```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import os

# Base project directory on Google Drive
BASE_DIR = "/content/drive/MyDrive/SkinAI_Project"

# Data dirs
DATA_DIR       = f"{BASE_DIR}/data"
RAW_DIR        = f"{DATA_DIR}/raw"
PROCESSED_DIR  = f"{DATA_DIR}/processed"
SYNTHETIC_DIR  = f"{DATA_DIR}/synthetic"

MODELS_DIR     = f"{BASE_DIR}/models"
NOTEBOOKS_DIR  = f"{BASE_DIR}/notebooks"

print("BASE_DIR      :", BASE_DIR)
print("DATA_DIR      :", DATA_DIR)
print("SYNTHETIC_DIR :", SYNTHETIC_DIR)
print("MODELS_DIR    :", MODELS_DIR)

print("\nSynthetic files:", os.listdir(SYNTHETIC_DIR))
print("Models files   :", os.listdir(MODELS_DIR))
```

```
BASE_DIR      : /content/drive/MyDrive/SkinAI_Project
DATA_DIR      : /content/drive/MyDrive/SkinAI_Project/data
SYNTHETIC_DIR : /content/drive/MyDrive/SkinAI_Project/data/synthetic
MODELS_DIR    : /content/drive/MyDrive/SkinAI_Project/models

Synthetic files: ['patient_profiles_10000.csv', 'treatment_records_10000.csv', 'disease_master.csv', 'symptom_list.csv', 'diseas
Models files   : ['skin_text_classifier_tfidf_logreg.joblib', 'sbert_logreg_classifier.joblib', 'sbert_embedder_name.json', 'dis
```

```
import joblib
import numpy as np

MODEL_PATH = os.path.join(MODELS_DIR, "skin_text_classifier_tfidf_logreg_final.joblib")
VECT_PATH  = os.path.join(MODELS_DIR, "vectorizer_tfidf.joblib")

clf = joblib.load(MODEL_PATH)
vectorizer = joblib.load(VECT_PATH)

print("Loaded classifier from:", MODEL_PATH)
print("Loaded vectorizer from :", VECT_PATH)

def predict_disease_from_text(text: str):
    """
    Predict a single disease label + confidence from free-text symptoms.
    """
    X = vectorizer.transform([text])
    pred = clf.predict(X)[0]
    proba = clf.predict_proba(X)[0]
    classes = list(clf.classes_)  # ensure list for indexing
    conf = float(proba[classes.index(pred)])
    return pred, conf

def predict_top_k_from_text(text: str, k: int = 5):
    """
    Return top-k (label, confidence) predictions for explainability.
    """
    X = vectorizer.transform([text])
    proba = clf.predict_proba(X)[0]
    classes = list(clf.classes_)
    idx_sorted = np.argsort(proba)[::-1]
    top = []
    for i in idx_sorted[:k]:
        top.append((classes[i], float(proba[i])))
    return top

# quick sanity test
example_text = "I have red, itchy, scaly patches on my elbows and knees."
```

```
pred_label, conf = predict_disease_from_text(example_text)
print("Example text:", example_text)
print("Predicted label:", pred_label)
print("Confidence     :", round(conf, 3))
print("\nTop-5 predictions:")
for lab, c in predict_top_k_from_text(example_text, k=5):
    print(" -", lab, "|", round(c, 3))
```

```
Loaded classifier from: /content/drive/MyDrive/SkinAI_Project/models/skin_text_classifier_tfidf_logreg_final.joblib
Loaded vectorizer from : /content/drive/MyDrive/SkinAI_Project/models/vectorizer_tfidf.joblib
Example text: I have red, itchy, scaly patches on my elbows and knees.
Predicted label: Psoriasis
Confidence     : 0.093

Top-5 predictions:
 - Psoriasis | 0.093
 - Eczema | 0.09
 - Atopic Dermatitis | 0.06
 - Tinea Ringworm Candidiasis | 0.044
 - Vascular lesion | 0.043
```

```
import pandas as pd

clinical_path   = os.path.join(SYNTHETIC_DIR, "clinical_cases_10000.csv")
print("clinical_path:", clinical_path, "| exists:", os.path.exists(clinical_path))

df_clinical = pd.read_csv(clinical_path)
print("Clinical cases shape:", df_clinical.shape)
print("Columns:", df_clinical.columns.tolist())

# Normalize treatment outcome and keep only improved rows
df_clinical["treatment_outcome_norm"] = (
    df_clinical["treatment_outcome"]
    .astype(str)
    .str.lower()
    .str.strip()
)

df_good = df_clinical[df_clinical["treatment_outcome_norm"] == "improved"].copy()
print("Rows with 'improved' outcome:", len(df_good))

if df_good.empty:
    print("[WARN] No 'improved' records – falling back to all records.")
    df_good = df_clinical.copy()

required_cols = ["primary_disease", "severity", "recommended_medicine", "is_over_the_counter"]
for col in required_cols:
    if col not in df_good.columns:
        raise KeyError(f"Required column '{col}' is missing in clinical_cases_10000.csv")

grouped = (
    df_good
    .groupby(["primary_disease", "severity", "recommended_medicine", "is_over_the_counter"], as_index=False)
    .size()
    .rename(columns={"size": "count"})
)

AVAILABLE_TREAT_DISEASES = sorted(grouped["primary_disease"].unique())
print("\nNumber of diseases in treatment stats:", len(AVAILABLE_TREAT_DISEASES))
print("First 20:", AVAILABLE_TREAT_DISEASES[:20])
```

```
clinical_path: /content/drive/MyDrive/SkinAI_Project/data/synthetic/clinical_cases_10000.csv | exists: True
Clinical cases shape: (10000, 20)
Columns: ['case_id', 'patient_id', 'age', 'gender', 'region', 'skin_type', 'fitzpatrick_type', 'primary_disease', 'disease_categ
Rows with 'improved' outcome: 3351

Number of diseases in treatment stats: 29
First 20: ['Acne rosacea', 'Acne vulgaris', 'Atopic dermatitis (Eczema)', 'Cellulitis', 'Chronic urticaria', 'Contact dermatitis
```

```
def get_top_treatments_for(disease: str, severity: str = None, top_k: int = 5) -> pd.DataFrame:
    """
    Return top treatment rows for a given disease and optional severity from 'grouped'.
    """
    df = grouped[grouped["primary_disease"].str.lower() == disease.lower()]
    if df.empty:
        print(f"[get_top_treatments_for] No data for disease='{disease}'.")
        print("Some available diseases:", AVAILABLE_TREAT_DISEASES[:20])
        return pd.DataFrame()

    if severity is not None:
        df_sev = df[df["severity"].astype(str).str.lower() == severity.lower()]
        if not df_sev.empty:
            df = df_sev
        else:
            print(f"[get_top_treatments_for] No severity='{severity}' for disease='{disease}', using all severities.")

    return df.sort_values("count", ascending=False).head(top_k)


def recommend_treatment(
    disease: str,
    severity: str = None,
    age: int = None,
    allergies: list = None,
    prefer_otc: bool = True,
    top_k: int = 3
):
    """
    Recommend treatments based on synthetic clinical data.
    """
    if allergies is None:
        allergies = []

    candidates = get_top_treatments_for(disease, severity, top_k=100).copy()
    if candidates.empty:
        print(f"[recommend_treatment] No candidates for disease='{disease}'.")
        return []

    # Allergy filter
    def is_safe(row):
        med = str(row["recommended_medicine"]).lower()
        return not any(a.lower() in med for a in allergies)

    candidates["is_safe"] = candidates.apply(is_safe, axis=1)
    safe_candidates = candidates[candidates["is_safe"]]

    if safe_candidates.empty:
        print("[recommend_treatment] No safe meds after allergy filtering; using all candidates.")
        safe_candidates = candidates

    # Prefer OTC if requested
    if prefer_otc:
        otc = safe_candidates[safe_candidates["is_over_the_counter"] == True]
        if len(otc) > 0:
            safe_candidates = otc

    result = (
        safe_candidates
        .sort_values("count", ascending=False)
        .head(top_k)
        .drop(columns=["is_safe"])
        .to_dict(orient="records")
    )
    return result
```

```
import networkx as nx

disease_master_path = os.path.join(SYNTHETIC_DIR, "disease_master.csv")
symptom_list_path   = os.path.join(SYNTHETIC_DIR, "symptom_list.csv")
edges_path          = os.path.join(SYNTHETIC_DIR, "disease_symptom_edges.csv")

df_disease_master = pd.read_csv(disease_master_path)
df_symptom_list   = pd.read_csv(symptom_list_path)
df_edges          = pd.read_csv(edges_path)

print("disease_master:", df_disease_master.shape)
print("symptom_list  :", df_symptom_list.shape)
print("edges         :", df_edges.shape)
```

```
G = nx.Graph()

# Disease nodes
for _, row in df_disease_master.iterrows():
    dname = str(row["disease_name"]).strip()
    if not dname:
        continue
    G.add_node(
        f"disease:{dname}",
        node_type="disease",
        disease_name=dname,
        category=str(row.get("category", "")).strip(),
        description=str(row.get("description", "")).strip()
    )

# Symptom nodes
symptom_id_col   = "symptom_id" if "symptom_id" in df_symptom_list.columns else None
symptom_name_col = "symptom_name" if "symptom_name" in df_symptom_list.columns else None

for _, row in df_symptom_list.iterrows():
    if symptom_id_col:
        sid = row[symptom_id_col]
    else:
        sid = str(row[symptom_name_col]).strip()
    sname = str(row[symptom_name_col]).strip() if symptom_name_col else str(sid)
    if not sname:
        continue
    G.add_node(
        f"symptom:{sid}",
        node_type="symptom",
        symptom_id=sid,
        symptom_name=sname
    )

# Edges
edge_count = 0
for _, row in df_edges.iterrows():
    dname = str(row["disease_name"]).strip()
    sid   = row["symptom_id"] if "symptom_id" in row else None
    d_node = f"disease:{dname}"
    s_node = f"symptom:{sid}"
    if d_node in G.nodes and s_node in G.nodes:
        G.add_edge(d_node, s_node)
        edge_count += 1

print(f"Knowledge graph built with {G.number_of_nodes()} nodes and {G.number_of_edges()} edges.")
print("Edges added from CSV:", edge_count)

# Helper maps
DISEASE_NODES = {
    data["disease_name"]: node_id
    for node_id, data in G.nodes(data=True)
    if data.get("node_type") == "disease"
}

SYMPTOM_ID_TO_NAME = {}
if "symptom_id" in df_symptom_list.columns:
    for _, row in df_symptom_list.iterrows():
        SYMPTOM_ID_TO_NAME[row["symptom_id"]] = str(row.get("symptom_name", "")).strip()
```

```
disease_master: (29, 3)
symptom_list  : (83, 2)
edges         : (89, 3)
Knowledge graph built with 112 nodes and 89 edges.
Edges added from CSV: 89
```

```
def get_symptoms_for_disease(disease_name: str, top_n: int = 10):
    dname_norm = disease_name.strip()
    d_node = DISEASE_NODES.get(dname_norm)
    if d_node is None:
        print(f"[get_symptoms_for_disease] Disease '{disease_name}' not found in KG.")
        return []

    neighbors = list(G.neighbors(d_node))
    symptom_list = []
    for n in neighbors:
        data = G.nodes[n]
        if data.get("node_type") != "symptom":
            continue
        sid = data.get("symptom_id")
        sname = data.get("symptom_name") or SYMPTOM_ID_TO_NAME.get(sid, "")
        symptom_list.append((sname, sid))

    symptom_list = sorted(symptom_list, key=lambda x: x[0])
    if top_n is not None:
        symptom_list = symptom_list[:top_n]
    return symptom_list


def get_related_diseases(disease_name: str, top_n: int = 5):
    dname_norm = disease_name.strip()
    d_node = DISEASE_NODES.get(dname_norm)
    if d_node is None:
        print(f"[get_related_diseases] Disease '{disease_name}' not found in KG.")
        return []

    neighbors = list(G.neighbors(d_node))
    disease_symptoms = {
        G.nodes[n].get("symptom_id") for n in neighbors
        if G.nodes[n].get("node_type") == "symptom"
    }

    related = []
    for other_name, other_node in DISEASE_NODES.items():
        if other_name == dname_norm:
            continue
        neighbors_other = list(G.neighbors(other_node))
        other_symptoms = {
            G.nodes[n].get("symptom_id") for n in neighbors_other
            if G.nodes[n].get("node_type") == "symptom"
        }
        shared = disease_symptoms.intersection(other_symptoms)
        if len(shared) > 0:
            cat = G.nodes[other_node].get("category", "")
            related.append({
                "disease_name": other_name,
                "shared_symptom_count": len(shared),
                "category": cat,
                "reason": "shared_symptoms"
            })

    if related:
        related_sorted = sorted(related, key=lambda x: x["shared_symptom_count"], reverse=True)
        if top_n is not None:
            related_sorted = related_sorted[:top_n]
        return related_sorted

    # fallback: same category
    print("[get_related_diseases] No shared-symptom diseases found. Using same-category fallback.")
    row_ref = df_disease_master[df_disease_master["disease_name"].str.strip() == dname_norm]
    if row_ref.empty or "category" not in row_ref.columns:
        return []

    cat = str(row_ref.iloc[0]["category"]).strip()
    same_cat = df_disease_master[
        (df_disease_master["category"].astype(str).str.strip() == cat) &
        (df_disease_master["disease_name"].str.strip() != dname_norm)
    ]

    results = []
    for _, row in same_cat.iterrows():
        other_name = str(row["disease_name"]).strip()
        results.append({
            "disease_name": other_name,
            "shared_symptom_count": 0,
            "category": cat,
            "reason": "same_category"
        })

    if top_n is not None:
        results = results[:top_n]
    return results
```

```
def explain_disease_with_kg(disease_name: str, top_symptoms: int = 10, top_related: int = 5):
    dname_norm = disease_name.strip()
    row = df_disease_master[df_disease_master["disease_name"].str.strip() == dname_norm]

    if not row.empty:
        category = str(row.iloc[0].get("category", "")).strip()
        description = str(row.iloc[0].get("description", "")).strip()
    else:
        category = ""
        description = ""

    symptoms = get_symptoms_for_disease(dname_norm, top_n=top_symptoms)
    related_diseases = get_related_diseases(dname_norm, top_n=top_related)

    return {
        "disease_name": dname_norm,
        "category": category,
        "description": description,
        "symptoms": [
            {"symptom_name": sname, "symptom_id": sid}
            for (sname, sid) in symptoms
        ],
        "related_diseases": related_diseases
    }


def print_kg_explanation(kg_info: dict):
    if not kg_info:
        print("No KG info available.")
        return

    dname = kg_info.get("disease_name", "")
    category = kg_info.get("category", "")
    desc = kg_info.get("description", "")

    print(f"=== Knowledge Graph Explanation for: {dname} ===\n")

    if category:
        print(f"Category: {category}")

    if desc:
        print("\nDescription:")
        print(desc)

    # Symptoms
    print("\nTop associated symptoms:")
    symptoms = kg_info.get("symptoms", [])
    if not symptoms:
        print(" - (no symptoms found in KG)")
    else:
        for s in symptoms:
            print(f" - {s['symptom_name']} (id={s['symptom_id']})")

    # Related diseases
    print("\nRelated diseases:")
    rel = kg_info.get("related_diseases", [])
    if not rel:
        print(" - (no related diseases found in KG)")
    else:
        for r in rel:
            print(
                f" - {r['disease_name']} "
                f"| shared_symptoms={r['shared_symptom_count']} "
                f"| category={r.get('category', '')} "
                f"| reason={r.get('reason', '')}"
            )
```

```
import re
import difflib

# Collect canonical disease names from treatment data + KG
CANONICAL_DISEASES = sorted(
    set(AVAILABLE_TREAT_DISEASES) |
    set(df_disease_master["disease_name"].astype(str).str.strip())
)

print("Total canonical disease names:", len(CANONICAL_DISEASES))

def normalize_disease_name(name: str) -> str:
    """
    Normalize disease name into a clean, comparable token string.
    Example:
      'Shingles (Herpes Zoster)'        -> 'herpes shingles zoster'
      'Herpes zoster (Shingles)'        -> 'herpes shingles zoster'
      'Atopic Dermatitis (Eczema)'      -> 'atopic dermatitis eczema'
    """
    if not isinstance(name, str):
        name = str(name)
    name = name.lower().strip()
    # remove all non-alphanumeric characters
    name = re.sub(r'[^a-z0-9\s]', ' ', name)
    # split into tokens, drop empties, deduplicate, sort
    tokens = [t for t in name.split() if t]
    tokens = sorted(set(tokens))
    return " ".join(tokens)

# Build map: normalized tokens -> canonical disease name
NORMALIZED_TO_CANONICAL = {}
for d in CANONICAL_DISEASES:
    norm = normalize_disease_name(d)
    NORMALIZED_TO_CANONICAL.setdefault(norm, d)

def map_to_synthetic_disease(label: str) -> str:
    """
    Maps classifier label to the closest disease name used in synthetic data & KG.
    Uses normalization and fuzzy matching on keyword tokens.
    Steps:
      1. Normalize the predicted label and try direct normalized match.
      2. If no direct match, fuzzy match normalized strings.
      3. If still nothing, return original label.
    """
    if not label:
        return label

    label_stripped = label.strip()
    label_norm = normalize_disease_name(label_stripped)

    # 1) direct normalized match
    if label_norm in NORMALIZED_TO_CANONICAL:
        mapped = NORMALIZED_TO_CANONICAL[label_norm]
        if mapped != label_stripped:
            print(f"[map_to_synthetic_disease] Normalized-mapped '{label_stripped}' -> '{mapped}'")
        return mapped

    # 2) fuzzy match between normalized strings
    all_norm_keys = list(NORMALIZED_TO_CANONICAL.keys())
    best_norm = difflib.get_close_matches(label_norm, all_norm_keys, n=1, cutoff=0.5)
    if best_norm:
        mapped = NORMALIZED_TO_CANONICAL[best_norm[0]]
        print(f"[map_to_synthetic_disease] Fuzzy-mapped '{label_stripped}' -> '{mapped}'")
        return mapped

    # 3) fallback
    print(f"[map_to_synthetic_disease] No mapping found for '{label_stripped}', using as-is.")
    return label_stripped
```

```
Total canonical disease names: 29
```

```
def analyze_case(
    user_text: str,
    severity: str = "moderate",
    age: int = None,
    allergies: list = None,
    prefer_otc: bool = True,
    top_treatments: int = 3,
    top_symptoms: int = 10,
    top_related: int = 5,
    top_model_preds: int = 5
):
    """
    End-to-end pipeline:
      1. Text -> disease prediction (classifier)
      2. Map predicted label -> synthetic disease name
      3. Recommend treatments
      4. KG explanation (symptoms + related diseases)
    """
    if allergies is None:
        allergies = []

    # 1) Text classification
    pred_label, conf = predict_disease_from_text(user_text)
    top_preds = predict_top_k_from_text(user_text, k=top_model_preds)

    # 2) Map to synthetic/KG disease name
    mapped_disease = map_to_synthetic_disease(pred_label)

    # 3) Treatment recommendation
    treatments = recommend_treatment(
        disease=mapped_disease,
        severity=severity,
        age=age,
        allergies=allergies,
        prefer_otc=prefer_otc,
        top_k=top_treatments
    )

    # 4) KG explanation
    kg_info = explain_disease_with_kg(mapped_disease, top_symptoms=top_symptoms, top_related=top_related)

    return {
        "user_text": user_text,
        "severity": severity,
        "age": age,
        "allergies": allergies,
        "prefer_otc": prefer_otc,

        "predicted_label": pred_label,
        "prediction_confidence": conf,
        "top_predictions": [
            {"label": lab, "confidence": c}
            for (lab, c) in top_preds
        ],

        "mapped_disease": mapped_disease,

        "treatments": treatments,
        "kg_explanation": kg_info,
    }
```

```
def print_analysis(result: dict):
    print("\n=== SkinAI Analysis Result ===\n")

    print("User text:", result["user_text"])
    print("Severity :", result["severity"])
    print("Age      :", result["age"])
    print("Allergies:", result["allergies"])
    print("Prefer OTC:", result["prefer_otc"])

    print("\n--- Classifier Output ---")
    print("Predicted label:", result["predicted_label"])
    print("Confidence     :", round(result["prediction_confidence"], 3))

    print("\nTop model predictions:")
    for p in result["top_predictions"]:
        print(f" - {p['label']} | conf={round(p['confidence'], 3)}")

    print("\nMapped disease (for synthetic data & KG):", result["mapped_disease"])

    print("\n--- Suggested Treatments ---")
    treatments = result.get("treatments", [])
    if not treatments:
        print("No treatments found for this mapped disease in synthetic dataset.")
    else:
        for t in treatments:
            med = t["recommended_medicine"]
            sev = t["severity"]
            otc = "yes" if t["is_over_the_counter"] else "no"
            cnt = t["count"]
            print(f" - {med} | severity={sev} | OTC={otc} | support_count={cnt}")

    print("\n--- Knowledge Graph Explanation ---")
    kg_info = result.get("kg_explanation")
    if kg_info:
        print_kg_explanation(kg_info)
    else:
        print("No KG information available.")
```

```
test_text = "I have very itchy red circular patches on my arm that are slowly spreading."
result = analyze_case(
    user_text=test_text,
    severity="moderate",
    age=25,
    allergies=["penicillin"],  # example
    prefer_otc=True,
    top_treatments=3,
    top_symptoms=10,
    top_related=5,
    top_model_preds=5
)

print_analysis(result)
```

```
[map_to_synthetic_disease] Normalized-mapped 'Ringworm (Tinea Corporis)' -> 'Tinea corporis (Ringworm)'
[get_related_diseases] No shared-symptom diseases found. Using same-category fallback.

=== SkinAI Analysis Result ===

User text: I have very itchy red circular patches on my arm that are slowly spreading.
Severity : moderate
Age      : 25
Allergies: ['penicillin']
Prefer OTC: True

--- Classifier Output ---
Predicted label: Ringworm (Tinea Corporis)
Confidence     : 0.074

Top model predictions:
 - Ringworm (Tinea Corporis) | conf=0.074
 - Vascular lesion | conf=0.06
 - Psoriasis | conf=0.056
 - Benign keratosis | conf=0.05
 - Atopic Dermatitis | conf=0.045

Mapped disease (for synthetic data & KG): Tinea corporis (Ringworm)

--- Suggested Treatments ---
 - topical antifungal cream | severity=moderate | OTC=yes | support_count=37

--- Knowledge Graph Explanation ---
=== Knowledge Graph Explanation for: Tinea corporis (Ringworm) ===

Category: fungal

Description:
Dermatophyte infection causing annular, scaly plaques.

Top associated symptoms:
 - central clearing (id=10)
 - ring-shaped rash (id=59)
 - scaly edge (id=65)

Related diseases:
 - Tinea pedis (Athlete's foot) | shared_symptoms=0 | category=fungal | reason=same_category
 - Tinea cruris (Jock itch) | shared_symptoms=0 | category=fungal | reason=same_category
 - Tinea capitis (Scalp ringworm) | shared_symptoms=0 | category=fungal | reason=same_category
 - Pityriasis versicolor | shared_symptoms=0 | category=fungal | reason=same_category
```

```
test_text = "My face often becomes red and flushed, especially on my cheeks and nose. It burns sometimes and gets worse after s
result = analyze_case(
    user_text=test_text,
    severity="moderate",
    age=25,
    allergies=["penicillin"],  # example
    prefer_otc=True,
    top_treatments=3,
    top_symptoms=10,
    top_related=5,
    top_model_preds=5
)

print_analysis(result)
```

```
[get_related_diseases] No shared-symptom diseases found. Using same-category fallback.

=== SkinAI Analysis Result ===

User text: My face often becomes red and flushed, especially on my cheeks and nose. It burns sometimes and gets worse after sun
Severity : moderate
Age      : 25
Allergies: ['penicillin']
Prefer OTC: True

--- Classifier Output ---
Predicted label: Rosacea
Confidence     : 0.078

Top model predictions:
 - Rosacea | conf=0.078
 - Atopic Dermatitis | conf=0.071
 - Actinic keratosis | conf=0.057
 - Eczema | conf=0.045
 - Vascular lesion | conf=0.045

Mapped disease (for synthetic data & KG): Rosacea

--- Suggested Treatments ---
 - metronidazole gel | severity=moderate | OTC=yes | support_count=16
 - azelaic acid | severity=moderate | OTC=yes | support_count=12

--- Knowledge Graph Explanation ---
=== Knowledge Graph Explanation for: Rosacea ===

Category: inflammatory

Description:
Chronic facial condition with erythema, telangiectasia, and papules.

Top associated symptoms:
 - facial redness (id=20)
 - flushing (id=26)
 - visible blood vessels (id=78)

Related diseases:
 - Acne vulgaris | shared_symptoms=0 | category=inflammatory | reason=same_category
 - Atopic dermatitis (Eczema) | shared_symptoms=0 | category=inflammatory | reason=same_category
 - Psoriasis vulgaris | shared_symptoms=0 | category=inflammatory | reason=same_category
 - Seborrheic dermatitis | shared_symptoms=0 | category=inflammatory | reason=same_category
 - Contact dermatitis (Irritant) | shared_symptoms=0 | category=inflammatory | reason=same_category
```