs(pmi_by, "PMI-2g")



#%% ---------- Hilfsfunktionen ----------
def probs_pipeline(model, texts):
    """Gibt eine Liste von {label: prob}-Dicts für Pipeline-Modelle (BoW/TF-IDF) zurück."""
    vec = model.named_steps["vec"]
    clf = model.named_steps["clf"]
    X = vec.transform(texts)
    P = clf.predict_proba(X)  # Form (n, n_classes)
    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_proba(sbert_model, sbert_clf, texts, batch_size=BATCH_SIZE):
    emb = sbert_model.encode(pd.Series(texts).astype(str).tolist(),
                             convert_to_numpy=True, batch_size=batch_size)
    P = sbert_clf.predict_proba(emb)
```

```python
    cls = sbert_clf.classes_
    out = []
    for p in P:
        out.append({c: float(p[i]) for i, c in enumerate(cls)})
    return out

def format_probs(prob_dict, order=LABEL_ORDER, ndigits=2):
    """Formatiert als 'negativ: 0.12 | neutral: 0.34 | positiv: 0.54'."""
    return " | ".join(f"{lbl}: {prob_dict.get(lbl, 0.0):.{ndigits}f}" for lbl in order)

def eval_model(name, model, X_test, y_test) -> None:
    y_pred = model.predict(X_test)
    print(f"\n=== {name} ===")
    print(classification_report(y_test, y_pred, digits=3))
    print("Accuracy:", accuracy_score(y_test, y_pred))

def eval_sbert(sbert_model, sbert_clf, X_test, y_test, batch_size=BATCH_SIZE):
    Xv = pd.Series(X_test).astype(str).tolist()
    emb_test = sbert_model.encode(Xv, convert_to_numpy=True, batch_size=batch_size)
    y_pred = sbert_clf.predict(emb_test)
    print("\n=== SBERT-Embeddings + LogisticRegression ===")
    print(classification_report(y_test, y_pred, digits=3))
    print("Accuracy:", accuracy_score(y_test, y_pred))

def sbert_predict(sbert_model, sbert_clf, texts, batch_size=BATCH_SIZE):
    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)


#%% ---------- Plotting ----------
CLASS_COLORS = {"negativ": "tab:red", "neutral": "tab:gray", "positiv": "tab:green"}
def pca_kmeans_plot(
    name, X_2d, labels, texts, highlight_vec_2d, highlight_text,
    annotate_points: bool = False, max_points: int | None = None, random_state: int = RANDOM_STATE):
    labels = np.asarray(labels)
    texts = np.asarray(texts)

    # auf max_points herunterkürzen
    if (max_points is not None) and (max_points < len(labels)):
```

```python
    rng = np.random.default_rng(random_state)
    idx_keep = []
    # proportional je Klasse; min 1 pro vorhandener Klasse
    for lab in np.unique(labels):
        lab_idx = np.where(labels == lab)[0]
        # Anteil pro Klasse ~ (Klassenanteil * max_points), mind. 1
        k = max(1, int(round(max_points * len(lab_idx) / len(labels))))
        k = min(k, len(lab_idx))  # nicht mehr als vorhanden
        idx_keep.extend(rng.choice(lab_idx, size=k, replace=False))
    idx_keep = np.array(sorted(idx_keep))
    X_2d = X_2d[idx_keep]
    texts = texts[idx_keep]
    labels = labels[idx_keep]

# KMeans nur zur Visualisierung
kmeans = KMeans(n_clusters=3, random_state=random_state, n_init=10).fit(X_2d)

plt.figure()
for lab in sorted(np.unique(labels)):
    mask = (labels == lab)
    plt.scatter(X_2d[mask, 0], X_2d[mask, 1], s=28, alpha=0.9, label=lab,
                c=CLASS_COLORS.get(lab, "tab:blue"))
    if annotate_points:
        for x, y, t in zip(X_2d[mask, 0], X_2d[mask, 1], texts[mask]):
            short = (t[:22] + "…") if len(t) > 22 else t
            plt.annotate(short, (x, y), fontsize=8, alpha=0.8)

# Zentren
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], s=100, marker="D", c="orange")

# Highlight: Marker + Text
if highlight_vec_2d is not None:
    xh, yh = float(highlight_vec_2d[0, 0]), float(highlight_vec_2d[0, 1])
    plt.scatter(xh, yh, s=160, marker="X", c="gold")
    if highlight_text:
        label = "«" + (highlight_text[:30] + "…" if len(highlight_text) > 30 else highlight_text) + "»"
        plt.annotate(label, (xh, yh), fontsize=10, alpha=0.9, color="gold")

plt.title(f"PCA+KMeans – {name}")
```

```python
    plt.xlabel("PC1"); plt.ylabel("PC2")
    plt.legend(title=None, loc="best", fontsize=9)
    plt.tight_layout()
    plt.show()


def plot_space_for_bow(model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    vec = model.named_steps["vec"]
    X_all = vec.transform(df_text)
    X_dense = X_all.toarray()
    pca = PCA(n_components=2, random_state=RANDOM_STATE).fit(X_dense)
    X_2d = pca.transform(X_dense)
    h_2d = pca.transform(vec.transform([highlight_text]).toarray())
    pca_kmeans_plot("BoW", X_2d, df_labels.values, df_text.values, h_2d, highlight_text,
                    annotate_points=annotate_points, max_points=max_points)

def plot_space_for_tfidf(model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    vec = model.named_steps["vec"]
    X_all = vec.transform(df_text)
    X_dense = X_all.toarray()
    pca = PCA(n_components=2, random_state=RANDOM_STATE).fit(X_dense)
    X_2d = pca.transform(X_dense)
    h_2d = pca.transform(vec.transform([highlight_text]).toarray())
    pca_kmeans_plot("TF-IDF", X_2d, df_labels.values, df_text.values, h_2d, highlight_text,
                    annotate_points=annotate_points, max_points=max_points)

def plot_space_for_sbert(sbert_model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    all_emb = sbert_model.encode(pd.Series(df_text).astype(str).tolist(), convert_to_numpy=True, batch_size=BATCH_SIZE)
    pca = PCA(n_components=2, random_state=RANDOM_STATE).fit(all_emb)
    X_2d = pca.transform(all_emb)
    h_2d = pca.transform(sbert_model.encode([highlight_text], convert_to_numpy=True, batch_size=BATCH_SIZE))
    pca_kmeans_plot("SBERT", X_2d, df_labels.values, df_text.values, h_2d, highlight_text,
                    annotate_points=annotate_points, max_points=max_points)

def pca_kmeans_plot_3d(
    name, X_3d, labels, texts, highlight_vec_3d, highlight_text,
    annotate_points: bool = False, max_points: int | None = None, random_state: int = RANDOM_STATE):
    labels = np.asarray(labels)
    texts = np.asarray(texts)
```

```python
# stratifiziert auf max_points kürzen
if (max_points is not None) and (max_points < len(labels)):
    rng = np.random.default_rng(random_state)
    idx_keep = []
    for lab in np.unique(labels):
        lab_idx = np.where(labels == lab)[0]
        k = max(1, int(round(max_points * len(lab_idx) / len(labels))))
        k = min(k, len(lab_idx))
        idx_keep.extend(rng.choice(lab_idx, size=k, replace=False))
    idx_keep = np.array(sorted(idx_keep))
    X_3d = X_3d[idx_keep]
    texts = texts[idx_keep]
    labels = labels[idx_keep]

# KMeans nur zur Visualisierung (auf 3D)
n_clusters = min(3, len(np.unique(labels)), len(X_3d))
kmeans = KMeans(n_clusters=n_clusters, random_state=random_state, n_init=10).fit(X_3d)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d")

for lab in sorted(np.unique(labels)):
    mask = (labels == lab)
    ax.scatter(
        X_3d[mask, 0], X_3d[mask, 1], X_3d[mask, 2],
        s=28, alpha=0.9, label=lab,
        c=CLASS_COLORS.get(lab, "tab:blue"),
        edgecolors="white", linewidths=0.4
    )
    if annotate_points:
        for x, y, z, t in zip(X_3d[mask, 0], X_3d[mask, 1], X_3d[mask, 2], texts[mask]):
            short = (t[:22] + "…") if len(t) > 22 else t
            ax.text(x, y, z, short, fontsize=8, alpha=0.8)

# Zentren
centers = kmeans.cluster_centers_
ax.scatter(centers[:, 0], centers[:, 1], centers[:, 2], s=120, marker="D", c="orange", edgecolors="white", linewidths=0.6)

# Highlight
if highlight_vec_3d is not None:
```

```
        xh, yh, zh = map(float, highlight_vec_3d.ravel())
        ax.scatter(xh, yh, zh, s=180, marker="X", c="gold", edgecolors="black", linewidths=0.6)
        if highlight_text:
            lab = "«" + (highlight_text[:30] + "…" if len(highlight_text) > 30 else highlight_text) + "»"
            ax.text(xh, yh, zh, lab, fontsize=10, color="black", weight="semibold")

    ax.set_title(f"PCA+KMeans (3D) — {name}", pad=10)
    ax.set_xlabel("PC1"); ax.set_ylabel("PC2"); ax.set_zlabel("PC3")
    leg = ax.legend(title=None, loc="best", fontsize=9, frameon=True)
    if leg and leg.get_frame(): leg.get_frame().set_alpha(0.9)
    plt.tight_layout()
    plt.show()

def plot_space_for_bow_3d(model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    vec = model.named_steps["vec"]
    X_all = vec.transform(df_text)
    X_dense = X_all.toarray()
    pca = PCA(n_components=3, random_state=RANDOM_STATE).fit(X_dense)
    X_3d = pca.transform(X_dense)
    h_3d = pca.transform(vec.transform([highlight_text]).toarray())
    pca_kmeans_plot_3d("BoW", X_3d, df_labels.values, df_text.values, h_3d, highlight_text,
                       annotate_points=annotate_points, max_points=max_points)

def plot_space_for_tfidf_3d(model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    vec = model.named_steps["vec"]
    X_all = vec.transform(df_text)
    X_dense = X_all.toarray()
    pca = PCA(n_components=3, random_state=RANDOM_STATE).fit(X_dense)
    X_3d = pca.transform(X_dense)
    h_3d = pca.transform(vec.transform([highlight_text]).toarray())
    pca_kmeans_plot_3d("TF-IDF", X_3d, df_labels.values, df_text.values, h_3d, highlight_text,
                       annotate_points=annotate_points, max_points=max_points)

def plot_space_for_sbert_3d(sbert_model, df_text, df_labels, highlight_text, annotate_points: bool = False, max_points: int | None = None):
    all_emb = sbert_model.encode(pd.Series(df_text).astype(str).tolist(), convert_to_numpy=True, batch_size=BATCH_SIZE)
    pca = PCA(n_components=3, random_state=RANDOM_STATE).fit(all_emb)
    X_3d = pca.transform(all_emb)
    h_3d = pca.transform(sbert_model.encode([highlight_text], convert_to_numpy=True, batch_size=BATCH_SIZE))
    pca_kmeans_plot_3d("SBERT", X_3d, df_labels.values, df_text.values, h_3d, highlight_text,
                       annotate_points=annotate_points, max_points=max_points)
```

```python
#%% ---------- Modelle trainieren und evaluieren ----------
def train_bow(X_train, y_train):
    pipe = Pipeline([
        ("vec", CountVectorizer(token_pattern=TOKEN_PATTERN)),
        ("clf", LogisticRegression(max_iter=1000, random_state=RANDOM_STATE))
    ])
    return pipe.fit(X_train, y_train)

def train_tfidf(X_train, y_train):
    pipe = Pipeline([
        ("vec", TfidfVectorizer(ngram_range=(1,2), token_pattern=TOKEN_PATTERN)),
        ("clf", LogisticRegression(max_iter=1000, random_state=RANDOM_STATE))
    ])
    return pipe.fit(X_train, y_train)

def train_sbert(X_train, y_train, model_name=SBERT_MODEL_NAME, batch_size=BATCH_SIZE):
    from sentence_transformers import SentenceTransformer
    sbert_model = SentenceTransformer(model_name)
    Xt = pd.Series(X_train).astype(str).tolist()
    emb_train = sbert_model.encode(Xt, convert_to_numpy=True, batch_size=batch_size)
    sbert_clf = LogisticRegression(max_iter=1000, random_state=RANDOM_STATE).fit(emb_train, y_train)
    return sbert_model, sbert_clf

# Clean & Raw nebeneinander halten
df = df.drop_duplicates(subset=["text"]).reset_index(drop=True)
df["text_clean"] = df["text"].astype(str).apply(preprocess_text)

# identische Indizes splitten
X_tr_clean, X_te_clean, y_train, y_test = train_test_split(
    df["text_clean"], df["label"], test_size=0.25,
    random_state=RANDOM_STATE, stratify=df["label"])

# die korrespondierenden Rohtexte ziehen:
X_tr_raw   = df.loc[X_tr_clean.index, "text"]
X_te_raw   = df.loc[X_te_clean.index, "text"]

# Train
bow   = train_bow(X_tr_clean, y_train)
```

```python
tfidf = train_tfidf(X_tr_clean, y_train)
sbert_model, sbert_clf = train_sbert(X_tr_raw, y_train)

# Eval
eval_model("BoW + LogisticRegression", bow,   X_te_clean, y_test)
eval_model("TF-IDF + LogisticRegression",     tfidf,     X_te_clean, y_test)
eval_sbert(sbert_model, sbert_clf, X_te_raw, y_test)


#%% ---------- Visualisierung ----------
# 2D
plot_space_for_bow(bow,   df["text_clean"], df["label"], preprocess_text(HIGHLIGHT_TEXT), annotate_points=False, max_points=100)
plot_space_for_tfidf(tfidf, df["text_clean"], df["label"], preprocess_text(HIGHLIGHT_TEXT), annotate_points=True, max_points=20)
plot_space_for_sbert(sbert_model, df["text"], df["label"], HIGHLIGHT_TEXT, annotate_points=True, max_points=9)

# 3D
plot_space_for_bow_3d(bow,   df["text_clean"], df["label"], preprocess_text(HIGHLIGHT_TEXT), annotate_points=False, max_points=200)
plot_space_for_tfidf_3d(tfidf, df["text_clean"], df["label"], preprocess_text(HIGHLIGHT_TEXT), annotate_points=False, max_points=200)
plot_space_for_sbert_3d(sbert_model, df["text"], df["label"], HIGHLIGHT_TEXT, annotate_points=False, max_points=200)


#%% ---------- Interaktive Schleife ----------
print("\nInteraktive Eingabe (leer lassen zum Beenden):")
while True:
    try:
        user_text = input("> ").strip()
    except (EOFError, KeyboardInterrupt):
        break
    if not user_text:
        break

    # Konsistente Inputs
    raw_inp   = user_text
    clean_inp = preprocess_text(user_text)

    # --- Top-Labels ---
    bow_pred   = bow.predict([clean_inp])[0]
    tfidf_pred = tfidf.predict([clean_inp])[0]
    sbert_pred = sbert_predict(sbert_model, sbert_clf, [raw_inp])[0]
```

```python
# --- Wahrscheinlichkeiten/Gewichtungen ---
bow_probs   = probs_pipeline(bow,   [clean_inp])[0]
tfidf_probs = probs_pipeline(tfidf, [clean_inp])[0]
sbert_probs = sbert_predict_proba(sbert_model, sbert_clf, [raw_inp])[0]

print("\n– Ergebnisse –")
print("BoW   ->", bow_pred,   " | ", format_probs(bow_probs))
print("TF-IDF->", tfidf_pred, " | ", format_probs(tfidf_probs))
print("SBERT ->", sbert_pred, " | ", format_probs(sbert_probs))
print()
print()
```