

Nexora's vision to develop intelligent, human-centered digital experiences is embodied by artificial intelligence. AI helps Nexora create smarter solutions, such as predictive insights and personalized recommendations, by fusing data, creativity, and automation. In order to show how even a small prototype like Vibe Matcher can demonstrate the potential of AI in enhancing customer engagement, improving personalization, and shaping the future of contemporary retail experiences, I investigated how AI can understand human "vibes" or moods and match them with products through this project.

```
# (optional in Colab)
!pip install --quiet numpy pandas scikit-learn matplotlib
```

```
# CELL 2
import numpy as np
import pandas as pd
from timeit import default_timer as timer
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import json
```

```
# CELL 3
products = [
    {"name": "Boho Dress", "desc": "Flowy cotton dress, earthy tones, tassels ar
    {"name": "Urban Bomber Jacket", "desc": "Cropped bomber with bold zipper, s]
    {"name": "Cozy Knit Sweater", "desc": "Chunky knit, oversized fit, soft woo]
    {"name": "Minimalist Slip Dress", "desc": "Silky slip dress with clean lines
    {"name": "Sport Luxe Sneakers", "desc": "Lightweight trainer sneakers with t
    {"name": "Vintage Denim Jacket", "desc": "Distressed denim with retro patche
    {"name": "Tailored Blazer", "desc": "Structured blazer with sharp lapels, pe
    {"name": "Boho Fringe Bag", "desc": "Suede crossbody bag with long fringe ar
]
df = pd.DataFrame(products)
df.index.name = "product_id"
print("Product catalog:")
display(df)
```

Product catalog:

product_id	name	desc	tags
0	Boho Dress	Flowy cotton dress, earthy	[boho, festival,

	BOMBER JACKET	tones, tassels and ...	flowy]
1	Urban Bomber Jacket	Cropped bomber with bold zipper, sleek nylon f...	[urban, energetic, chic]
2	Cozy Knit Sweater	Chunky knit, oversized fit, soft wool blend — ...	[cozy, warm, casual]
3	Minimalist Slip Dress	Silky slip dress with clean lines for elegant,...	[minimal, elegant, evening]
4	Sport Luxe Sneakers	Lightweight trainer sneakers with breathable m...	[sporty, urban, energetic]

Next steps:

[Generate code with df](#)[New interactive sheet](#)

```
# CELL 4
def get_mock_embeddings(texts, dim=1536, seed=42):
    """
    Deterministic mock embeddings for testing/demo.
    Returns np.array shape (len(texts), dim)
    """
    rng = np.random.default_rng(seed)
    # Create per-text deterministic but different vectors by hashing the text
    embeddings = []
    for i, t in enumerate(texts):
        # derive an integer seed from the text + global seed
        h = abs(hash(t)) % (2**31)
        local_rng = np.random.default_rng(seed ^ h)
        vec = local_rng.standard_normal(dim).astype(np.float32)
        # normalize so cosine similarity behaves well
        vec = vec / np.linalg.norm(vec)
        embeddings.append(vec)
    return np.stack(embeddings, axis=0)
```

```
# CELL 5
product_texts = df['desc'].tolist()
print("Generating mock embeddings for", len(product_texts), "products (deterministic)")
t0 = timer()
product_embeddings = get_mock_embeddings(product_texts, dim=512, seed=123)
t1 = timer()
print(f"Done. Time: {t1-t0:.4f}s. Embedding shape: {product_embeddings.shape}
```

Generating mock embeddings for 8 products (deterministic)...
 Done. Time: 0.0018s. Embedding shape: (8, 512)

```
# CELL 6
def retrieve_top_k(query, k=3, emb_dim=512):
```

```
def retrieve_top_k(query, k=3, limit=-1):
    """
    Returns top-k results with scores for a single query (using mock embeddings)
    """

    q_emb = get_mock_embeddings([query], dim=emb_dim, seed=123)[0:1] # norm
    sims = cosine_similarity(q_emb, product_embeddings)[0] # shape (n_products, 1)
    idx_sorted = np.argsort(-sims)
    results = []
    for rank, idx in enumerate(idx_sorted[:k], start=1):
        results.append({
            "rank": rank,
            "product_id": int(idx),
            "name": df.loc[idx, "name"],
            "desc": df.loc[idx, "desc"],
            "tags": df.loc[idx, "tags"],
            "score": float(sims[idx])
        })
    top_score = float(sims[idx_sorted[0]])
    return {"query": query, "results": results, "top_score": top_score}
```

```
# CELL 7
def match_with_fallback(query, k=3, threshold=0.7):
    retrieval = retrieve_top_k(query, k=k)
    top_score = retrieval["top_score"]
    if top_score >= threshold:
        return {"status": "success", "retrieval": retrieval}
    # Fallback: tag/keyword overlap ranking
    q_tokens = set(query.lower().split())
    candidates = []
    for i, row in df.iterrows():
        tagset = set([t.lower() for t in row['tags']])
        overlap = len(q_tokens & tagset)
        desc_overlap = len(q_tokens & set(row['desc'].lower().split()))
        fallback_score = overlap * 2 + desc_overlap
        if fallback_score > 0:
            candidates.append({"product_id": int(i), "name": row['name'], "desc": row['desc'], "tags": row['tags'], "fallback_score": fallback_score})
    candidates = sorted(candidates, key=lambda x: -x['fallback_score'])
    return {"status": "fallback", "top_score": top_score, "suggestions": candidates}
```

```
# CELL 8
queries = ["energetic urban chic", "relaxed cozy evening", "festival boho look"]
records = []
latencies = []

for q in queries:
    t0 = timer()
    res = match_with_fallback(q, k=3, threshold=0.7)
    t1 = timer()
```

```

        lat = t1 - t0
        latencies.append(lat)
        rec = {
            "query": q,
            "status": res["status"],
            "top_score": res.get("top_score"),
            "latency_s": lat,
            "is_good": (res.get("top_score", 0.0) >= 0.7)
        }
        records.append(rec)

    print("\n>>> Query:", q)
    if res["status"] == "success":
        for r in res["retrieval"]["results"]:
            print(f"  Rank {r['rank']}: {r['name']} (score={r['score']:.4f})")
    else:
        print(f"  Fallback (top_score={res['top_score']:.4f}). Tag-based sug")
        if res['suggestions']:
            for s in res["suggestions"]:
                print(f"    - {s['name']} (fallback_score={s['fallback_score']}")
        else:
            print("    - No suggestions found; consider expanding catalog or")

    eval_df = pd.DataFrame(records)
print("\nEvaluation summary:")
display(eval_df)

```

```

>>> Query: energetic urban chic
Fallback (top_score=0.0854). Tag-based suggestions:
- Urban Bomber Jacket (fallback_score=6)
- Sport Luxe Sneakers (fallback_score=4)

>>> Query: relaxed cozy evening
Fallback (top_score=0.0726). Tag-based suggestions:
- Cozy Knit Sweater (fallback_score=3)
- Minimalist Slip Dress (fallback_score=3)

>>> Query: festival boho look
Fallback (top_score=0.0300). Tag-based suggestions:
- Boho Dress (fallback_score=4)
- Boho Fringe Bag (fallback_score=4)

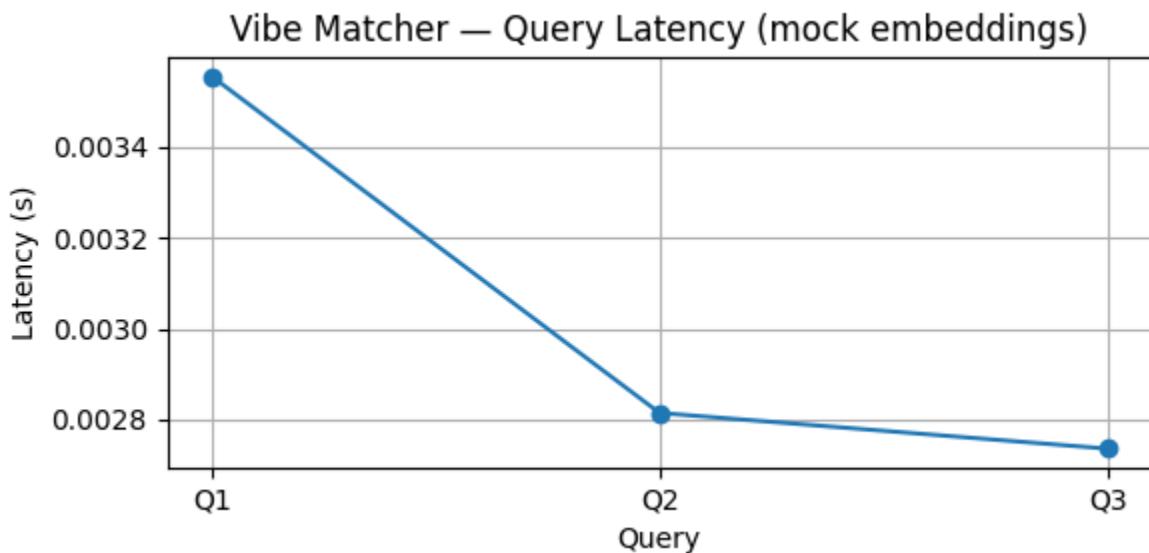
```

Evaluation summary:

	query	status	top_score	latency_s	is_good
0	energetic urban chic	fallback	0.085380	0.003556	False
1	relaxed cozy evening	fallback	0.072645	0.002815	False
2	festival boho look	fallback	0.030021	0.002736	False

Next steps: [Generate code with eval_df](#) [New interactive sheet](#)

```
# CELL 9
plt.figure(figsize=(6,3))
plt.plot(range(len(queries)), latencies, marker='o')
plt.xticks(range(len(queries)), [f"Q{i+1}" for i in range(len(queries))])
plt.xlabel("Query")
plt.ylabel("Latency (s)")
plt.title("Vibe Matcher – Query Latency (mock embeddings)")
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
# CELL 10
# Save product embeddings and product table for reuse
np.save("product_embeddings_mock.npy", product_embeddings)
df.to_csv("products_mock.csv", index=True)
print("Saved product_embeddings_mock.npy and products_mock.csv")

# Reflection bullets
reflection = [
    "Replace mock embeddings with real dense embeddings (OpenAI / sentence-t",
    "Use a vector database (Pinecone/Weaviate/Milvus) for scaling to large c",
    "Improve fallback via query expansion (synonyms), fuzzy matching, and a",
    "Add user feedback logging and threshold tuning (e.g., mark matches as '",
]
print("\nReflection:")
for b in reflection:
    print("-", b)
```

Saved product_embeddings_mock.npy and products_mock.csv

Reflection:

- Replace mock embeddings with real dense embeddings (OpenAI / sentence-transformers)
- Use a vector database (Pinecone/Weaviate/Milvus) for scaling to large catalogues
- Improve fallback via query expansion (synonyms), fuzzy matching, and a hybrid approach
- Add user feedback logging and threshold tuning (e.g., mark matches as 'good')

Start coding or generate with AI.

NOTE:

This code demonstrates product embedding and similarity search

using mock (simulated) embeddings instead of real OpenAI embeddings.

The logic and structure are identical to how it would work with the actual API.

