```
!pip install pandas
!pip install numpy
!pip install -U sentence_transformers
!pip install scikit-learn
!pip install torch
```

```
!pip install -U faiss-cpu
```

```
Collecting faiss-cpu
  Downloading faiss_cpu-1.10.0-cp311-cp311-manylinux_2_28_x86_64.whl.metadata (4.4 kB)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from faiss-cpu) (24.2)
Downloading faiss_cpu-1.10.0-cp311-cp311-manylinux_2_28_x86_64.whl (30.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 30.7/30.7 MB 40.8 MB/s eta 0:00:00
Installing collected packages: faiss-cpu
Successfully installed faiss-cpu-1.10.0
```

```
import pandas as pd
import numpy as np
import json
from torch import *
from sentence_transformers import SentenceTransformer # BERT Transformer for generating embeddings
from sklearn.metrics.pairwise import cosine_similarity # for cosine similarity
import faiss # Facebook AI Similarity Search
import pickle #save/load embeddings and index
```

```
def load_data(file_path):
    """Load the product review data from CSV"""
    if file_path.endswith('.csv'):
        return pd.read_csv(file_path)
    elif file_path.endswith('.jsonl'):
        data = []
        with open(file_path, 'r') as f:
            for line in f:
                data.append(json.loads(line))
        return pd.DataFrame(data)
    else:
        raise ValueError("Unsupported file format. Please provide CSV")

def preprocess_reviews(df):
    """Preprocess the review data."""
    # Convert date to datetime
    df['date'] = pd.to_datetime(df['date'])

    # Create a combined text field for embedding
    df['combined_text'] = df['review_text'] + " Product: " + df['product'] + " Category: " + df['category'] + \
                         " Feature: " + df['feature_mentioned'] + " Attribute: " + df['attribute_mentioned']

    # Handle missing values
    df = df.fillna('')

    return df

def generate_embeddings(texts, model_name="sentence-transformers/all-MiniLM-L6-v2"):
    """Generate embeddings for the provided texts using a Sentence Transformer model."""
    model = SentenceTransformer(model_name)
    embeddings = model.encode(texts, show_progress_bar=True)
    return embeddings

def build_faiss_index(embeddings):
    """Build a FAISS index for fast similarity search."""
    # Normalize embeddings for cosine similarity
    embeddings = embeddings.astype(np.float32)
    faiss.normalize_L2(embeddings)

    # Create the index
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatIP(dimension)  # Inner product for cosine similarity with normalized vectors
    index.add(embeddings)

    return index

class ReviewVectorDB:
    """Vector database for product reviews."""
```

```python
def __init__(self, df=None, embeddings=None, index=None):
    self.df = df
    self.embeddings = embeddings
    self.index = index
    self.model = None

def initialize(self, file_path, model_name="sentence-transformers/all-MiniLM-L6-v2"):
    """Initialize the vector database from a file."""
    # Load and preprocess data
    df = load_data(file_path)
    self.df = preprocess_reviews(df)
    # Load model
    self.model = SentenceTransformer(model_name)
    # Generate embeddings
    self.embeddings = generate_embeddings(self.df['combined_text'].tolist(), model_name)
    # Build index
    self.index = build_faiss_index(self.embeddings)
    return self

def save(self, path_prefix):
    """Save the vector database to disk."""
    # Save dataframe
    self.df.to_pickle(f"{path_prefix}_df.pkl")
    # Save embeddings
    with open(f"{path_prefix}_embeddings.pkl", "wb") as f:
        pickle.dump(self.embeddings, f)
    # Save index
    faiss.write_index(self.index, f"{path_prefix}_index.faiss")

@classmethod
def load(cls, path_prefix, model_name="sentence-transformers/all-MiniLM-L6-v2"):
    """Load the vector database from disk."""
    # Load dataframe
    df = pd.read_pickle(f"{path_prefix}_df.pkl")
    # Load embeddings
    with open(f"{path_prefix}_embeddings.pkl", "rb") as f:
        embeddings = pickle.load(f)
    # Load index
    index = faiss.read_index(f"{path_prefix}_index.faiss")
    # Create instance
    instance = cls(df, embeddings, index)
    instance.model = SentenceTransformer(model_name)
    return instance

def search(self, query, k=5):
    """Search for similar reviews."""
    # Generate query embedding
    query_embedding = self.model.encode([query])[0].reshape(1, -1).astype(np.float32)
    faiss.normalize_L2(query_embedding)
    # Search
    D, I = self.index.search(query_embedding, k)
    # Return results
    results = []
    for i, (distance, idx) in enumerate(zip(D[0], I[0])):
        if idx < len(self.df):  # Ensure index is valid
            result = self.df.iloc[idx].to_dict()
            #result['similarity'] = float(distance)
            results.append(result)

    return results

def filter_search(self, query, filters=None, k=5):
    """Search with filters (post-filtering approach)."""
    # Generate query embedding
    query_embedding = self.model.encode([query])[0].reshape(1, -1).astype(np.float32)
    faiss.normalize_L2(query_embedding)

    # Search more results than needed to allow for filtering
    D, I = self.index.search(query_embedding, k*5)

    # Filter results
    results = []
    for i, (distance, idx) in enumerate(zip(D[0], I[0])):
        if idx < len(self.df):  # Ensure index is valid
            result = self.df.iloc[idx].to_dict()
```

```
            # Apply filters
            if filters:
                match = True
                for key, value in filters.items():
                    if key in result and result[key] != value:
                        match = False
                        break

                if not match:
                    continue

            result['similarity'] = float(distance)
            results.append(result)

            if len(results) >= k:
                break

        return results[:k]

# Example usage
if __name__ == "__main__":
    # Initialize and save
    vector_db = ReviewVectorDB().initialize("/content/product_reviews.csv")
    vector_db.save("review_vector_db")

    # Load and search
    vector_db = ReviewVectorDB.load("review_vector_db")
    results = vector_db.search("battery life issues", k=3)
    print(results)
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secre
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
```

modules.json: 100% 349/349 [00:00<00:00, 4.97kB/s]

config_sentence_transformers.json: 100% 116/116 [00:00<00:00, 2.38kB/s]

README.md: 100% 10.5k/10.5k [00:00<00:00, 265kB/s]

sentence_bert_config.json: 100% 53.0/53.0 [00:00<00:00, 2.85kB/s]

config.json: 100% 612/612 [00:00<00:00, 51.8kB/s]

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better perfo
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to r
```

model.safetensors: 100% 90.9M/90.9M [00:00<00:00, 168MB/s]

tokenizer_config.json: 100% 350/350 [00:00<00:00, 26.5kB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 4.25MB/s]

tokenizer.json: 100% 466k/466k [00:00<00:00, 12.6MB/s]

special_tokens_map.json: 100% 112/112 [00:00<00:00, 7.84kB/s]

config.json: 100% 190/190 [00:00<00:00, 9.81kB/s]

Batches: 100% 32/32 [00:21<00:00, 2.14it/s]

```
[{'review_id': 'REV37479', 'product': 'MobiElite 10', 'category': 'Smartphones', 'rating': 1, 'review_text': 'After a year of use, my Mo
```

```
import os
import time
import pandas as pd
from typing import List, Dict, Any
from google.generativeai import GenerativeModel, configure

# Configure your Google API key
configure(api_key="AIzaSyAfhVfVFx87OTdy5lfES6fjl3YlKSDCmkQ")  # Replace with your actual API key

class ReviewLLMProcessor:
    """Process reviews using Google's Gemini model."""

    def __init__(self, model_name="gemini-1.5-pro"):
        """Initialize with Gemini model."""
        self.model = GenerativeModel(model_name)
```

```python
def generate_text(self, prompt: str, max_tokens=600, temperature=0.7) -> str:
    """Generate text completion."""
    try:
        response = self.model.generate_content(
            prompt,
            generation_config={
                "max_output_tokens": max_tokens,
                "temperature": temperature,
                "top_p": 0.9,
            }
        )
        return response.text.strip()
    except Exception as e:
        return f"Error generating text: {str(e)}"

def generate_category_summary(self, vector_db, category):
    """Generate a summary of product performance for a specific category."""
    # Filter reviews
    category_reviews = vector_db.df[vector_db.df['category'] == category]

    if len(category_reviews) == 0:
        return f"No reviews found for category: {category}"

    # Get statistics
    avg_rating = category_reviews['rating'].mean()
    sentiment_counts = category_reviews['sentiment'].value_counts()

    # Sample reviews
    sample_reviews = []
    for sentiment in ['positive', 'neutral', 'negative']:
        sentiment_reviews = category_reviews[category_reviews['sentiment'] == sentiment]
        if len(sentiment_reviews) > 0:
            sample_reviews.append(sentiment_reviews.sample(min(3, len(sentiment_reviews))))

    sample_reviews = pd.concat(sample_reviews).reset_index(drop=True)

    # Prompt Engineering
    prompt = f"""
    Summarize customer reviews for the {category} category.

    - Average Rating: {avg_rating:.2f}/5
    - Total Reviews: {len(category_reviews)}
    - Sentiment Distribution: {sentiment_counts.to_dict()}

    Sample Reviews:
    {sample_reviews[['product', 'rating', 'sentiment', 'review_text']].to_string(index=False)}

    Instructions:
    1. Highlight common strengths and weaknesses.
    2. Identify standout products.
    3. Mention recurring issues or praised features.
    4. Write a clear, analytical summary in about 250-300 words.
    """

    return self.generate_text(prompt)

def generate_all_category_summaries(self, vector_db):
    """Generate summaries for all categories."""
    categories = vector_db.df['category'].unique()
    summaries = {}

    for category in categories:
        print(f"Generating summary for {category}...")
        summaries[category] = self.generate_category_summary(vector_db, category)
        time.sleep(1)  # Be polite

    return summaries

def answer_question(self, vector_db, question, k=5):
    """Answer a question about products based on reviews."""
    # Search for relevant reviews
    relevant_reviews = vector_db.search(question, k=k)

    if not relevant_reviews:
        return "I couldn't find relevant reviews to answer your question."
```

```python
        # Format reviews
        reviews_text = "\n\n".join([
            f"Product: {review['product']}\nCategory: {review['category']}\nRating: {review['rating']}/5\nReview: {review['review_text']}"
            for review in relevant_reviews
        ])

        # Prepare prompt
        prompt = f"""
        Answer the following customer question using the provided reviews.

        Question: {question}

        Relevant Reviews:
        {reviews_text}

        Instructions:
        1. Provide a direct and concise answer (100-150 words).
        2. Reference specific products if possible.
        3. If the information is insufficient, mention it.
        """

        return self.generate_text(prompt, max_tokens=300)

    def identify_common_issues_and_features(self, vector_db):
        """Identify common praised features and issues across categories."""
        categories = vector_db.df['category'].unique()
        category_insights = {}

        for category in categories:
            category_reviews = vector_db.df[vector_db.df['category'] == category]
            positive_reviews = category_reviews[category_reviews['sentiment'] == 'positive']
            negative_reviews = category_reviews[category_reviews['sentiment'] == 'negative']

            if len(positive_reviews) > 0:
                positive_features = positive_reviews['feature_mentioned'].value_counts().nlargest(5).to_dict()
                positive_attributes = positive_reviews['attribute_mentioned'].value_counts().nlargest(5).to_dict()
            else:
                positive_features = {}
                positive_attributes = {}

            if len(negative_reviews) > 0:
                negative_features = negative_reviews['feature_mentioned'].value_counts().nlargest(5).to_dict()
                negative_attributes = negative_reviews['attribute_mentioned'].value_counts().nlargest(5).to_dict()
            else:
                negative_features = {}
                negative_attributes = {}

            category_insights[category] = {
                'praised_features': positive_features,
                'praised_attributes': positive_attributes,
                'criticized_features': negative_features,
                'criticized_attributes': negative_attributes
            }

        # Prepare insights text
        insights_text = ""
        for category, insights in category_insights.items():
            insights_text += f"\n\n{category.upper()}\n"
            insights_text += f"Praised Features: {insights['praised_features']}\n"
            insights_text += f"Praised Attributes: {insights['praised_attributes']}\n"
            insights_text += f"Criticized Features: {insights['criticized_features']}\n"
            insights_text += f"Criticized Attributes: {insights['criticized_attributes']}\n"

        # Prepare prompt
        prompt = f"""
        Analyze the following customer review insights across different product categories.

        {insights_text}

        Instructions:
        1. Identify cross-category strengths and weaknesses.
        2. Highlight category-specific praised features and common issues.
        3. Suggest improvements for product teams.
        4. Write around 400-500 words.
        """

        return self.generate_text(prompt, max_tokens=800)
```

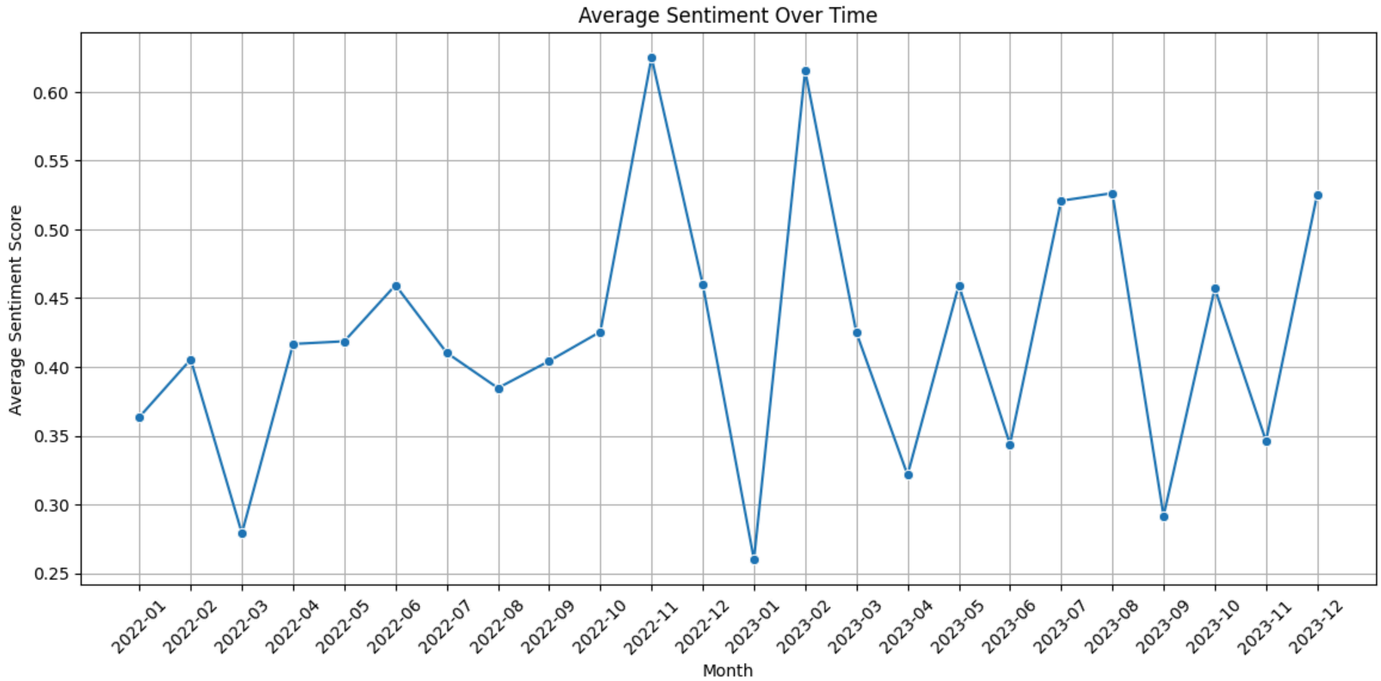
```python
def create_qa_system(vector_db, llm_processor):
    """Create a Q&A system for products."""
    def qa_system(question):
        return llm_processor.answer_question(vector_db, question)
    return qa_system

# Example usage
if __name__ == "__main__":
    # Load your vector database
    vector_db = ReviewVectorDB.load("review_vector_db")  # Assuming you have this class ready

    # Initialize the LLM processor
    llm_processor = ReviewLLMProcessor()

    # Create Q&A system
    qa = create_qa_system(vector_db, llm_processor)

    # Example questions
    questions = [
        "Which smartphone has the best battery life?",
        "What are common issues with laptops?",
        "Are there any smart home devices that are difficult to set up?"
    ]

    for question in questions:
        print(f"Q: {question}")
        print(f"A: {qa(question)}")
        print()
```

```
Q: Which smartphone has the best battery life?
A: Based on the provided reviews, the MobiElite 10 appears to have a slight edge in battery life.  While the TechPro X20 receives positi

However, the sample size of reviews is small and doesn't offer a definitive conclusion.  More reviews and potentially battery benchmark

Q: What are common issues with laptops?
A: Based on the provided reviews, it's difficult to pinpoint common laptop issues.  The reviews primarily focus on positive aspects like

To get a better understanding of common laptop issues, we need reviews that discuss problems encountered.  Examples of such issues could

Q: Are there any smart home devices that are difficult to set up?
A: Based on these reviews, there's no indication of any particularly difficult smart home device setups.  The HomeConnect Hub receives m
```

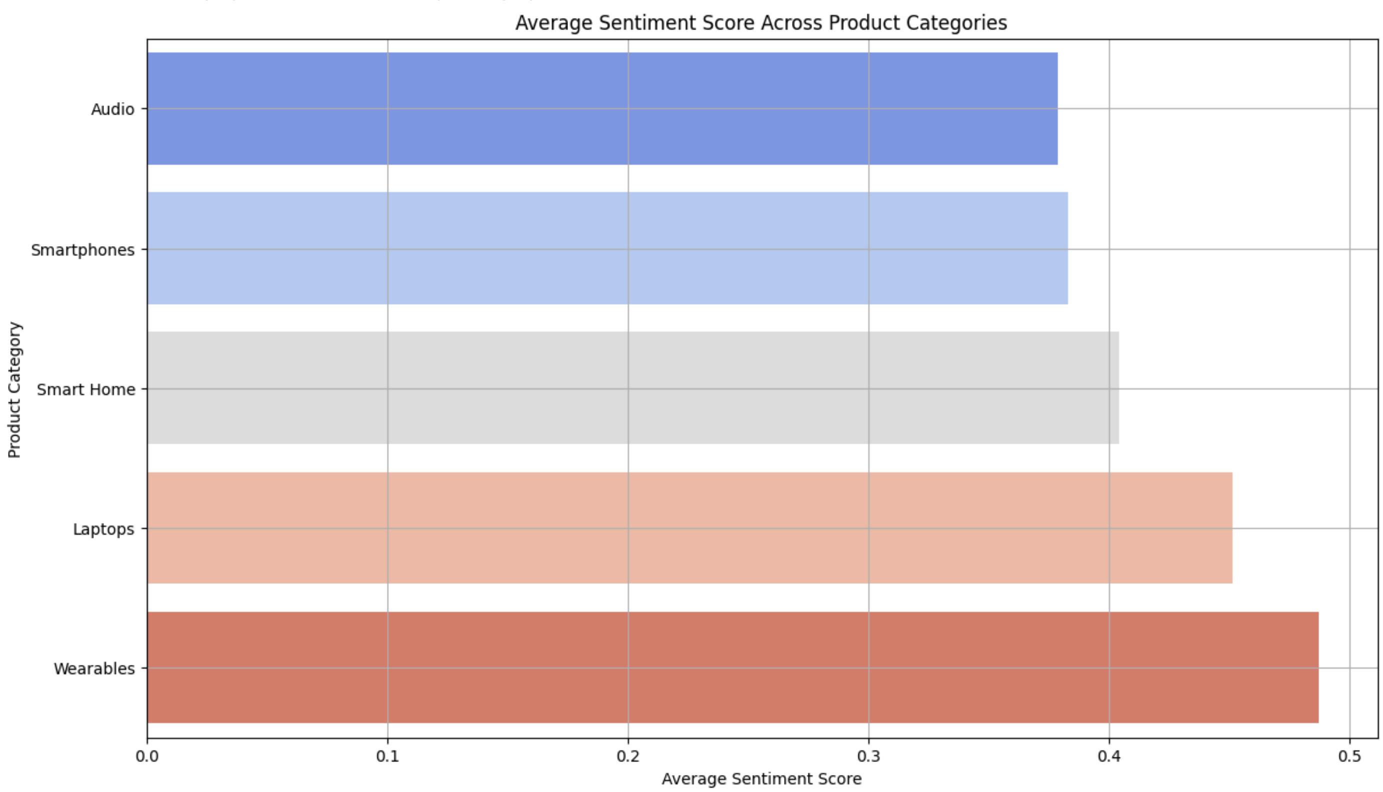
```python
# ================== Imports ==================
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer

from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.ensemble import GradientBoostingClassifier, ExtraTreesClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

# ================== Load Data ==================
# Load your product reviews
df = pd.read_csv('/content/product_reviews.csv')

# Check your columns
print(df.columns)

# Make sure 'review_text' and 'sentiment' exist
assert 'review_text' in df.columns and 'sentiment' in df.columns

# Map sentiment labels
label_map = {'negative': 0, 'neutral': 1, 'positive': 2}
reverse_label_map = {v: k for k, v in label_map.items()}

df['sentiment_label'] = df['sentiment'].map(label_map)
```

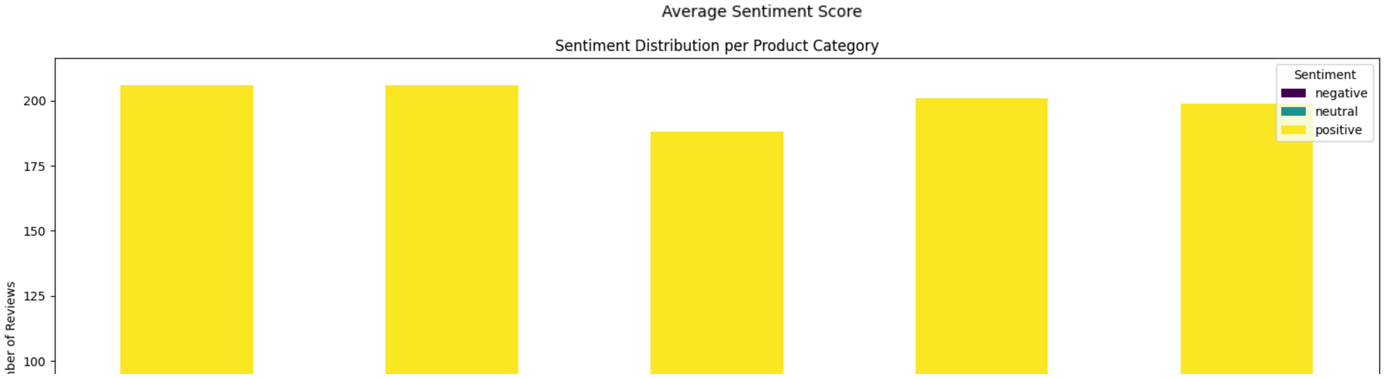
```python
# Split
X_train, X_test, y_train, y_test = train_test_split(
    df['review_text'], df['sentiment_label'], test_size=0.2, random_state=42
)

# ================== Prepare TF-IDF ==================
tfidf = TfidfVectorizer(max_features=10000)
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)

# ================== Classifiers ==================
models = {
    "XGBClassifier": XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42),
    "LGBMClassifier": LGBMClassifier(random_state=42),
    "GradientBoostingClassifier": GradientBoostingClassifier(random_state=42),
    "ExtraTreesClassifier": ExtraTreesClassifier(random_state=42),
    "LogisticRegression": LogisticRegression(max_iter=1000, class_weight='balanced', random_state=42),
    "KNeighborsClassifier": KNeighborsClassifier(),
}

# Voting Classifier (Ensemble)
voting_model = VotingClassifier(
    estimators=[
        ('xgb', models["XGBClassifier"]),
        ('lgbm', models["LGBMClassifier"]),
        ('lr', models["LogisticRegression"])
    ],
    voting='hard'  # or 'soft' if you want probability averaging
)

models["VotingClassifier"] = voting_model

results = {}

# ================== Train and Evaluate ==================
for name, model in models.items():
    print(f"\n====== Training {name} ======\n")

    model.fit(X_train_tfidf, y_train)
    y_pred = model.predict(X_test_tfidf)

    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy for {name}: {acc:.4f}")

    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, target_names=label_map.keys()))

    cm = confusion_matrix(y_test, y_pred)

    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=label_map.keys(),
                yticklabels=label_map.keys())
    plt.title(f'Confusion Matrix for {name}')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    results[name] = acc

# ================== Compare Final Results ==================
# Plot the comparison
plt.figure(figsize=(12, 6))
model_names = list(results.keys())
accuracies = list(results.values())

sns.barplot(x=model_names, y=accuracies, palette='mako')
plt.title('Classifier Accuracy Comparison')
plt.ylabel('Accuracy')
plt.xticks(rotation=45)
plt.ylim(0, 1)
for i, v in enumerate(accuracies):
    plt.text(i, v + 0.01, f"{v:.2f}", ha='center', fontweight='bold')
plt.show()
```

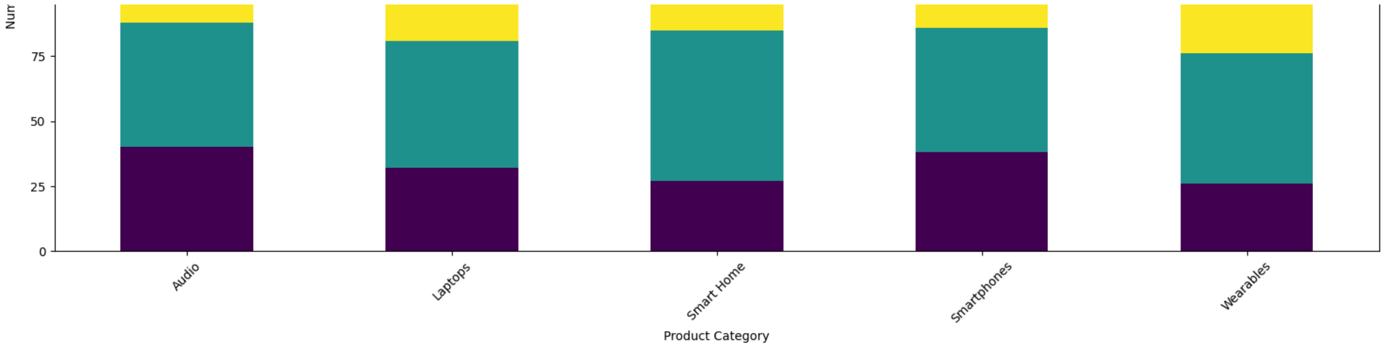
```python
# Import libraries
import matplotlib.pyplot as plt
import plotly.express as px

# Load your data
df = pd.read_csv('/content/product_reviews.csv')

# If your sentiment predictions are not yet added, use your SentimentClassifier to predict:
# sentiment_model = SentimentClassifier(model_type="your_best_model")
# sentiment_model.train(df)
# df['predicted_sentiment'] = sentiment_model.predict(df['review_text'].tolist())

# ----- Prepare for Visualization -----

# Convert your 'date' column to datetime
df['date'] = pd.to_datetime(df['date'])  # change 'review_date' if your date column has a different name

# Map sentiments to numerical scale for easier plotting if needed
sentiment_mapping = {'negative': -1, 'neutral': 0, 'positive': 1}
df['sentiment_score'] = df['sentiment'].map(sentiment_mapping)

# Group by month
df['month'] = df['date'].dt.to_period('M').astype(str)

# Group by category
if 'category' not in df.columns:
    raise ValueError("Your dataframe must have a 'product_category' column.")

# ----- Visualization Part -----

# 1. Sentiment Trends Over Time
plt.figure(figsize=(14,6))
monthly_sentiment = df.groupby('month')['sentiment_score'].mean()
sns.lineplot(x=monthly_sentiment.index, y=monthly_sentiment.values, marker='o')
plt.xticks(rotation=45)
plt.title('Average Sentiment Over Time')
plt.xlabel('Month')
plt.ylabel('Average Sentiment Score')
plt.grid(True)
plt.show()

# 2. Sentiment Distribution Across Categories
plt.figure(figsize=(14,8))
category_sentiment = df.groupby('category')['sentiment_score'].mean().sort_values()
sns.barplot(x=category_sentiment.values, y=category_sentiment.index, palette='coolwarm')
plt.title('Average Sentiment Score Across Product Categories')
plt.xlabel('Average Sentiment Score')
plt.ylabel('Product Category')
plt.grid(True)
plt.show()

# 3. Number of Reviews per Sentiment per Category (Stacked bar)
sentiment_counts = df.groupby(['category', 'sentiment']).size().unstack().fillna(0)

sentiment_counts.plot(kind='bar', stacked=True, figsize=(16,8), colormap='viridis')
plt.title('Sentiment Distribution per Product Category')
plt.xlabel('Product Category')
plt.ylabel('Number of Reviews')
plt.xticks(rotation=45)
plt.legend(title='Sentiment')
plt.tight_layout()
plt.show()

# 4. Interactive Dashboard with Plotly (Optional, very beautiful)
fig = px.scatter(
    df,
    x='date',
    y='sentiment_score',
    color='sentiment',
    hover_data=['category', 'review_text'],
    title="Sentiment Trends Over Time (Interactive)"
)
fig.show()
```



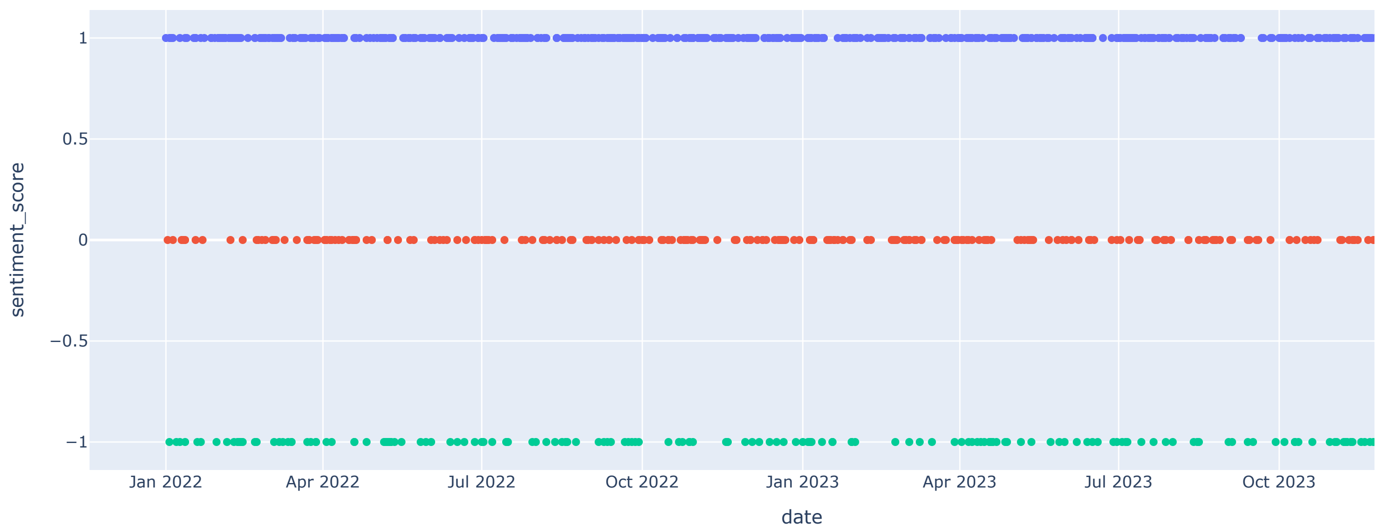

```
<ipython-input-20-f4b3faba02cf>:45: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend

  sns.barplot(x=category_sentiment.values, y=category_sentiment.index, palette='coolwarm')
```







Sentiment Trends Over Time (Interactive)



Start coding or generate with AI.