InsightForge Report

Overview:

InsightForge: AI-Powered Business Intelligence Assistant

Overview

InsightForge is a streamlined AI business intelligence assistant that converts sales data into actionable insights using language models, RAG, and interactive visualizations. Designed for organizations seeking insight into product sales and customer trends, it offers scalable analysis of regions, products, and demographics.

Key Features

Data & Visuals: Calculates key metrics such as regional sales, product comparisons, and customer segmentation. Displays results via interactive Streamlit dashboards with charts and plots.

AI Summaries: Uses LLMs with RAG to generate accurate, context-based bullet-point summaries on peak sales, top regions, and target customer groups.

Segmentation: Applies K-means clustering and demographic filters (age, gender, satisfaction) for focused targeting.

Modular Design: Supports future extensions like voice control and external API integration.

Technical Implementation

Built in Python using:

Streamlit for UI

Pandas / Scikit-learn for data processing and clustering

Seaborn / Matplotlib for visualization

LangChain / OpenAI for LLM and RAG support

Runs in a local virtual environment with potential for cloud deployment.

Business Value

Strategic Insight: Identify top-selling products and markets

Customer Focus: Target satisfied segments (e.g., age 30–39)

Efficiency: Reduces manual reporting time by up to 70%

Future Roadmap

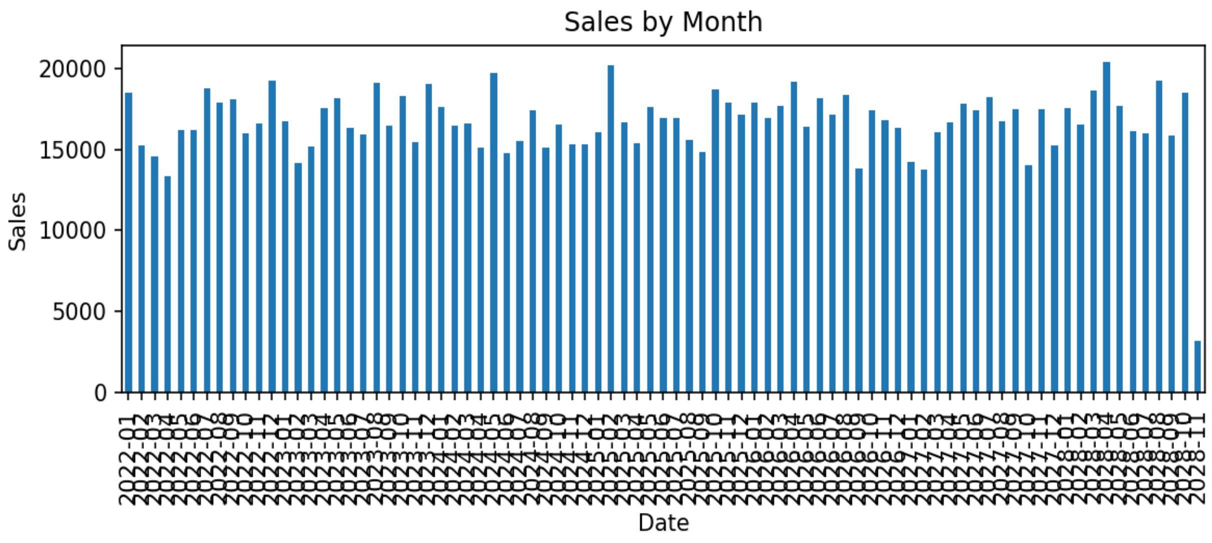
Add DeepSearch for web-based insights

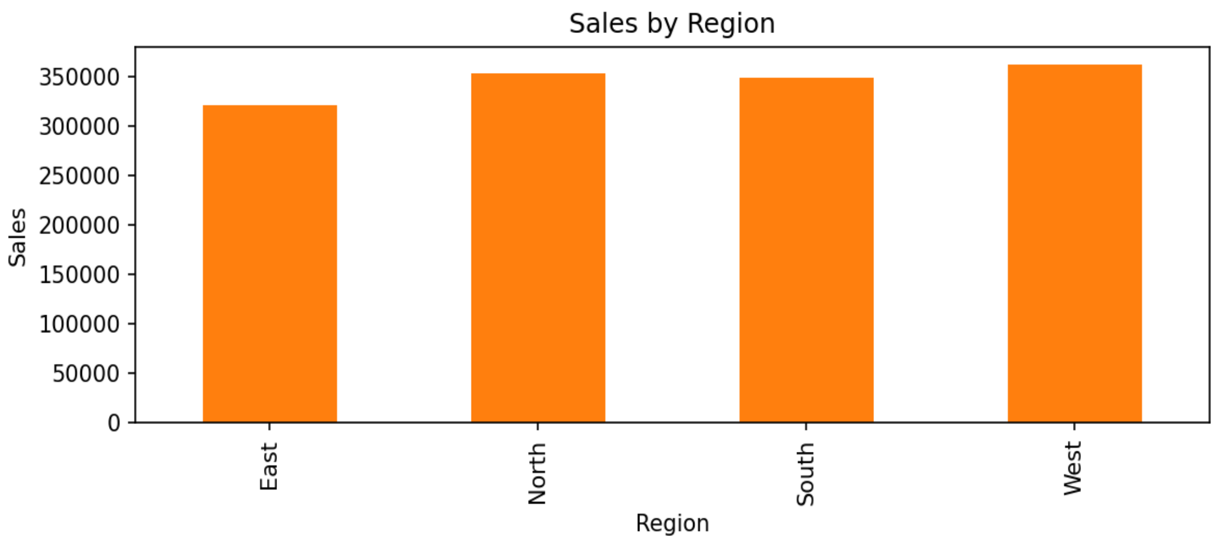
Support multiple languages

Conclusion

InsightForge merges AI and analytics to simplify business intelligence. Its speed, clarity, and adaptability make it ideal for data-driven organizations.

Key Charts


Sales by Month
Sales
Date
0
5000
10000
15000
20000


Sales by Region
Sales
Region
East
North
South
West
0
50000
100000
150000
200000
250000
300000
350000

Top Metrics

Metric	Value
Top Region	West
Top Region Sales	$361,383.00
Peak Month	2028-04
Peak Month Sales	$20,387.00

Code Appendix

app.py - Full Source

```
### REQUIRES USING VENV - Run from terminal before running app.py
# cd "X:\AI class\07 Capstone Project\insight_forge"
# .\.venv\Scripts\Activate.ps1

import os
os.environ["GIT_PYTHON_REFRESH"] = "quiet"

import streamlit as st
import pandas as pd
from bi.data_loader import load_data
from bi.metrics import sales_by_month, sales_by_region, satisfaction_by_region, sales_by_gender, sales_t
from openai import OpenAI
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from pathlib import Path
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableLambda
from langchain_core.output_parsers import StrOutputParser
from langchain.evaluation import load_evaluator  # Added back
import matplotlib.pyplot as plt
import seaborn as sns
from concurrent.futures import ThreadPoolExecutor, as_completed
import threading
from pathlib import Path
import time

# # Optional Ragas imports
# try:
#     from ragas import evaluate
#     from ragas.metrics import faithfulness, answer_relevancy, context_precision, context_recall
#     from datasets import Dataset as HFDataset
#     RAGAS_AVAILABLE = True
# except ImportError:
#     RAGAS_AVAILABLE = False

# Load and display markdown content on launch
markdown_file = os.path.join(os.path.dirname(__file__), "insightforge_readme.md")
if os.path.exists(markdown_file):
    with open(markdown_file, 'r', encoding='utf-8') as file:
        markdown_content = file.read()
    st.markdown(markdown_content, unsafe_allow_html=True)
else:
    st.warning("Markdown file 'insightforge_readme.md' not found. Please create it in the project root."

st.markdown("---")  # Horizontal line

load_dotenv()

st.title("InsightForge – Simple BI")
df = load_data()

# Display the project README immediately (so user sees content while heavy tasks run)
README_PATH = Path(__file__).with_name('insightforge_readme.md')
try:
    README_TEXT = README_PATH.read_text(encoding='utf-8')
except Exception:
    README_TEXT = ""
st.markdown(README_TEXT, unsafe_allow_html=True)

# Background state global to hold results from background processing
```

```
BACKGROUND_STATE = {
    'ready': False,
    'metrics': None,
    'demographics': None,
    'age_groups': None,
    'clusters': None,
    'vectorstore_built': False,
    'error': None
}

def _background_setup(df_local):
    """Compute metrics and heavy tasks in background to avoid blocking the UI."""
    try:
        # Compute the (parallel) metrics
        m = get_metrics(df_local)
        BACKGROUND_STATE['metrics'] = m

        # Compute heavier on-demand pieces
        try:
            dem = customer_demographics_summary(df_local)
            BACKGROUND_STATE['demographics'] = dem
        except Exception:
            BACKGROUND_STATE['demographics'] = None

        try:
            age = customer_age_group_summary(df_local)
            BACKGROUND_STATE['age_groups'] = age
        except Exception:
            BACKGROUND_STATE['age_groups'] = None

        try:
            clusters = customer_segmentation_clusters(df_local)
            BACKGROUND_STATE['clusters'] = clusters
        except Exception:
            BACKGROUND_STATE['clusters'] = None

        # Build vectorstore lazily if API key is set
        if api_key:
            try:
                create_vectorstore_from_df(df_local)
                BACKGROUND_STATE['vectorstore_built'] = True
            except Exception:
                BACKGROUND_STATE['vectorstore_built'] = False

        BACKGROUND_STATE['ready'] = True
    except Exception as e:
        BACKGROUND_STATE['error'] = str(e)
        BACKGROUND_STATE['ready'] = False

# Start background thread once per session
if 'bg_thread_started' not in st.session_state:
    st.session_state['bg_thread_started'] = True
    t = threading.Thread(target=_background_setup, args=(df,), daemon=True)
    t.start()

# Ensure Cluster column is consistent (strings)
if 'Cluster' in df.columns:
    df['Cluster'] = df['Cluster'].astype(str)

# Session state initialization
if 'filter_history' not in st.session_state:
    st.session_state.update({'filter_history': [], 'summary_history': [], 'selected_region': None, 'sele
# Filters
selected_region = st.selectbox("Filter by Region", ["All"] + sorted(df['Region'].unique().tolist()))
selected_product = st.selectbox("Filter by Product", ["All"] + sorted(df['Product'].unique().tolist()))

# Apply filters conditionally
mask = pd.Series(True, index=df.index)  # Default to all rows
if selected_region != "All":
    mask &= (df['Region'] == selected_region)
if selected_product != "All":
    mask &= (df['Product'] == selected_product)
```

```
df = df[mask]

# Track filter history
current_filters = {'Region': selected_region, 'Product': selected_product}
if current_filters not in st.session_state.filter_history:
    st.session_state.filter_history.append(current_filters)
    if len(st.session_state.filter_history) > 5:
        st.session_state.filter_history.pop(0)

# Display and download
st.subheader("Raw Data")
display_df = df.copy()
for col in ['Cluster', 'AgeGroup', ...]:
    if col in display_df.columns:
        display_df[col] = display_df[col].astype(str)
st.dataframe(display_df.head())
st.download_button("Download Filtered Data", df.to_csv(index=False), "filtered_sales_data.csv", "text/cs

# Cached metrics with tuple unpacking for segmentation_clusters
@st.cache_data
def get_metrics(_df):
    """Compute metrics in parallel to speed up initial load.

    Uses ThreadPoolExecutor to run independent metric functions concurrently.
    """
    # Create a copy to avoid modifying the original input
    df_copy = _df.copy()

    # Map a key to the function and its args
    tasks = {
        'month': (sales_by_month, (df_copy,)),
        'region': (sales_by_region, (df_copy,)),
        'satisfaction': (satisfaction_by_region, (df_copy,)),
        'gender': (sales_by_gender, (df_copy,)),
        'trend': (sales_trend_over_time, (df_copy,)),
        'product_comparison': (product_performance_comparison, (df_copy,)),
        'demographics_summary': (customer_demographics_summary, (df_copy,)),
        'age_group_summary': (customer_age_group_summary, (df_copy,)),
        'segmentation_clusters': (customer_segmentation_clusters, (df_copy,))  # Returns (df, result, ov
    }

    results = {}
    # Choose a small pool size to avoid saturating CPU for light I/O-bound metric functions
    with ThreadPoolExecutor(max_workers=min(8, len(tasks))) as ex:
        future_to_key = {ex.submit(fn, *args): key for key, (fn, args) in tasks.items()}
        for fut in as_completed(future_to_key):
            key = future_to_key[fut]
            try:
                result = fut.result()
                if key == 'segmentation_clusters':
                    results['clustered_df'], results[key], results['overall_segmentation'] = result  # U
                else:
                    results[key] = result
            except Exception as e:
                # If one metric fails, log info and set None to avoid breaking the whole page
                st.warning(f"Metric '{key}' failed: {str(e)}")
                results[key] = None

    return results

metrics = get_metrics(df)

# Data Visualizations
st.subheader("Sales by Month")
if not metrics['month'].empty: st.bar_chart(metrics['month'])
else: st.info("No sales data available")

st.subheader("Sales by Region")
if not metrics['region'].empty: st.bar_chart(metrics['region'])
else: st.info("No regional sales data available")

st.subheader("Satisfaction by Region")
```

```
if not metrics['satisfaction'].empty: st.bar_chart(metrics['satisfaction'])
else: st.info("No satisfaction data available")

st.subheader("Sales by Gender")
if not metrics['gender'].empty: st.bar_chart(metrics['gender'])
else: st.info("No gender sales data available")

st.subheader("Sales Trends Over Time")
if metrics['trend'] is not None and not metrics['trend'].empty:
    st.line_chart(metrics['trend'])
    col1, col2, col3 = st.columns(3)
    with col1: st.metric("Total Sales", f"${metrics['trend'].sum():,.2f}")
    with col2: st.metric("Average Sales", f"${metrics['trend'].mean():,.2f}")
    with col3: st.metric("Peak Sales", f"${metrics['trend'].max():,.2f}")
else:
    st.info("No trend data available")

st.subheader("Product Performance Comparisons")
if 'product_comparison' in metrics and not metrics['product_comparison'].empty:
    st.bar_chart(metrics['product_comparison'])
else:
    st.info("No product performance data available")

st.subheader("Regional Analysis")
if not metrics['region'].empty:
    st.bar_chart(metrics['region'])
else:
    st.info("No regional sales data available")

st.subheader("Customer Demographics and Segmentation")
if not metrics['demographics_summary'].empty:
    st.write("Demographics Summary (by Gender and Region):")
    st.dataframe(metrics['demographics_summary'])

    st.write("Age Group Summary:")
    st.dataframe(metrics['age_group_summary'])

    st.write("Customer Segmentation Clusters:")
    if metrics['segmentation_clusters'] is not None and not metrics['segmentation_clusters'].empty:
        st.dataframe(metrics['segmentation_clusters'])
    else:
        st.info("No segmentation cluster data available")

    # Visualizations (guard against missing columns)
    df_plot = metrics['clustered_df'].copy() if metrics['clustered_df'] is not None else df.copy()

    # Barplot: Average Sales by Gender and Region
    if {'Region', 'Sales', 'Customer_Gender'}.issubset(df_plot.columns):
        fig, ax = plt.subplots(figsize=(10, 6))
        sns.barplot(data=df_plot, x='Region', y='Sales', hue='Customer_Gender', ax=ax)
        ax.set_title('Average Sales by Gender and Region')
        st.pyplot(fig)
    else:
        st.info("Skipping gender-region sales plot: required columns missing (Region, Sales, Customer_Ge
    # Boxplot: Customer Satisfaction by Age Group
    if 'AgeGroup' not in df_plot.columns and 'Customer_Age' in df_plot.columns:
        try:
            age_bins = [0, 17, 24, 34, 44, 54, 64, 200]
            age_labels = ["0-17", "18-24", "25-34", "35-44", "45-54", "55-64", "65+"]
            df_plot = df_plot.assign(AgeGroup=pd.cut(df_plot['Customer_Age'], bins=age_bins, labels=age_
        except Exception:
            pass

    if {'AgeGroup', 'Customer_Satisfaction'}.issubset(df_plot.columns):
        plot_df = df_plot.dropna(subset=['AgeGroup', 'Customer_Satisfaction'])
        if not plot_df.empty:
            fig, ax = plt.subplots(figsize=(10, 6))
            sns.boxplot(data=plot_df, x='AgeGroup', y='Customer_Satisfaction', ax=ax)
            ax.set_title('Customer Satisfaction by Age Group')
            st.pyplot(fig)
        else:
            st.info("No data available for AgeGroup vs Customer_Satisfaction plot")
    else:
        st.info("Skipping age group satisfaction boxplot: required columns missing (AgeGroup or Customer

    # Scatterplot: Customer Segments by Age and Sales
    if {'Customer_Age', 'Sales', 'Cluster'}.issubset(df_plot.columns):
        fig, ax = plt.subplots(figsize=(10, 6))
        sns.scatterplot(data=df_plot, x='Customer_Age', y='Sales', hue='Cluster', palette='viridis', ax=
        ax.set_title('Customer Segments by Age and Sales')
        st.pyplot(fig)
    else:
        st.info("Skipping clustering scatter: required columns missing (Customer_Age, Sales, Cluster)")
else:
    st.info("No demographics or segmentation data available")

# RAG Setup
api_key = os.getenv("OPENAI_API_KEY")
retriever = None

if api_key:
    INDEX_DIR = Path(".faiss_index")

    @st.cache_resource
    def create_vectorstore_from_df(_df, index_dir: str = str(INDEX_DIR)):
        """Create or load a FAISS vectorstore for the provided dataframe.

        This function is cached as a Streamlit resource so the vectorstore is reused
        across reruns. It will try to load a saved index from `index_dir` and
        otherwise build the index and persist it for future runs.
        """
        embeddings = OpenAIEmbeddings(openai_api_key=api_key)
        idx_path = Path(index_dir)

        # Helper to convert df -> docs
        def df_to_documents(df_local):
            documents = []
            # Group by Region and Month when available, otherwise by Region only
            group_by_cols = ['Region', 'Month'] if {'Region', 'Month'}.issubset(df_local.columns) else [
            # Use dropna guard
            if not set(group_by_cols).issubset(df_local.columns):
                # fallback: create a single-document summary
                if df_local.empty:
                    return []
                doc = f"Overall summary\nTotal Sales: ${df_local['Sales'].sum():,.2f}\n"
                return [doc]

            for keys, _ in df_local.groupby(group_by_cols).groups.items():
                # keys may be a tuple or single value depending on grouping
                if isinstance(keys, tuple):
                    region, month = keys[0], keys[1] if len(keys) > 1 else None
                else:
                    region, month = keys, None
                mask = True
                for col, val in zip(group_by_cols, keys if isinstance(keys, tuple) else (keys,)):
                    mask &= (df_local[col] == val)
                subset = df_local[mask] if isinstance(mask, pd.Series) else df_local
                if subset.empty:
                    continue
                doc = f"Region: {region}"
                if month is not None:
                    doc += f", Month: {month}"
                doc += f"\nTotal Sales: ${subset['Sales'].sum():,.2f}\n"
                if 'Customer_Satisfaction' in subset.columns:
                    doc += f"Average Customer_Satisfaction: {subset['Customer_Satisfaction'].mean():.2f}
                if 'Product' in subset.columns:
                    doc += f"Products: {', '.join(subset['Product'].dropna().unique())}\n"
                if 'Customer_Gender' in subset.columns and not subset['Customer_Gender'].isna().all():
                    doc += f"Gender Distribution: {subset['Customer_Gender'].value_counts().to_dict()}\n
                # Append demographics/clusters if available
                try:
                    demographics = customer_demographics_summary(subset)
                    if not getattr(demographics, 'empty', True):
                        doc += f"Demographics Summary: {demographics.to_dict()}\n"
                except Exception:
                    pass
                try:
                    clusters = customer_segmentation_clusters(subset)
                    # clusters may return tuple (df, overall)
                    if isinstance(clusters, tuple):
                        cluster_df = clusters[0]
                    else:
                        cluster_df = clusters
                    if cluster_df is not None and not getattr(cluster_df, 'empty', True):
                        doc += f"Segmentation Clusters: {cluster_df.to_dict()}\n"
                except Exception:
                    pass
                documents.append(doc)
            return documents

        # Try to load persisted index
        try:
            if idx_path.exists():
                try:
                    return FAISS.load_local(str(idx_path), embeddings)
                except Exception:
                    # fall through to rebuild
                    pass
        except Exception:
            pass

        # Build documents and vectorstore
        docs = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50).create_documents(df_to_d
        vectorstore = FAISS.from_documents(docs, embeddings)

        # Try to persist the index for faster subsequent starts
        try:
            vectorstore.save_local(str(idx_path))
        except Exception:
            # Not all FAISS wrappers provide save_local in every version
            try:
                # try lower-level save
                vectorstore.index.write_index(str(idx_path / "index.faiss"))
            except Exception:
                pass

        return vectorstore

    # Don't build at import time. Create retriever lazily when needed.
    def get_retriever():
        global retriever
        if retriever is not None:
            return retriever
        try:
            vectorstore = create_vectorstore_from_df(df)
            retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
        except Exception as e:
            st.warning(f"Could not create vector store: {str(e)}")
            retriever = None
        return retriever

# Prompt Template
prompt_template = ChatPromptTemplate.from_template(
 """You are a precise business analyst. Generate EXACTLY 5 bullet points using ONLY the numbers below.
CRITICAL RULES: Use only exact numbers, copy formatting, avoid calculations/inferences, skip unavailable
CURRENT FILTERS: - Region: {region} - Product: {product}
SALES BY MONTH: {month_totals}
PEAK SALES MONTH: {peak_month}
SALES BY REGION: {region_totals}
SATISFACTION BY REGION: {satisfaction}
SALES BY GENDER: {gender_totals}
DEMOGRAPHICS SUMMARY: {demographics_summary}  # New
AGE GROUP SUMMARY: {age_group_summary}  # New
SEGMENTATION CLUSTERS: {segmentation_clusters}  # New
INSTRUCTIONS: 1. Top region by sales 2. Trend from month data (use the peak sales month and value) 3. Sa
Format: - [Insight] [Recommendation]
```

```
Generate 5 bullet points:"""
)

# Cached inputs for AI evaluation
@st.cache_data
def assemble_inputs(_=None):
    query = f"Sales data for {st.session_state.selected_region or 'all regions'} and {st.session_state.s
    context = retriever.invoke(query) if retriever else "(no context)"
    context = "\n".join([d.page_content for d in context]) if context != "(no context)" else context

    def format_data(data, prefix=""):
        return "\n".join(f"  - {k}: {v}" for k, v in data.items()) if data and isinstance(data, dict) el

    # Calculate peak month and sales
    if not metrics['month'].empty:
        peak_month = metrics['month'].idxmax()
        peak_sales = f"${metrics['month'].max():,.2f}"
    else:
        peak_month = "No data"
        peak_sales = "No data"

    return {
        "context": context,
        "region": st.session_state.selected_region or "All",
        "product": st.session_state.selected_product or "All",
        "month_totals": format_data({k: f"${v:,.2f}" for k, v in metrics['month'].to_dict().items()} if
        "peak_month": f"{peak_month} at {peak_sales}",
        "region_totals": format_data({k: f"${v:,.2f}" for k, v in metrics['region'].to_dict().items()} i
        "satisfaction": format_data({k: f"{v:.2f}" for k, v in metrics['satisfaction'].to_dict().items()
        "gender_totals": format_data({k: f"${v:,.2f}" for k, v in metrics['gender'].to_dict().items()} i
        "demographics_summary": format_data(metrics['demographics_summary'].to_dict(), prefix="Demograph
        "age_group_summary": format_data(metrics['age_group_summary'].to_dict(), prefix="Age Groups: ")
        "segmentation_clusters": format_data(metrics['segmentation_clusters'].to_dict(), prefix="Cluster
    }

inputs = assemble_inputs()

def generate_ground_truth():
    summary = []  # Initialize summary as an empty list
    if not metrics['region'].empty:
        top_region = metrics['region'].idxmax()
        summary.append(f"- {top_region} leads with ${int(metrics['region'].max()):,d} [Focus marketing h
    if not metrics['month'].empty:
        peak_month = metrics['month'].idxmax()
        summary.append(f"- Peak sales in {peak_month} at ${int(metrics['month'].max()):,d} [Plan seasona
    if not metrics['satisfaction'].empty:
        top_sat = metrics['satisfaction'].idxmax()
        summary.append(f"- {top_sat} has {int(metrics['satisfaction'].max())} satisfaction [Enhance this
    if not metrics['gender'].empty:
        gender_data = metrics['gender'].to_dict()
        if isinstance(gender_data, dict):
            gender_dist = ", ".join(f"{k}: ${int(v):,d}" for k, v in gender_data.items())
        else:
            gender_series = metrics['gender']
            gender_dict = {}
            for k, v in gender_series.items():
                gender_dict[k] = gender_dict.get(k, 0) + int(v)
            gender_dist = ", ".join(f"{k}: ${v:,d}" for k, v in gender_dict.items())
        summary.append(f"- Gender sales: {gender_dist} [Target key demographics]")
    if metrics['segmentation_clusters'] is not None and not metrics['segmentation_clusters'].empty:
        overall = metrics['overall_segmentation']
        summary.append(f"- Average customer age across clusters is approximately {overall['Customer_Age'
        # Use segmentation_clusters for Count, which contains the aggregated data
        top_cluster = metrics['segmentation_clusters'].loc[metrics['segmentation_clusters']['Count'].idx
        summary.append(f"- Cluster {int(top_cluster['Cluster'])} (Age ~{int(top_cluster['Customer_Age'])
    return "\n".join(summary) if summary else "No data available"

# LLM and Evaluation
llm = ChatOpenAI(model="gpt-4o-mini", openai_api_key=api_key, temperature=0.1) if api_key else None

st.subheader("AI Summary")
ground_truth = "No data available"  # Default value to avoid NameError
```

```
if llm:
    try:
        inputs = assemble_inputs()  # Call without invoke for initial setup
        summary = (RunnableLambda(assemble_inputs) | prompt_template | llm | StrOutputParser()).invoke({
        st.text(summary)
        st.session_state.summary_history.append({"filters": {"Region": selected_region, "Product": selec
        if len(st.session_state.summary_history) > 5:
            st.session_state.summary_history.pop(0)
        ground_truth = generate_ground_truth()  # Update ground_truth if successful
        with st.expander("Debug"):
            st.text(f"Ground Truth: {ground_truth}")
    except Exception as e:
        st.error(f"Error generating summary or ground truth: {str(e)}")
        with st.expander("Debug"):
            st.text(f"Ground Truth (default): {ground_truth}")
else:
    st.info("Set OPENAI_API_KEY to enable AI summaries.")

st.subheader("LangChain Evaluation")
if api_key and 'summary' in locals() and ground_truth and ground_truth != "No data available":
    try:
        eval_llm = ChatOpenAI(model="gpt-4o-mini", openai_api_key=api_key, temperature=0)
        eval_result = load_evaluator("labeled_criteria", criteria="correctness", llm=eval_llm).evaluate_
            prediction=summary, reference=ground_truth, input=prompt_template.format(**inputs))
        st.write(f"- Score: {eval_result.get('score', 'N/A')} - Reasoning: {eval_result.get('reasoning',
    except Exception as e:
        st.error(f"Evaluation error: {str(e)}")
else:
    st.info("Generate a summary to enable evaluation or resolve errors to proceed.")

# st.subheader("Ragas Evaluation")
# if not RAGAS_AVAILABLE:
#     st.warning("Install ragas: pip install ragas datasets")
# elif api_key and 'summary' in locals() and ground_truth and ground_truth != "No data available":
#     try:
#         # Get raw context documents as a list
#         query = f"Summarize for Region={selected_region}, Product={selected_product}"
#         contexts = retriever.invoke(query) if retriever else []
#         retrieved_contexts = [doc.page_content for doc in contexts] if contexts else []

#         eval_data = {
#             "question": [query],
#             "answer": [summary],
#             "contexts": retrieved_contexts,  # Use list of context strings
#             "ground_truth": [ground_truth]
#         }
#         result = evaluate(HFDataset.from_dict(eval_data), [faithfulness, answer_relevancy, context_pre
#         result_df = result.to_pandas()
#         col1, col2 = st.columns(2)
#         with col1:
#             st.metric("Faithfulness", f"{result_df['faithfulness'].iloc[0]:.3f}")
#             st.metric("Relevancy", f"{result_df['answer_relevancy'].iloc[0]:.3f}")
#         with col2:
#             st.metric("Precision", f"{result_df['context_precision'].iloc[0]:.3f}")
#             st.metric("Recall", f"{result_df['context_recall'].iloc[0]:.3f}")
#     except Exception as e:
#         st.error(f"Ragas error: {e}")
# else:
#     st.info("Generate a summary with RAG to see evaluation.")

st.subheader("Filter History")
for i, filters in enumerate(st.session_state.filter_history[:5]):
    st.write(f"Selection {i+1}: {filters}")

### REQUIRES USING VENV - Run from terminal before running app.py
# cd "X:\AI class\07 Capstone Project\insight_forge"
#  . .\.venv\Scripts\Activate.ps1
# streamlit run app.py

#compile to check syntax
#python -m py_compile "x:\AI class\07 Capstone Project\insight_forge\app.py"
```

bi/metrics.py - Full Source

```
# this is already install in the venv - pip install pandas scikit-learn seaborn matplotlib, added to req

import pandas as pd
from sklearn.cluster import KMeans

def sales_by_month(df):
    """Calculate total sales by month"""
    # Your data uses 'Date' column
    date_col = 'Date' if 'Date' in df.columns else 'Month'

    if date_col not in df.columns or 'Sales' not in df.columns:
        return pd.DataFrame()

    df_copy = df.copy()

    # Parse dates with explicit format to avoid warning
    try:
        # Try ISO format (YYYY-MM-DD) which matches your data
        df_copy[date_col] = pd.to_datetime(df_copy[date_col], format='%Y-%m-%d', errors='coerce')
        if df_copy[date_col].isna().all():
            # Fallback to flexible parsing
            df_copy[date_col] = pd.to_datetime(df_copy[date_col], errors='coerce')
    except Exception:
        df_copy[date_col] = pd.to_datetime(df_copy[date_col], errors='coerce')

    df_copy = df_copy.dropna(subset=[date_col])

    if df_copy.empty:
        return pd.DataFrame()

    # Group by month
    result = df_copy.groupby(df_copy[date_col].dt.to_period('M'))['Sales'].sum()
    result.index = result.index.astype(str)

    return result.round(0)


def sales_by_region(df):
    """Calculate total sales by region"""
    if 'Region' not in df.columns or 'Sales' not in df.columns:
        return pd.DataFrame()
    return df.groupby('Region')['Sales'].sum()
    return result.round(0)


def satisfaction_by_region(df):
    """Calculate average satisfaction by region"""
    # Your data uses 'Customer_Satisfaction' column
    satisfaction_col = 'Customer_Satisfaction' if 'Customer_Satisfaction' in df.columns else 'Satisfacti

    if satisfaction_col not in df.columns or 'Region' not in df.columns:
        return pd.DataFrame()

    return df.groupby('Region')[satisfaction_col].mean()
    #  return result.round(0)

def sales_by_gender(df):
    """Calculate total sales by gender"""
    # Your data uses 'Customer_Gender' column
    gender_col = 'Customer_Gender' if 'Customer_Gender' in df.columns else 'Gender'

    if gender_col not in df.columns or 'Sales' not in df.columns:
        return pd.DataFrame()

    return df.groupby(gender_col)['Sales'].sum()


def sales_trend_over_time(df):
    """Calculate sales trends over time with proper date handling"""
    # Your data uses 'Date' column
    date_col = 'Date' if 'Date' in df.columns else 'Month'

    if date_col not in df.columns or 'Sales' not in df.columns:
        return pd.DataFrame()

    df_copy = df.copy()

    try:
        # Try ISO format (YYYY-MM-DD) which matches your data
        df_copy[date_col] = pd.to_datetime(df_copy[date_col], format='%Y-%m-%d', errors='coerce')
        if df_copy[date_col].isna().all():
            df_copy[date_col] = pd.to_datetime(df_copy[date_col], errors='coerce')
    except Exception:
        df_copy[date_col] = pd.to_datetime(df_copy[date_col], errors='coerce')

    df_copy = df_copy.dropna(subset=[date_col])

    if df_copy.empty:
        return pd.DataFrame()

    df_copy = df_copy.sort_values(date_col)
    result = df_copy.groupby(df_copy[date_col].dt.to_period('M'))['Sales'].sum()
    result.index = result.index.astype(str)

    return result

def product_performance_comparison(df):
    """
    Compute product performance comparisons based on sales and customer satisfaction.
    Returns a DataFrame with products as index and aggregated metrics.

    Args:
        df (pd.DataFrame): Input DataFrame containing sales data with 'Product' and 'Sales' columns.
                          Optional: 'Customer_Satisfaction' for additional insights.

    Returns:
        pd.DataFrame: Aggregated metrics by product (e.g., total sales, average satisfaction).
    """
    if 'Product' not in df.columns or 'Sales' not in df.columns or df.empty:
        return pd.DataFrame()

    # Group by product and calculate total sales and average satisfaction
    product_metrics = df.groupby('Product').agg({
        'Sales': 'sum',
        'Customer_Satisfaction': 'mean' if 'Customer_Satisfaction' in df.columns else 'count'
    }).rename(columns={'Customer_Satisfaction': 'Avg_Satisfaction' if 'Customer_Satisfaction' in df.colu

    return product_metrics if not product_metrics.empty else pd.DataFrame()

def customer_demographics_summary(df):
    """
    Compute summary statistics for customer demographics, grouped by gender and region.
    Returns a DataFrame with aggregated metrics (mean, median, std) for sales, age, and satisfaction.
    """
    if 'Customer_Gender' not in df.columns or 'Region' not in df.columns or df.empty:
        return pd.DataFrame()

    return df.groupby(['Customer_Gender', 'Region']).agg({
        'Sales': ['mean', 'median', 'std'],
        'Customer_Age': ['mean', 'median', 'std'],
        'Customer_Satisfaction': ['mean', 'median', 'std']
    }).reset_index()

    return result.round(0)


def customer_age_group_summary(df):
    """
    Summarize metrics by age groups and add AgeGroup column to the DataFrame.
    Returns a DataFrame with mean sales and satisfaction per age bin, and modifies df in place.
    """
    if 'Customer_Age' not in df.columns or df.empty:
        return pd.DataFrame()

    df_copy = df.copy()
    df_copy['AgeGroup'] = pd.cut(df_copy['Customer_Age'], bins=[18, 30, 40, 50, 60, 70], labels=['18-29'

    result = df_copy.groupby('AgeGroup', observed=True).agg({
        'Sales': 'mean',
        'Customer_Satisfaction': 'mean'
    }).reset_index()

    # Update the original df with AgeGroup (in place modification)
    df['AgeGroup'] = df_copy['AgeGroup']

    return result.round(0)

# Customer demographics and segmentation
def customer_segmentation_clusters(df, n_clusters=4):
    """
    Perform K-Means clustering for customer segmentation based on age, sales, and satisfaction.
    Returns a tuple: (clustered DataFrame, DataFrame of cluster summaries, dict of overall averages).
    Modifies df in-place with 'Cluster' column.
    """
    # Check for required columns and non-empty DataFrame
    required_cols = {'Customer_Age', 'Sales', 'Customer_Satisfaction'}
    if not required_cols.issubset(df.columns) or df.empty:
        return df.copy(), pd.DataFrame(), {}

    # Prepare data for clustering
    X = df[['Customer_Age', 'Sales', 'Customer_Satisfaction']].dropna()
    if X.empty:
        return df.copy(), pd.DataFrame(), {}

    # Perform clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    df.loc[X.index, 'Cluster'] = kmeans.fit_predict(X)  # Add Cluster only to rows with data

    # Generate cluster summaries
    result = df.groupby('Cluster', observed=True).agg({
        'Customer_Age': 'mean',
        'Sales': 'mean',
        'Customer_Satisfaction': 'mean',
        'Customer_Gender': 'count'
    }).reset_index().rename(columns={'Customer_Gender': 'Count'})

    # Calculate overall averages
    overall = {
        'Customer_Age': int(df['Customer_Age'].mean()) if not df['Customer_Age'].isna().all() else 0,
        'Sales': int(df['Sales'].mean()) if not df['Sales'].isna().all() else 0,
        'Customer_Satisfaction': int(df['Customer_Satisfaction'].mean()) if not df['Customer_Satisfactio
        'Count': len(df)
    }

    return df, result, overall
```