AI-Powered Intelligent Smartphone Recommendation Bot

2025

AI-Powered Smartphone Recommendation Bot using Azure AI S

Project Overview

This project presents the design and development of an intelligent, conversational recommendation bot that assists users in finding smartphones tailored to their technical preferences. Built on **Microsoft Azure** and powered by a **K-Nearest Neighbors (KNN)** machine learning model, the bot processes user inputs across 14 product features—including RAM, processor speed, camera quality, battery capacity, and price—to deliver personalized, data-driven recommendations. The solution integrates **Azure Bot Service** for conversational orchestration, **Flask** for API management, and **Telegram** as the communication channel, demonstrating a complete end-to-end AI application.

Business Problem

The smartphone market is saturated with hundreds of models launching annually, each with multifaceted specifications. Consumers face significant decision fatigue when manually comparing technical attributes across brands, often struggling to balance performance, budget, and personal preferences. For e-retailers, this translates to lost sales opportunities and higher return rates. An AI-powered recommendation system addresses this by automatically surfacing "phones like this" in real-time, balancing technical similarity with user preferences, thereby streamlining the purchase journey, increasing conversion rates, and enhancing customer satisfaction.

Solution Approach

Data Acquisition & Preprocessing: A web scraping pipeline was developed to extract structured smartphone data from Flipkart, including specifications, ratings, and pricing. The raw data underwent extensive preprocessing—handling missing values, splitting the corpus column into meaningful features, converting data types, and scaling numeric attributes using **StandardScaler** to ensure uniformity for model training.

Recommendation Engine: A **KNN model** was trained on the processed feature set using Euclidean distance to identify the most similar phones based on user-defined preferences. The system incorporates a price-filtering mechanism (±20% of the user's budget) to ensure recommendations are both feature-aligned and budget-appropriate.

Conversational Interface: A 14-step stateful conversation flow was implemented using the **Azure Bot Framework**, guiding users through each feature input with validation and error handling. The bot maintains session states and triggers the recommendation engine once all inputs are collected.

Deployment & Integration: The bot was deployed as a **Flask application** on **Azure App Service** and integrated with **Telegram** via the Bot Framework, enabling real-time, multi-channel accessibility.

CHIRAG ARUNKUMAR VYAS

Technology Stack

- **Microsoft Azure Bot Service** – Conversation orchestration and channel integration.
- **Azure App Service** – Hosting the Flask backend.
- **Azure Cognitive Services** – Multimodal interaction capabilities.
- **Python** – Core development language.
- **Scikit-learn** – KNN model training and feature scaling.
- **Flask** – REST API for bot message handling.
- **Pandas & NumPy** – Data manipulation and preprocessing.
- **Telegram API** – Communication channel for end-user interaction.

Key Achievements & Testing

The bot was rigorously tested across functional, performance, and error-handling scenarios. Results demonstrated accurate conversation flow, valid recommendation outputs, and robust handling of invalid inputs. Performance testing under concurrent loads validated the system's scalability, with recommendations generated within sub-second latency for single-user scenarios. Price-filtering logic ensured budget-aligned suggestions, and the integration with Telegram confirmed seamless multi-channel operability.

Business Applications

This solution framework is highly adaptable across industries requiring personalized product discovery, including e-commerce (for electronics, fashion, and home goods), travel (for itinerary planning), education (for course recommendations), and healthcare (for wellness suggestions). The modular architecture enables rapid customization for diverse product catalogs and user preference models.

Future Enhancements

Planned improvements include implementing **Cosmos DB** for persistent session storage, deploying the model on **Azure ML** for optimized inference, integrating **adaptive cards** for richer user experiences, and incorporating sentiment analysis to refine recommendations based on user feedback.

Table of content

• Choose a real-world application for your recommendation bot.

Problem Statement:

•**Information Overload**: Hundreds of smartphone models launch yearly, each with multifaceted specs (RAM, CPU, camera, battery, display), prices, and user ratings.

•**User Pain**: Manually comparing these dimensions is slow and error-prone, especially for non-expert buyers.

•**Core Challenge**: Automatically surface "phones like this" in real-time, balancing technical similarity, user preference (ratings), and budget (price).

1. Context & Motivation

- **Smartphone Market Explosion**
  - Over 1,000 new smartphone models released worldwide each year.
  - Vast spec-sheet variations: CPU, RAM, storage, camera modules, battery sizes, display technologies, network bands, OS versions.
- **User Decision Pain-Points**
  - Hard to compare dozens of technical attributes across brands.
  - Difficulty balancing performance vs. budget vs. personal preferences (gaming, photography, battery life).
  - Time-consuming manual research on e-commerce sites and tech reviews.

---

2. Business Value

- **For Consumers**:
  - Instant "phones like this" suggestions tailored to their chosen model.
  - Confidence in purchases through transparent, spec- and rating-based reasoning.
- **For E-Retailers & Marketplaces**:
  - Increased cross-sell and upsell: recommending slightly higher-end or budget-friendly alternatives.
  - Higher conversion rates and average order values.
  - Reduced return rates by matching user needs more accurately.

---

3. Key Features of the Mobile Recommender Bot

Feature	Description
Content-Based Matching	Finds phones with similar technical "corpus" (RAM, CPU, camera, battery).
Hybrid Ranking	Blends spec-similarity with user ratings and price proximity.
Lightweight & Fast	TF-IDF + cosine similarity → sub-second recommendations on web or mobile.
Explainable Rules	"Why this phone?" explanations via top spec matches (e.g. same chipset).
Image & Price Display	Shows phone image, price, and rating alongside recommendations.

---

4. User Interaction Flow

1. **User selects** a phone model they like (or enters its name).
2. Bot **computes similarity** against all models in the catalogue (using precomputed TF-IDF vectors).
3. Bot **ranks** candidates by hybrid score (specs + ratings + price).
4. Bot **returns** top-N suggestions with images, prices, ratings, and "key matching specs" highlights.
5. User can **refine** by adjusting "budget slider" or "battery priority" toggle.

2. Dataset Overview

Sample Data (Cleaned)

Name	Rating	Price (₹)	Image URL	Corpus (Key Specs)
REDMI Note 12 Pro 5G (Onyx Black)	4.2	23,999	[URL]	**Storage:** 128GB, **RAM:** 6GB, **Processor:** Mediatek Dimensity 1080, **Camera:** 50MP+8MP+2MP, **Battery:** 5000mAh, **Display:** 6.67" AMOLED (120Hz)
OPPO F11 Pro (Aurora Green)	4.5	20,999	[URL]	**Storage:** 128GB, **RAM:** 6GB, **Processor:** Helio P70, **Camera:** 48MP+5MP, **Battery:** 4000mAh, **Display:** 6.5" TFT-LTPS
REDMI Note 11 (Starburst White)	4.2	13,149	[URL]	**Storage:** 64GB, **RAM:** 4GB, **Processor:** Snapdragon 680, **Camera:** 50MP, **Battery:** 5000mAh, **Display:** 6.43" AMOLED
OnePlus Nord CE 5G (Blue Void)	4.1	21,999	[URL]	**Storage:** 256GB, **RAM:** 12GB, **Processor:** Snapdragon 750G, **Camera:** 64MP, **Battery:** 4500mAh, **Display:** 6.43" Fluid AMOLED

Key Features for Recommendation

1. **Corpus (Primary) → TF-IDF for technical similarity**.
2. **Rating → Boost highly-rated phones**.
3. **Price → Filter/weight budget-friendly options**.

- **problem statement and how AI-based recommendations will add value.**

The **problem statement** revolves around creating a **Content-Based Recommendation System** for a product dataset, particularly for mobile phones, based on various features such as RAM, price, processor speed, camera specifications, battery capacity, and more. The goal is to develop a **recommendation bot** that can suggest relevant mobile phones to users based on the features they value, without needing any user reviews or ratings data.

Key Challenges:

- The dataset does not contain user reviews, so recommendations must be purely based on **product features** like RAM, processor, camera megapixels, etc.

- The recommendation system needs to provide users with suggestions based on **similarity** in product attributes, helping them discover products they may not have initially considered.
- The system should be **scalable**, handling a large variety of mobile phones and their diverse features.

How AI-based Recommendations Will Add Value

1. Personalized User Experience

AI-powered content-based recommendation systems can deliver personalized suggestions to users by analyzing the specific attributes they are interested in. Instead of relying on general popularity or ratings, the system will suggest mobile phones based on **matching features** with the user's preferences.

For example, if a user prefers a phone with at least 6GB of RAM, a specific processor speed, or a camera with a certain megapixel count, the system can filter and recommend products that meet these requirements. This allows users to focus on features that matter to them, enhancing their shopping experience.

2. Product Discovery

Many users may not be aware of all available products that meet their preferences. A recommendation system introduces users to mobile phones that they might not have considered otherwise. AI models can highlight options that fit users' **specific needs**, promoting variety and increasing the likelihood of finding the best match.

3. Time-saving

Rather than manually sifting through thousands of options, users can rely on the bot to automatically filter through the available products based on their needs. This reduces decision fatigue and improves the overall efficiency of the purchasing process.

4. Real-time Updates

As new phones are released or existing ones are updated, the AI system can quickly adapt by incorporating new product data. This ensures that recommendations are always up-to-date and relevant.

Target Users and Expected Interactions

1. Target Users

- **Tech Enthusiasts and Early Adopters**: Users who are keen on staying updated with the latest mobile technologies and look for specific features such as camera quality, RAM, or processor speed.
- **Budget-Conscious Shoppers**: Users who are looking for the best deals and want to find a phone that fits their budget, but also meets their specific requirements (e.g., price under a certain threshold).
- **Casual Users**: Users who might not be very tech-savvy and need a simplified method to select a phone based on high-level specifications like RAM, camera quality, battery life, and price.
- **Online Shoppers**: Users engaging in online shopping who seek personalized suggestions without browsing through a wide range of products.

2. Expected Interactions The expected interactions between the target users and the AI-based recommendation bot are:

- **Initial Query**: Users will interact with the bot by asking for recommendations based on product attributes. For example, "Show me phones with at least 6GB of RAM and a battery of 5000 mAH."
- **Filter Criteria**: The user may specify specific preferences (e.g., price range, camera quality, brand preference). The bot will use these inputs to narrow down the product list and present options accordingly.
- **Product Comparison**: Users may want to compare several options, asking the bot to provide comparisons based on key specifications. The bot should be able to list the differences, helping the user make a more informed decision.
- **Continuous Feedback Loop**: Users can refine their requests based on initial recommendations. For example, if a user initially asks for "phones with 6GB of RAM", they may follow up with "recommend phones with 8GB of RAM under 25000 rupees."
- **Explaining Features**: If a user is not familiar with specific product attributes (e.g., processor speed, camera megapixels), the bot can provide explanations about these features, helping users understand their significance.

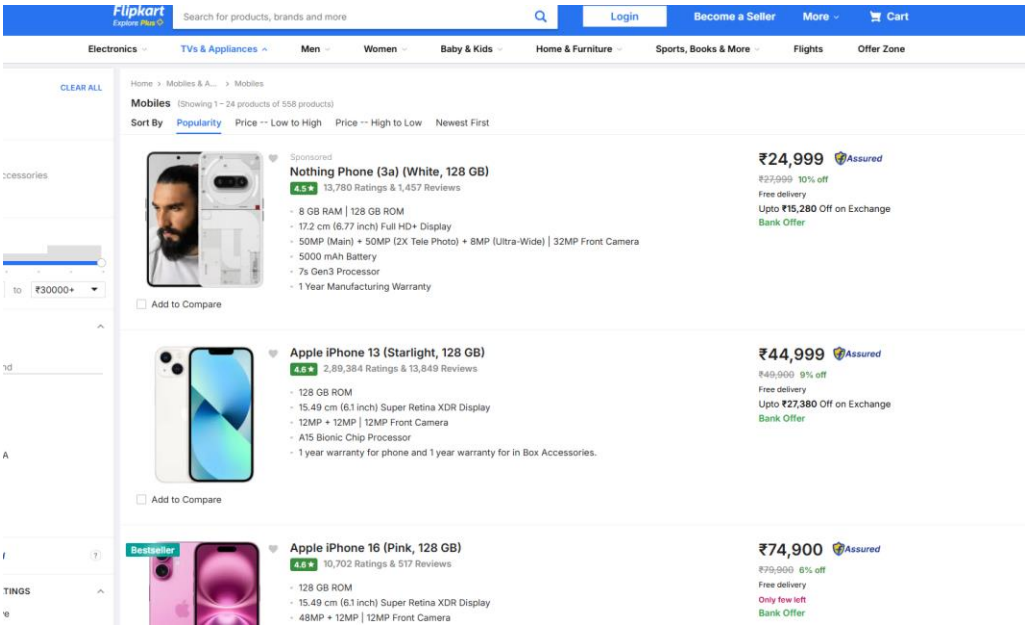
3. User Expectations

- **Relevant and Personalized Suggestions**: Users expect the bot to recommend phones that are highly relevant to their needs. A content-based approach based on features ensures this.
- **Speed and Efficiency**: Users want quick responses with minimal input. The bot should be able to filter and suggest options almost instantly.
- **Clarity and Simplicity**: The interface should be user-friendly, and the interaction should be simple and intuitive, particularly for non-tech-savvy users.
- **Customization**: Users should be able to fine-tune recommendations based on detailed specifications and preferences.

• Data Acquisition Pipeline from Flipkart *(web scraping)*

Introduction to Web Scraping

What is Web Scraping?



Web scraping is the automated process of extracting data from websites. It involves fetching web pages, parsing their content (HTML/XML), and collecting structured information for analysis, storage, or further processing.

Why is it Used?

1. **Data Collection** – Gather product details (e.g., prices, specs), news articles, or social media trends.
2. **Competitive Analysis** – Monitor pricing, features, and availability across e-commerce sites.
3. **Research & AI Training** – Build datasets for machine learning models (e.g., recommendation systems).
4. **Real-Time Monitoring** – Track stock prices, weather updates, or job listings.

How Does Web Scraping Work?

1. **Send HTTP Requests** – A scraper requests a webpage (like Flipkart's smartphone listings).
2. **Parse HTML** – Tools like Beautiful Soup (Python) extract structured data (product names, prices, etc.).
3. **Store Data** – Save results in CSV, JSON, or databases for analysis.

Challenges & Ethics

- **Bot Detection** – Sites like Flipkart may block scrapers using CAPTCHAs or IP bans.
- **Legal Risks** – Some websites prohibit scraping in their Terms of Service (ToS).
- **Data Quality** – Inconsistent formatting (e.g., "₹20,000" vs. "20000") requires cleaning.

Common Tools

- **Python Libraries**: BeautifulSoup, Scrapy, Selenium (for dynamic sites).
- **No-Code Tools**: Octoparse, ParseHub (for beginners).

Conclusion

Web scraping is a powerful technique for extracting web data efficiently, but it must be done responsibly to respect website policies and avoid misuse.

Why Web Scraping on Flipkart for Smartphone Data?

1. Access to Structured Product Data

- Flipkart hosts **detailed smartphone specifications** (RAM, processor, camera, battery, etc.) in a semi-structured format.
- This data is **not readily available via public APIs**, making scraping the most efficient way to collect it.

2. Real-Time & Updated Information

- Prices, ratings, and availability change frequently.
- Scraping ensures **up-to-date data** for accurate recommendations.

3. Competitive Analysis & Market Research

- Compare phones across brands (Xiaomi, Samsung, OnePlus, etc.).
- Identify **trends** (e.g., "Are 5G phones getting cheaper?").

4. Building Custom Datasets for AI/ML Models

- Recommendation systems (like yours) need **large, structured datasets**.
- Flipkart provides **high-quality, categorized data** (images, ratings, specs).

5. No Official API Alternative

- Flipkart's **Seller API is restricted** (requires approval).

- Public datasets (Kaggle) may be **outdated or incomplete**.

Challenges of Scraping Flipkart

1. Anti-Scraping Measures

- **CAPTCHAs, IP blocks, and dynamic content** (JavaScript rendering).
- **Solution**: Use rotating proxies, Selenium, or headless browsers.

2. Data Consistency Issues

- **Varied formatting** (e.g., "6GB RAM" vs. "6 GB RAM").
- **Missing fields** (some products lack ratings or specs).
- **Solution**: Data cleaning pipelines with regex and fallback checks.

3. Legal & Ethical Concerns

- **Flipkart's Terms of Service (ToS)** may prohibit scraping.
- **Best Practice**:
  - Scrape **publicly available data only**.
  - **Limit request rates** (avoid overloading servers).
  - **Use for research, not commercial resale**.

Alternatives to Web Scraping

1. **Flipkart Affiliate API** (Requires approval, limited access).
2. **Third-Party Data Providers** (Paid, e.g., Data Weave, Price API).
3. **Pre-Collected Datasets** (Kaggle, but may lack recent data).

Key Takeaways

Why Scrape?

- Get **real-time, structured smartphone data** for analysis.
- Build **custom datasets** for AI models (e.g., recommendation systems).

Challenges?

- Anti-bot systems, data cleaning, legal risks.

Solutions?

- Use **proxies, Selenium, and ethical scraping practices**.

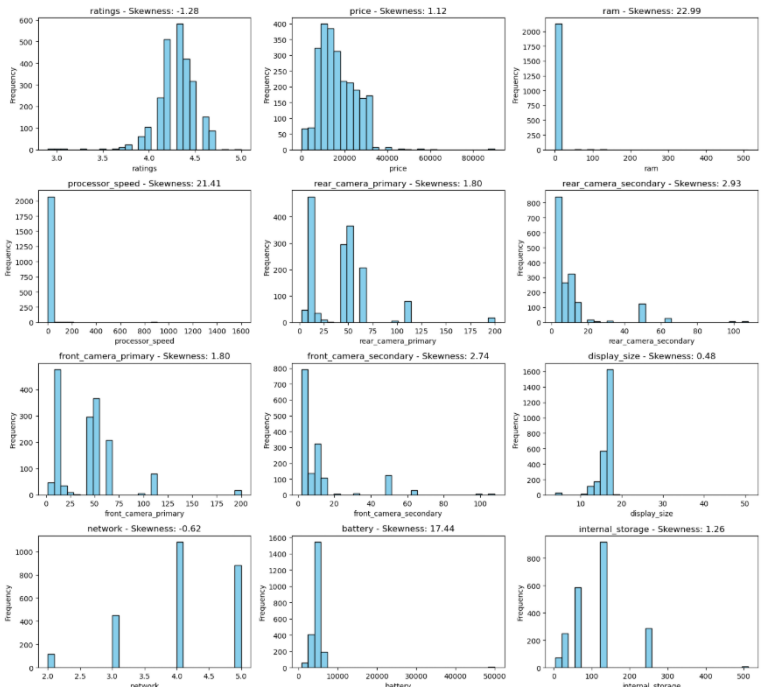
• DATA preprocessing and preparation

1. Handling Missing Values

- We began by checking the dataset for **missing values** in each column using the isnull () function.
- After identifying the columns with missing values, we used appropriate strategies to fill those missing values:
  - For **numerical columns**, we filled missing values with **mean**, **median**, or **mode** based on the distribution and type of the data. The exact method was applied depending on the skewness and nature of the data.
  - For **categorical columns** (like os, processor_type), we used the **mode** (most frequent value) to fill missing entries.
- **Result:** Missing values were handled appropriately across both numerical and categorical columns.

```
[88]:
import pandas as pd
import matplotlib.pyplot as plt

# Assuming the DataFrame is already loaded as df
# Calculate skewness for numeric columns
numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns
skewness_values = df[numeric_columns].skew()

# Display skewness values
print("Skewness Values:")
print(skewness_values)

# Plot histograms for each numeric column in one row of three graphs
fig, axes = plt.subplots(6, 3, figsize=(15, 20))  # Adjust the number of rows and columns based on your data
axes = axes.flatten()  # Flatten axes to easily iterate over them

# Plot histogram for each numeric column
for i, col in enumerate(numeric_columns):
    axes[i].hist(df[col].dropna(), bins=30, color='skyblue', edgecolor='black')  # Drop NA values before plotting
    axes[i].set_title(f'{col} - Skewness: {skewness_values[col]:.2f}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Frequency')

# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```

```
Skewness Values:
ratings                -1.283340
price                   1.123912
ram                    22.993727
processor_speed        21.409509
rear_camera_primary     1.803058
rear_camera_secondary   2.927778
front_camera_primary    1.803058
front_camera_secondary  2.744070
display_size            0.483405
network                -0.615387
battery                17.442931
internal_storage        1.262516
display_width          -0.613568
display_height          1.663414
dtype: float64
```



2. Splitting the corpus

- The corpus column contains text data that provides product-related information (e.g., storage size, RAM, OS version, processor details).
- **Text cleaning** was performed by splitting the corpus text into useful information:
  - We used string operations to extract meaningful features (e.g., RAM, Storage, Processor, etc.).
  - For instance, we split corpus on delimiters like Storage or RAM and extracted corresponding details like the size of RAM, storage, or the type of processor.
  - The text was parsed and processed to extract values like RAM size, Storage, Processor Type, and more.
- **Result:** The corpus column was transformed into separate columns with meaningful values (e.g., ram_in_GB, storage_in_GB, etc.).

Extracting require informatio and perform feature slection

```
import pandas as pd
import re

def extract_features(corpus):
    features = {}

    # Extract Storage
    features['storage'] = re.search(r'Storage(\d+GB)', corpus)
    features['storage'] = features['storage'].group(1) if features['storage'] else None

    # Extract RAM
    features['ram'] = re.search(r'RAM(\d+)', corpus)
    features['ram'] = features['ram'].group(1) + " GB" if features['ram'] else None

    # Extract OS
    features['os'] = re.search(r'System([A-Za-z0-9 ]+)', corpus)
    features['os'] = features['os'].group(1) if features['os'] else None

    # Extract Processor Type
    features['processor_type'] = re.search(r'Processor Type([A-Za-z0-9 ]+)', corpus)
    features['processor_type'] = features['processor_type'].group(1) if features['processor_type'] else None

    # Extract Processor Speed
    features['processor_speed'] = re.search(r'Processor Speed([0-9.]+)', corpus)
    features['processor_speed'] = features['processor_speed'].group(1) if features['processor_speed'] else None

    # Extract Rear Camera Primary
    features['rear_camera_primary'] = re.search(r'(\d+)MP \d+MP \d+MP (\d+)MP', corpus)
    features['rear_camera_primary'] = features['rear_camera_primary'].group(1) if features['rear_camera_primary'] else None

    # Extract Rear Camera Secondary
    features['rear_camera_secondary'] = re.search(r'(\d+)MP \d+MP (\d+)MP', corpus)
    features['rear_camera_secondary'] = features['rear_camera_secondary'].group(2) if features['rear_camera_secondary'] else None

    # Extract Front Camera Primary
    features['front_camera_primary'] = re.search(r'(\d+)MP \d+MP (\d+)MP (\d+)MP', corpus)
    features['front_camera_primary'] = features['front_camera_primary'].group(1) if features['front_camera_primary'] else None

    # Extract Front Camera Secondary
    features['front_camera_secondary'] = re.search(r'(\d+)MP \d+MP (\d+)MP (\d+)MP', corpus)
    features['front_camera_secondary'] = features['front_camera_secondary'].group(2) if features['front_camera_secondary'] else None

    # Extract Display Size
    features['display_size'] = re.search(r'Display Size([\d.]+ cm)', corpus)
    features['display_size'] = features['display_size'].group(1) if features['display_size'] else None

    # Extract Display Resolution
    features['display_resolution'] = re.search(r'Resolution(\d+ x \d+)', corpus)
    features['display_resolution'] = features['display_resolution'].group(1) if features['display_resolution'] else None

    # Extract Display Type
    features['display_type'] = re.search(r'Display Type([A-Za-z0-9 ]+)', corpus)
    features['display_type'] = features['display_type'].group(1) if features['display_type'] else None
```

```
df.head()
```

IRL	corpus	storage	ram	os	processor_type	processor_speed	...	front_camera_primary	front_camera_secondary	display_size	display_resolution
/k...	Storage128 GBRAM6 SystemAndroid 12Processor T...	None	6 GB	Android 12Processor TypeMediatek Dimensity 108...	Mediatek Dimensity 1080Processor Speed2	2.6	...	50	2	16.94 cm	2400 x 1080
/k...	Storage128 GBRAM6 GBExpandable Storage256GB S...	256GB	6 GB	Android Pie 9	MediaTek Helio P70 Octa Core 2	2.1	...	48	48	16.51 cm	2340 x 1080
/k...	Storage64 GBRAM4 SystemAndroid 11Processor Sp...	None	4 GB	Android 11Processor Speed2	None	2.4	...	None	None	16.33 cm	2400 x 1080
/k...	Storage256 GBRAM12 SystemAndroid Q 11Processo...	None	12 GB	Android Q 11Processor TypeQualcomm Snapdragon ...	Qualcomm Snapdragon Octa Core 750G 5G Processo...	2.4	...	None	None	16.33 cm	2400 x 1080
/k...	Storage128 SystemiOS 15Processor TypeA15 Bion...	None	None	iOS 15Processor TypeA15 Bionic Chip 12MP 12MP ...	A15 Bionic Chip 12MP 12MP 12MP 63MP 12MP 12MP ...	None	...	12	12	13.72 cm	2340 x 1080

3. Renaming Columns

renaming dataset

```
# Rename columns
df.rename(columns={
    'ram': 'ram_in_GB',
    'price': 'price_in_rupees',
    'processor_speed': 'processor_speed_in_GHz',
    'rear_camera_primary': 'rear_camera_primary_megapixel',
    'rear_camera_secondary': 'rear_camera_secondary_megapixel',
    'front_camera_primary': 'front_camera_primary_megapixel',
    'front_camera_secondary': 'front_camera_secondary_megapixel',
    'display_size': 'display_size_in_cm',
    'network': 'network_G',
    'battery': 'battery_in_mAH',
    'internal_storage': 'internal_storage_in_GB'
}, inplace=True)
```

```
df.head()
```

	name	ratings	price_in_rupees	imgURL	corpus	ram_in_GB	os	processor_type	processor_speed_in_GHz
0	REDMI Note 12 Pro 5G (Onyx Black, 128 GB)	4.2	23999	https://rukminim2.flixcart.com/image/312/312/x...	Storage128 GBRAM6 SystemAndroid 12Processor T...	6.0	Android 12Processor TypeMediatek Dimensity 108...	Mediatek Dimensity 1080Processor Speed2	2.6
1	OPPO F11 Pro (Aurora Green, 128 GB)	4.5	20999	https://rukminim2.flixcart.com/image/312/312/k...	Storage128 GBRAM6 GBExpandable Storage256GB S...	6.0	Android Pie 9	MediaTek Helio P70 Octa Core 2	2.1
2	REDMI Note 11 (Starburst White, 64 GB)	4.2	13149	https://rukminim2.flixcart.com/image/312/312/x...	Storage64 GBRAM4 SystemAndroid 11Processor Sp...	4.0	Android 11Processor Speed2	Qualcomm Snapdragon 680Processor Speed2	2.4
3	OnePlus Nord CE 5G (Blue Void, 256 GB)	4.1	21999	https://rukminim2.flixcart.com/image/312/312/x...	Storage256 GBRAM12 SystemAndroid Q 11Processo...	12.0	Android Q 11Processor TypeQualcomm Snapdragon ...	Qualcomm Snapdragon Octa Core 750G 5G Processo...	2.4
4	APPLE iPhone 13 mini (Blue, 128 GB)	4.6	3537	https://rukminim2.flixcart.com/image/312/312/k...	Storage128 SystemiOS 15Processor TypeA15 Bion...	6.0	iOS 15Processor TypeA15 Bionic Chip 12MP 12MP ...	A15 Bionic Chip 12MP 12MP 12MP 63MP 12MP 12MP ...	2.2

- We renamed columns to make them more descriptive and aligned with the domain context.
  - For example:
  - ram → ram_in_GB
  - price → price_in_rupees
  - processor_speed → processor_speed_in_GHz
  - rear_camera_primary → rear_camera_primary_megapixel
  - display_size → display_size_in_cm
  - And similar updates for other feature columns.

4. Data Type Conversion

- Some columns, like price, ram, and processor_speed, were initially in string format. We converted them to **numerical formats** (integers or floats) as needed for computational purposes.
- We used **pandas** methods like astype() to ensure that columns were in the appropriate data type:

```
51]: # Remove non-digit characters and convert to integer
     df['internal_storage'] = df['internal_storage'].str.extract(r'(\d+)').astype(float)
```

```
54]: df['ram'] = df['ram'].str.replace('GB', '', regex=False).astype(float)
     # Processor Speed: remove any non-numeric characters (e.g., "GHz") and convert to float
     df['processor_speed'] = df['processor_speed'].str.extract(r'([\d.]+)').astype(float)

     # Rear Camera Primary
     df['rear_camera_primary'] = df['rear_camera_primary'].str.extract(r'(\d+)').astype(float)

     # Rear Camera Secondary
     df['rear_camera_secondary'] = df['rear_camera_secondary'].str.extract(r'(\d+)').astype(float)

     # Front Camera Primary
     df['front_camera_primary'] = df['front_camera_primary'].str.extract(r'(\d+)').astype(float)

     # Front Camera Secondary
     df['front_camera_secondary'] = df['front_camera_secondary'].str.extract(r'(\d+)').astype(float)
```

```
55]: # Display Resolution: extract width and height separately
     resolution_split = df['display_resolution'].str.extract(r'(\d+)\s*[xX×]\s*(\d+)')
     df['display_width'] = resolution_split[0].astype(float)
     df['display_height'] = resolution_split[1].astype(float)
```

```
58]: df = df.drop(columns=['display_resolution'])
     df.head()
```

rear_camera_primary	rear_camera_secondary	front_camera_primary	front_camera_secondary	display_size	network	battery	internal_storage	display_width	display_he
50.0	2.0	50.0	2.0	16.94 cm	5G	5000 mAh	128.0	2400.0	10
48.0	48.0	48.0	48.0	16.51 cm	256G	4000 mAh	128.0	2340.0	10
NaN	NaN	NaN	NaN	16.33 cm	4G	5000 mAh	64.0	2400.0	10
NaN	NaN	NaN	NaN	16.33 cm	750G	4500 mAh	256.0	2400.0	10
12.0	12.0	12.0	12.0	13.72 cm	5G	None	NaN	2340.0	10

---

5. Feature Engineering

- We performed some **feature engineering**:
  - Extracted new columns like ram_in_GB, storage_in_GB, processor_speed_in_GHz from the cleaned and transformed corpus column.
  - Other extracted features included camera megapixels (e.g., rear_camera_primary_megapixel, front_camera_primary_megapixel), and battery capacity in mAH.

- **Result:** New, clean columns were added to the dataset that captured specific product features.

---

6. Removing Duplicates

- We checked for and removed any duplicate rows in the dataset. Duplicates could occur if the data was scraped multiple times or if identical entries existed for the same product.

---

7. Dropping Irrelevant Columns

- We dropped any columns that were not useful for the recommendation system, such as imgURL (URL of the image) , refresh rate, display type etc. or corpus (which we already processed).

```
df = df.drop(columns=['refresh_rate', 'display_type'])
df.head()
```

	name	ratings	price	imgURL	corpus	storage	ram	os	processor_
0	REDMI Note 12 Pro 5G (Onyx Black, 128 GB)	4.2	23999	https://rukminim2.flixcart.com/image/312/312/x...	Storage128 GBRAM6 SystemAndroid 12Processor T...	None	6 GB	Android 12Processor TypeMediatek Dimensity 108...	Medi Dime 1080Proce Spe
1	OPPO F11 Pro (Aurora Green, 128 GB)	4.5	20999	https://rukminim2.flixcart.com/image/312/312/k...	Storage128 GBRAM6 GBExpandable Storage256GB S...	256GB	6 GB	Android Pie 9	MediaTek H P70 Octa
2	REDMI Note 11 (Starburst White, 64 GB)	4.2	13149	https://rukminim2.flixcart.com/image/312/312/x...	Storage64 GBRAM4 SystemAndroid 11Processor Sp...	None	4 GB	Android 11Processor Speed2	N
3	OnePlus Nord CE 5G (Blue Void, 256 GB)	4.1	21999	https://rukminim2.flixcart.com/image/312/312/x...	Storage256 GBRAM12 SystemAndroid Q 11Processo...	None	12 GB	Android Q 11Processor TypeQualcomm Snapdragon ...	Qualc Snapdra Octa Core 7 5G Proce
	APPLE				Storage128			iOS	A15 Bionic

- **Result:** The dataset was cleaned by removing unnecessary columns.

---

8. Final Check for Missing Values and Data Consistency

- After the preprocessing steps, we performed a final check to ensure that no missing values remained in the dataset and that the data was consistent.

```
df.isnull().sum()
```

```
name                    0
ratings                 0
price                   0
imgURL                  0
corpus                  0
ram                     0
os                      0
processor_type          0
processor_speed         0
rear_camera_primary     0
rear_camera_secondary   0
front_camera_primary    0
front_camera_secondary  0
display_size            0
network                 0
battery                 0
internal_storage        0
display_width           0
display_height          0
dtype: int64
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2534 entries, 0 to 2545
Data columns (total 19 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   name                          2534 non-null   object
 1   ratings                       2534 non-null   float64
 2   price_in_rupees               2534 non-null   int64
 3   imgURL                        2534 non-null   object
 4   corpus                        2534 non-null   object
 5   ram_in_GB                     2534 non-null   float64
 6   os                            2534 non-null   object
 7   processor_type                2534 non-null   object
 8   processor_speed_in_GHz        2534 non-null   float64
 9   rear_camera_primary_megapixel 2534 non-null   float64
 10  rear_camera_secondary_megapixel 2534 non-null  float64
 11  front_camera_primary_megapixel 2534 non-null   float64
 12  front_camera_secondary_megapixel 2534 non-null  float64
 13  display_size_in_cm            2534 non-null   float64
 14  network_G                     2534 non-null   float64
 15  battery_in_mAH                2534 non-null   float64
 16  internal_storage_in_GB        2534 non-null   float64
 17  display_width                 2534 non-null   float64
 18  display_height                2534 non-null   float64
dtypes: float64(13), int64(1), object(5)
memory usage: 460.5+ KB
```

Outcome of Initial Preprocessing

After performing these preprocessing steps, the dataset was cleaned, relevant columns were created, and the data was ready for use in the content-based filtering system. The processed data was:

- Free from missing values.
- Contained useful, descriptive features for each product.
- Cleaned from unnecessary text or irrelevant columns.

This reprocessed dataset was now in a suitable format for building the recommendation system based on product content attributes.

Feature Selection by Dropping Irrelevant Columns

Short Notes:

- To simplify the dataset and retain only meaningful features for modelling, unnecessary columns were removed.
- Dropped columns: 'os', 'processor_type', 'imgURL', and 'corpus' because they were either non-numeric, irrelevant for recommendation, or redundant.
- After dropping, a cleaned DataFrame (df_cleaned_1) was created for further processing and model training.

Scaling Features Using StandardScaler and Saving the Scaler

[30]:

	name	ratings	price_in_rupees	ram_in_GB	processor_speed_in_GHz	rear_camera_primary_megapixel	rear_camera_secondary_megapixel	fi
0	REDMI Note 12 Pro 5G (Onyx Black, 128 GB)	-0.448061	0.816435	-0.052064	-0.049155	0.219857	-0.610711	
1	OPPO F11 Pro (Aurora Green, 128 GB)	0.948479	0.467749	-0.052064	-0.059329	0.136680	2.851519	
2	REDMI Note 11 (Starburst White, 64 GB)	-0.448061	-0.444645	-0.202071	-0.053224	0.136680	-0.234382	
3	OnePlus Nord CE 5G (Blue Void, 256 GB)	-0.913575	0.583977	0.397956	-0.053224	0.136680	-0.234382	
4	APPLE iPhone 13 mini (Blue, 128 GB)	1.413992	-1.561834	-0.052064	-0.057294	-1.360501	0.141948	

```
from sklearn.preprocessing import StandardScaler

# Step 1: Separate 'name' column
names = df_cleaned_1['name']
features = df_cleaned_1.drop('name', axis=1)

# Step 2: Scale all numeric features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(features)

# Step 3: Create final DataFrame with name and scaled features
scaled_df = pd.DataFrame(scaled_features, columns=features.columns)
scaled_df.insert(0, 'name', names)

# Step 5: Save the scaler
joblib.dump(scaler, r"E:\2. MDS DEAKIN UNIVERSITY\TASK\Task8\notebook\final_data_for recommendation system\pkl  model\scaler.pkl")  # sav
```

Short Notes:

- **Separated** the 'name' column to preserve it without scaling, since it is a categorical identifier.
- **Applied** StandardScaler from sklearn. preprocessing to **scale all numeric features** to have zero mean and unit variance, improving model performance.
- **Created** a new DataFrame (scaled_df) combining the scaled features along with the original 'name'.
- **Saved** the fitted scaler object using joblib.dump() for future use during model inference to ensure consistent scaling.

• Training the KNN Model and Preparing User Input

[32]:

```
# ------- Step 2: Train the KNN model -------
from sklearn.neighbors import NearestNeighbors

knn = NearestNeighbors(n_neighbors=5, metric='euclidean')
knn.fit(scaled_df.drop('name', axis=1))  # Train only on features, not on name
```

[32]:

```
NearestNeighbors(metric='euclidean')
```

[33]:

```
# Example of user input (in original unscaled form)
user_input = {
    'ratings': 4.5,
    'price_in_rupees': 60000,
    'ram_in_GB': 6,
    'processor_speed_in_GHz': 2.4,
    'rear_camera_primary_megapixel': 50,
    'rear_camera_secondary_megapixel': 2,
    'front_camera_primary_megapixel': 16,
    'front_camera_secondary_megapixel': 2,
    'display_size_in_cm': 17,
    'network_G': 5,
    'battery_in_mAH': 5000,
    'internal_storage_in_GB': 64,
    'display_width': 2400,
    'display_height': 1080
}

user_df = pd.DataFrame([user_input])
```

[34]:

```
# Standardize user input using the same scaler
user_scaled = pd.DataFrame(scaler.transform(user_df), columns=user_df.columns)
```

```
joblib.dump(knn,'trained_model.pkl')
```

[35]:

```
['trained_model.pkl']
```

[36]:

```
knn_loaded = joblib.load('trained_model.pkl')
distances, indices = knn_loaded.kneighbors(user_scaled)
similar_indices = indices[0]

# Get original data for these indices
recommendations = df_cleaned_1.iloc[similar_indices]

# ------- Step 5: Filter by Price Range -------
user_price = user_input['price_in_rupees']
lower_limit = user_price * 0.8
upper_limit = user_price * 1.2

filtered = recommendations[
    (recommendations['price_in_rupees'] >= lower_limit) &
    (recommendations['price_in_rupees'] <= upper_limit)
]

# ------- Step 6: Output Result -------
if not filtered.empty:
    print("📱 Price-Filtered Recommended Phones:")
    print(filtered[['name', 'price_in_rupees']].reset_index(drop=True))
else:
    print("📱 No phones found in similar features and price range.")
```

```
📱 Price-Filtered Recommended Phones:
                                              name  price_in_rupees
0  OPPO Reno10 Pro+ 5G (Silvery Grey, 256 GB)            54999
1         OPPO R17 Pro (Radiant Mist, 128 GB)            49990
```

Finding Similar Phones (Using KNN Model)

- The trained KNN model is used to find the 5 nearest phones based on user input.
- The system retrieves the original phone details corresponding to the nearest neighbors.

Filtering by Price Range

- To ensure better matches, a price filter is applied.
- Only phones within ±20% of the user's given price are selected.
- This ensures recommendations are not only feature-similar but also budget-appropriate.

Final Output: Displaying Recommendations

- If matching phones are found within the price range, their names and prices are shown.
- If no phones match both features and price range, a message is displayed informing the user.

Final Summary:

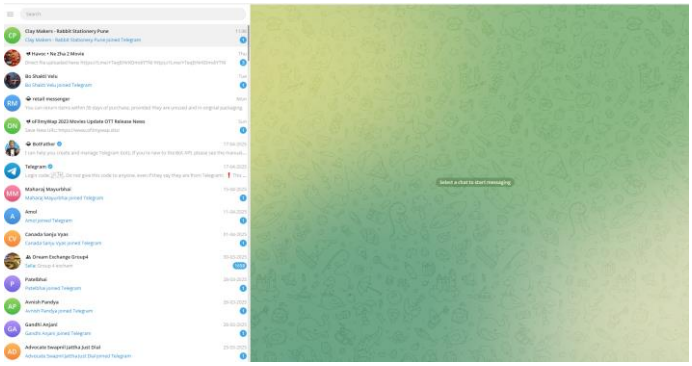
- Identify similar phones based on features.
- Apply a realistic price filter.
- Display the most relevant phone recommendations or a friendly "no match" message
- **Model-Based Filtering**:

  ➔ **train** a machine learning model (KNN in your case) on the features of the mobile phones.

  ➔ The model **learns patterns** in the data and **makes predictions or recommendations** based on that.

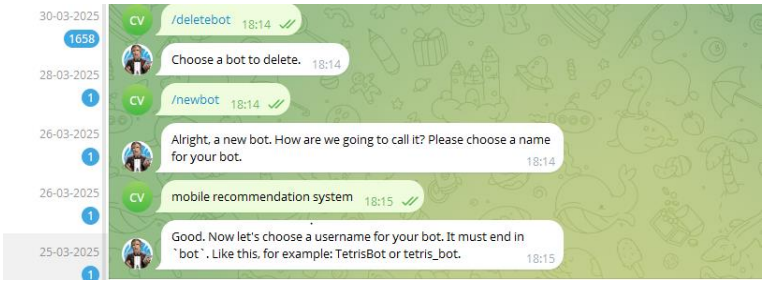
-

- **Telegram chat bot development procedure**



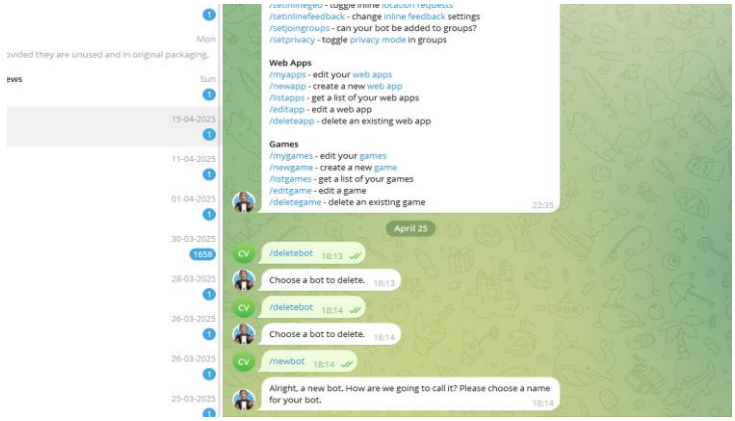
Chatbot Using Telegram - Overview and Procedure

A **chatbot** is a software application designed to simulate human conversation through text or voice interactions. Telegram, a popular messaging app, provides an API that allows developers to create bots to interact with users on the platform. Using a Telegram bot, you can automate customer service, answer frequently asked questions, offer recommendations, and much more.



Why Use Telegram for Chatbot Development?

1. **User Base**: Telegram is widely used across the world, especially in regions like Europe, the Middle East, and parts of Asia.
2. **Ease of Integration**: Telegram provides a simple and well-documented API for bot development, making it easier for developers to integrate AI or automated workflows.
3. **Rich Features**: Telegram supports rich media like images, videos, and inline keyboards, enabling interactive and feature-rich chatbots.
4. **Security**: Telegram offers end-to-end encryption, ensuring secure communication between the bot and users.



Procedure to Create a Telegram Bot

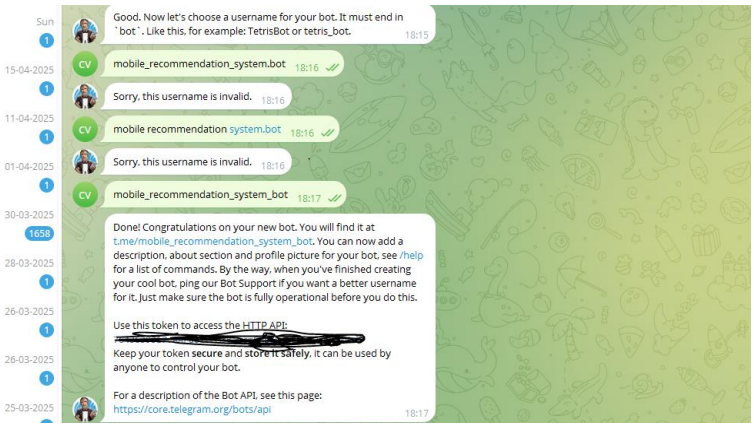
Here is the step-by-step procedure to create a Telegram bot:

Step 1: Register Your Bot with Telegram

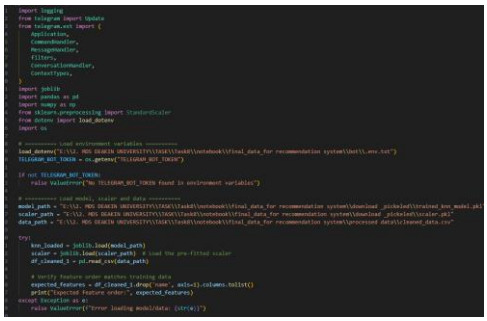
1. **Open Telegram**: Start by opening the Telegram app on your device or accessing the Telegram web app.
2. **Search for the BotFather**:

The **BotFather** is the official Telegram bot that helps you create and manage your bots. In the search bar, type "@BotFather" and select the verified BotFather.

3. **Create a New Bot**:
   - Type /newbot and send the message.
   - You will be asked to **name your bot**. This will be the display name (e.g., "Smart Bot").
   - Next, you will be asked to provide a **username for your bot**. The username must be unique and end with bot (e.g., "smart_bot").
4. **Get Your Bot Token**:
   - Once the bot is created, the BotFather will provide you with a **token**. This token is essential for interacting with the Telegram Bot API. Save this token as you will need it to access your bot programmatically.



- **End-to-End Integration of a KNN Recommendation System with Telegram using Python and VS Code**



1. Imports and Environment Setup

- **Libraries**: Various libraries are used including logging for tracking bot activity, telegram for bot interaction, joblib for loading pre-trained models, pandas for data handling, numpy for numerical operations, and sklearn for scaling features.
- **Environment Variables**: The bot token is loaded using dotenv from an .env file. This keeps sensitive information (like the bot token) secure and external to the code.

2. Model, Scaler, and Data Loading

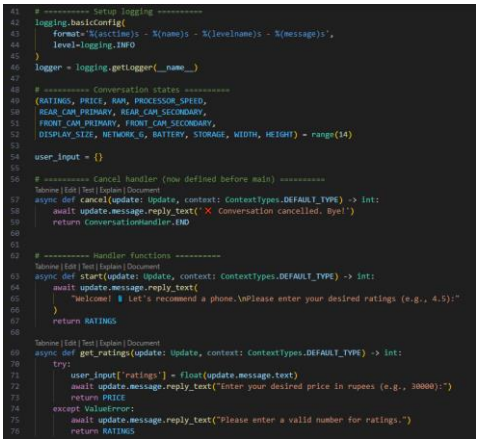
- **Model**: The trained KNN model is loaded using joblib.load(), which will be used to make predictions on user input.
- **Scaler**: A pre-fitted scaler is loaded to normalize user input before passing it to the model.
- **Data**: A cleaned dataset (cleaned_data.csv) is loaded to match user input with the original features used for training.

3. Logging Setup

- Basic logging is set up to track any errors or important actions during the bot's execution. It helps with debugging.

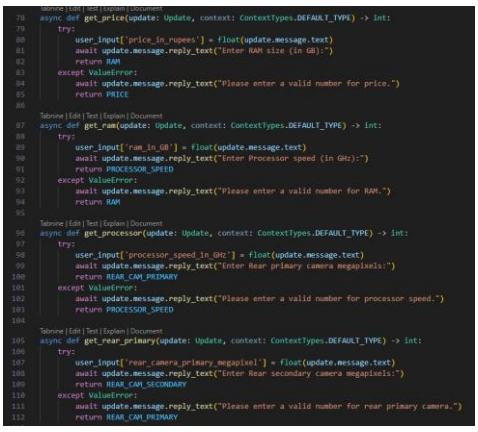
4. Conversation States

- The bot's conversation is structured into multiple states, each corresponding to a user input request. The states cover all the features used in the recommendation, such as **ratings**, **price**, **RAM**, **processor speed**, **camera megapixels**, etc.



5. Handler Functions

- **Handlers** are functions that manage different stages of the conversation. Each function:
  - Requests user input (e.g., ratings, price, RAM).
  - Validates the input.
  - Moves to the next question or, in case of invalid input, prompts the user again.



6. Recommendation Logic

- Once all the necessary features are gathered, the system constructs a feature vector from the user input.
- This input is scaled using the same scaler that was used during training.
- The KNN model is used to find the closest matches from the dataset based on the user's preferences.
- Recommendations are filtered by price, allowing a 20% margin above and below the user's budget.
- The top recommendations are returned to the user, including their names and prices.
- The bot also shows the user's search criteria to confirm the details.

7. Cancel Command

- A cancel function is provided to allow users to exit the conversation at any point with the /cancel command.

8. Conversation Flow

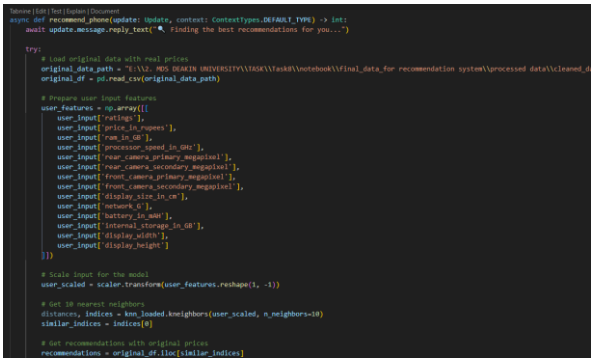
- The ConversationHandler defines the flow of the bot. It specifies:
  - **Entry Point**: The /start command initiates the conversation.
  - **States**: Each state corresponds to a question about a specific phone feature.
  - **Fallbacks**: The /cancel command allows the user to exit the conversation at any time.

9. Bot Execution

- The bot runs continuously, waiting for users to input data, and provides the best recommendations based on the user's preferences.

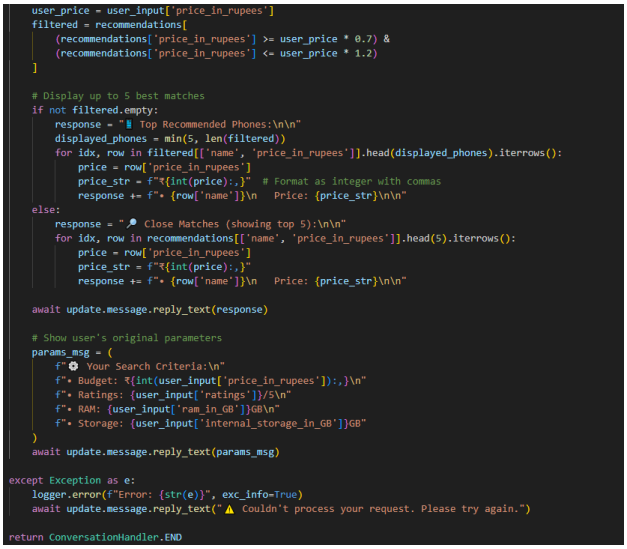
10. Error Handling

- Errors during the recommendation process (e.g., invalid data, missing model) are logged, and the user is informed that something went wrong.



Key Points:

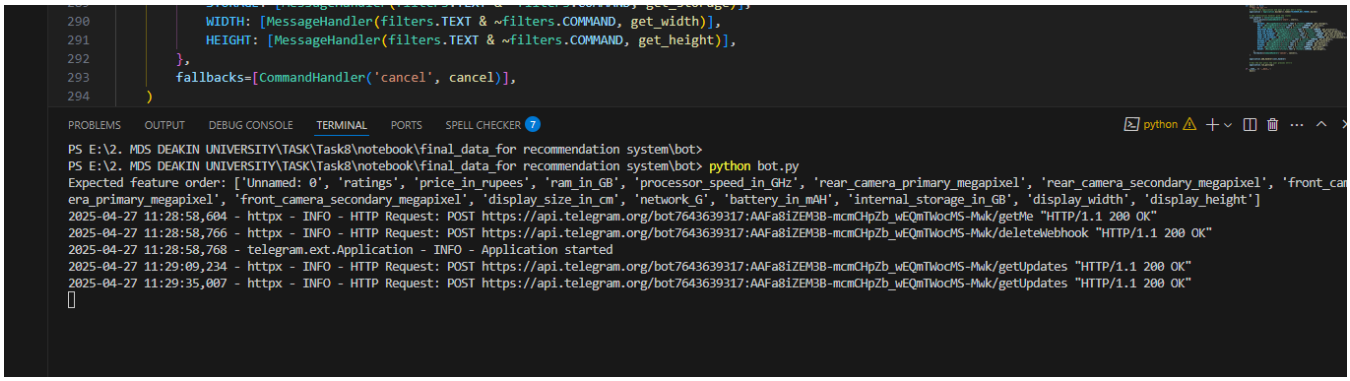
- **Flexibility**: The bot can handle dynamic user input for various phone features, ensuring a personalized recommendation.
- **User-Friendly**: The use of simple prompts and error handling ensures a smooth experience, even if users make mistakes.
- **Data-Driven**: The recommendation logic is based on a trained machine learning model and a well-prepared dataset, ensuring accurate results.
- **Extendable**: Future improvements could include adding more phone features, integrating additional recommendation models, or deploying the bot

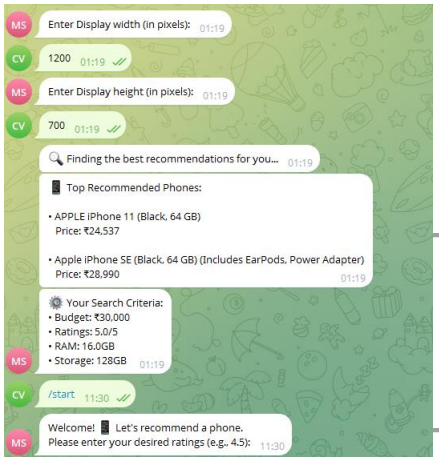


as part of a larger app.

This bot is a robust and scalable solution for offering personalized phone recommendations based on user inputs and a pre-trained machine learning mode

- **Testing and Running python. Py and output observation**





Objective

To create a content-based recommendation system that suggests smartphones based on user preferences, using technical specifications like RAM, camera quality, processor speed, display size, and battery capacity.

Model Used

- **Algorithm**: K-Nearest Neighbors (KNN)
- **Library**: scikit-learn (NearestNeighbors)
- **Distance Metric**: Euclidean Distance

Preprocessing Steps

- **Data Cleaning**:
  - Removed unnecessary columns: 'os', 'processor_type', 'imgURL', 'corpus'.

    - Ensured numeric and categorical features were clean and consistent.
- **Label Encoding**:
    - Initially, categorical fields like 'os' and 'processor_type' were encoded using LabelEncoder.
    - This was later removed due to inconsistencies in user input handling.
- **Feature Scaling**:
    - Applied StandardScaler to normalize all numeric features, improving distance-based comparisons.
    - Ensured column names for scaled features matched the original dataset to avoid sklearn warnings.

Features Used for Recommendation

- **ratings**
- **price_in_rupees**
- **ram_in_GB**
- **processor_speed_in_GHz**
- **rear_camera_primary_megapixel**
- **rear_camera_secondary_megapixel**
- **front_camera_primary_megapixel**
- **front_camera_secondary_megapixel**
- **display_size_in_cm**
- **network_G**
- **battery_in_mAH**
- **internal_storage_in_GB**
- **display_width**
- **display_height**

Recommendation Logic

1. Accept a user-defined phone configuration as input.
2. Standardize the input using the scaler fitted on the training data.
3. Use the KNN model to find the 5 closest phones in feature space.
4. Further filter recommendations by price (within ±20% of the user's budget).

Strengths

- **Fast and easy to implement**.
- **Returns intuitive, similar product suggestions** based on multiple technical features.
- **Non-parametric**: No training required for KNN.

Limitations

- **Performance depends on well-scaled features**.
- KNN struggles with **sparse** or **high-dimensional** data.
- **No handling of cold start problems** or unseen categorical values.
- Sensitive to **outliers** if not preprocessed correctly.

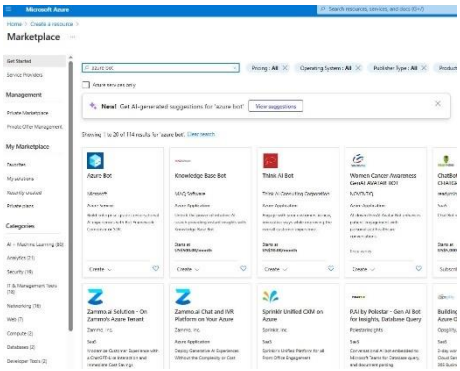
Future Improvements

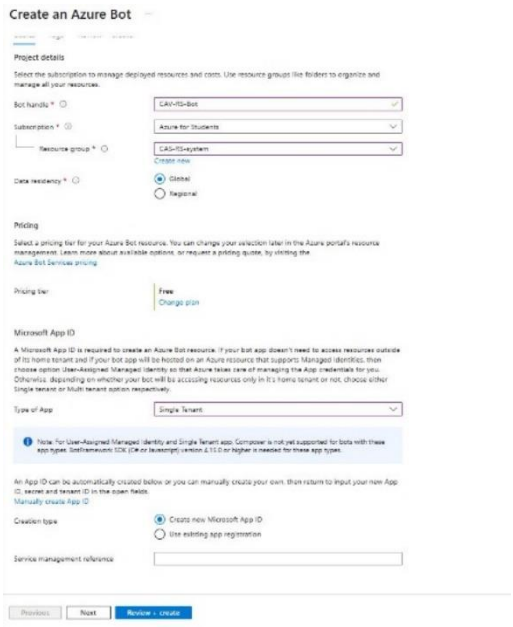
- Implement **PCA** or **t-SNE** for dimensionality reduction and noise filtering.
- Replace KNN with **clustering-based filtering** or **deep learning models** for more nuanced similarity analysis.
- Integrate a **hybrid recommendation system** by combining user reviews, sentiment scores, or collaborative filtering.
- **Deploy as an API** or integrate it into a mobile/web application for wider accessibility.

Conclusion

The KNN-based recommendation system provides a solid starting point for suggesting smartphones based on technical specifications. It's effective for content-based filtering, but can be improved and scaled using advanced techniques for deployment in real-world applications.

- **Setup Microsoft Azure Bot**





1. Prerequisites

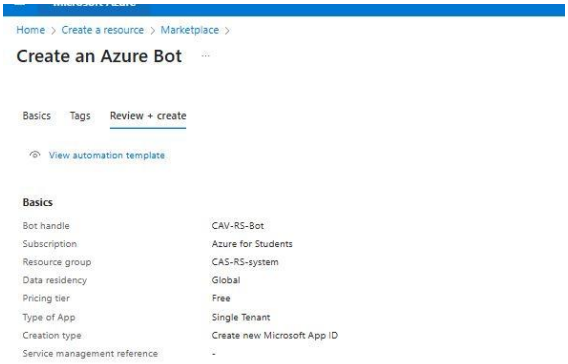
Before setting up an Azure Bot, ensure you have the following:

- **Azure Subscription**: You'll need an Azure subscription to create resources on the Azure platform.
- **Bot Framework SDK**: Install the Bot Framework SDK and related tools for development.
- **Microsoft Azure CLI**: To manage resources from the command line.
- **Visual Studio Code**: For bot development with necessary extensions like Azure Functions, Bot Framework, and Python extensions.

2. Create an Azure Bot Service

Follow these steps to create a bot service:

1. **Sign in to Azure**: Go to the Azure portal and sign in with your Azure account.
2. **Create a Bot Resource**:
   - Navigate to **Create a resource** → **AI + Machine Learning** → **Web App Bot**.
   - Fill in the necessary details:
     - **Bot name**: Choose a unique name for your bot.
     - **Subscription**: Select your Azure subscription.
     - **Resource group**: Create or select an existing resource group.
     - **Location**: Choose the region closest to you.
     - **Pricing tier**: Select a pricing tier (e.g., F0 for free tier during development).
     - **Bot template**: Select a bot template (e.g., Echo Bot, QnA Maker Bot, etc.).
     - **Programming language**: Choose the language you're using (e.g., Python, C#, Node.js).
3. **Create and Deploy**: Once you've filled in the details, click on **Create** to deploy the bot. Azure will automatically create all the necessary resources for the bot (e.g., Azure Bot Service, App Service, and Application Insights for monitoring).

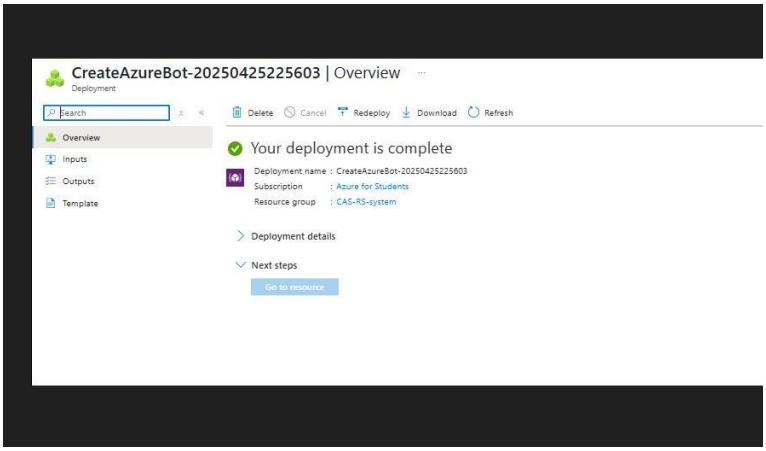


3. Configure Bot Settings

1. **Bot Channels**: After the bot is created, you need to configure it for different channels (e.g., Web Chat, Microsoft Teams, Slack, Facebook Messenger).
   - In the Azure portal, go to your Bot resource.
   - Under the **Channels** section, select the channel you want to configure (e.g., Web Chat).
   - Follow the on-screen instructions to set up the channel.
2. **Bot Framework Registration**:
   - In the **Settings** section of the bot resource, you'll find the **Microsoft App ID** and **Password**. You'll need these credentials when developing your bot to authenticate it with the Azure Bot Service.

4. Develop the Bot

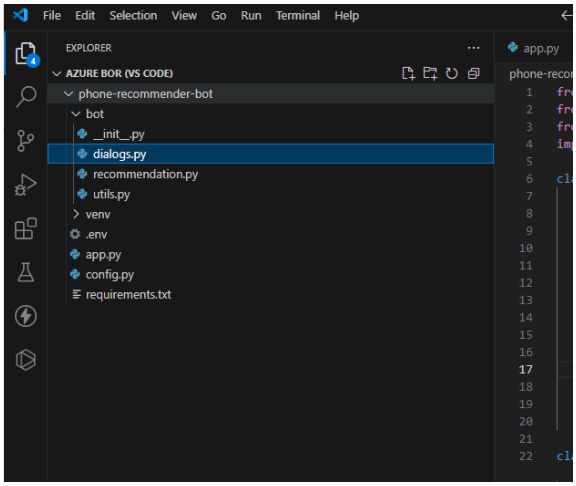
You can use **Bot Framework Composer** or **Bot Framework SDK** to develop your bot.



Set Up This Azure Bot Structure

Here's a step-by-step guide to create this structure from scratch:

1. Create Project Structure



. Create the Project Directory

This is the first step in setting up your project. Create the root directory for your project:

mkdir phone-recommender-bot

cd phone-recommender-bot

2. Create Directory Structure

Now, create the subdirectories to organize your project files:

bash

mkdir -p bot

This will create the bot directory where the logic and other related files for your bot will reside.

3. Create Essential Files

At this point, create the core files for your project:

touch app.py config.py requirements.txt .env

- app.py: This is the main entry point for your application. It will initialize and run your bot.
- config.py: A configuration file for storing settings like API keys, database configurations, etc.
- requirements.txt: This will list all the Python packages required for your project, which can be installed via pip.
- .env: Store environment variables (like API tokens) securely here.

4. Create the Bot Logic Files

Now create the necessary files within the bot directory to implement the logic of your recommendation bot:

touch bot/__init__.py bot/dialogs.py bot/recommendation.py bot/utils.py

- bot/__init__.py: This file is required to make the bot folder a Python package. It's a convention in Python that allows the folder to be recognized as a module that can be imported.
- bot/dialogs.py: Contains the conversation handling logic for your bot. It will define how the bot interacts with users (e.g., collecting user preferences and giving recommendations).
- bot/recommendation.py: This file will hold the logic related to phone recommendations, possibly using a KNN model, algorithms, or other recommendation approaches.
- bot/utils.py: A utility file to handle helper functions and reusable code across your project.

Example Folder Structure:

bash

CopyEdit

```
phone-recommender-bot/
|
├── app.py              # Main entry point for the app
├── config.py           # Configuration settings (e.g., API keys)
├── requirements.txt    # List of required Python packages
├── .env                # Environment variables
|
├── bot/
|   ├── __init__.py     # Marks the folder as a package
|   ├── dialogs.py      # Conversation handlers
|   ├── recommendation.py  # Phone recommendation logic (e.g., KNN model)
|   └── utils.py        # Helper functions
```

5. Install Dependencies

Once you have your requirements.txt file ready, you can install the necessary packages by running:

pip install -r requirements.txt

This sets up all the dependencies for your project (like Flask, scikit-learn, etc.).

Additional Notes:

- **Version Control**: It's good practice to initialize a Git repository in your project directory and add necessary files to .gitignore (like .env, __pycache__, etc.).

git init

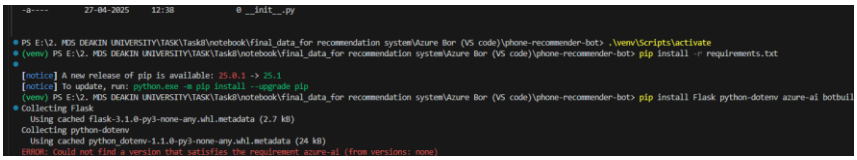
echo ".env" >> .gitignore

echo "__pycache__/" >> .gitignore

- **Environment Variables**: Store sensitive information like API keys, tokens, or other credentials in the .env file, and load them using python-dotenv.

With this structure, you have a clean, modular setup that should scale well for your bot application.

2. Set Up Virtual Environment



1. Create a Virtual Environment

A virtual environment allows you to isolate your project's dependencies from other Python projects. You can create a virtual environment using the following commands:

- **For Windows:**
  - Run the command: python -m venv venv
  - This will create a directory named venv that contains the virtual environment.
- **For Mac/Linux:**
  - Run the command: python3 -m venv venv
  - This will create a directory named venv just like in the Windows command.

2. Activate the Virtual Environment

Once the virtual environment is created, you need to activate it to use it:

- **For Windows:**
  - Run the command: venv\Scripts\activate
- **For Mac/Linux:**
  - Run the command: source venv/bin/activate

When activated, you'll see (venv) at the beginning of your terminal prompt, indicating that you're working inside the virtual environment.

3. Deactivate the Virtual Environment

To deactivate the virtual environment and return to the global Python environment, simply run:

- deactivate

By using a virtual environment, you ensure that the dependencies required by your project do not conflict with other Python projects on your system.

3. Install Required Packages

Install Required Packages:

Once your virtual environment is activated, you can install the necessary packages using pip. For your project, you need to install the following packages:

- **Flask**: A lightweight web framework for building web applications.
- **python-dotenv**: For reading environment variables from .env files.
- **scikit-learn**: A machine learning library for building and training models.
- **numpy**: A package for numerical computing in Python.
- **pandas**: A library for data manipulation and analysis.
- **joblib**: Used for serializing and deserializing Python objects (e.g., models).

To install these packages, run:

bash

CopyEdit

pip install flask python-dotenv scikit-learn numpy pandas joblib

Generate requirements.txt File:

After installing the required packages, it is a good practice to save the list of installed packages along with their versions in a requirements.txt file. This makes it easier for others to recreate the environment by installing the same dependencies.

To generate requirements.txt, run the following command:

bash

CopyEdit

pip freeze > requirements.txt

This will output the list of installed packages along with their version numbers into the requirements.txt file. You can share this file with others to replicate the environment.

4. Prepare Your Model Files

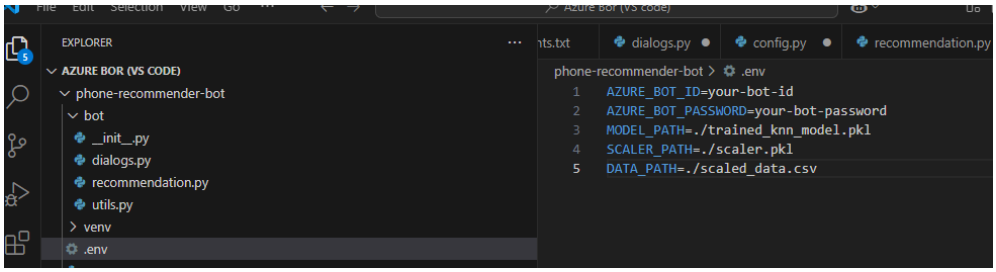
Copy these files from your existing project to the new project root:

- trained_knn_model.pkl (your KNN model)
- scaler.pkl (your scaler)
- scaled_data.csv (your processed data)

5. Set Up Environment Variables

1. Creating the .env File:

You can create a .env file to store your environment variables, which will be loaded into your application using the python-dotenv package. Here's an example of what the .env file might look like:



- **AZURE_BOT_ID**: This is the identifier for your Azure bot.
- **AZURE_BOT_PASSWORD**: This is the password or secret key associated with your Azure bot.
- **MODEL_PATH**: The relative or absolute path to the trained machine learning model (trained_knn_model.pkl).
- **SCALER_PATH**: The path to the scaler object (scaler.pkl) used to standardize input features before prediction.
- **DATA_PATH**: The path to the preprocessed dataset (scaled_data.csv), which can be used for model predictions.

2. Loading Environment Variables in Python:

To load these environment variables into your Python code, use the python-dotenv package. You would typically load the environment variables at the start of your application like this:

3. Using Environment Variables:

Once the environment variables are loaded, you can use them in your application code. For example:

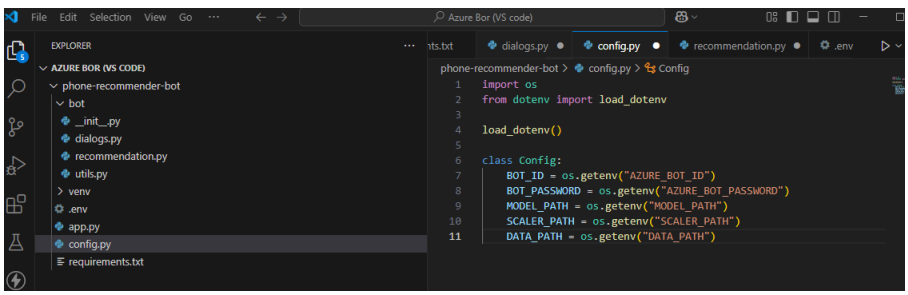
- **For the bot**: You will use AZURE_BOT_ID and AZURE_BOT_PASSWORD for authentication.
- **For loading the model and scaler**: You will use MODEL_PATH and SCALER_PATH to load the model and scaler objects.
- **For data access**: You can use DATA_PATH to load the dataset.

This ensures that sensitive information is not hardcoded into your code and is managed securely.

4. Important Notes:

- **.env File Location**: Ensure that the .env file is placed in the root directory of your project.
- **Security**: Never commit the .env file to version control (e.g., Git). You can add .env to your .gitignore file to keep it out of your Git repository.
- **Deployment**: When deploying your application to cloud platforms like Azure, make sure to configure environment variables directly in the platform's environment settings, as they may not automatically load from the .env file.

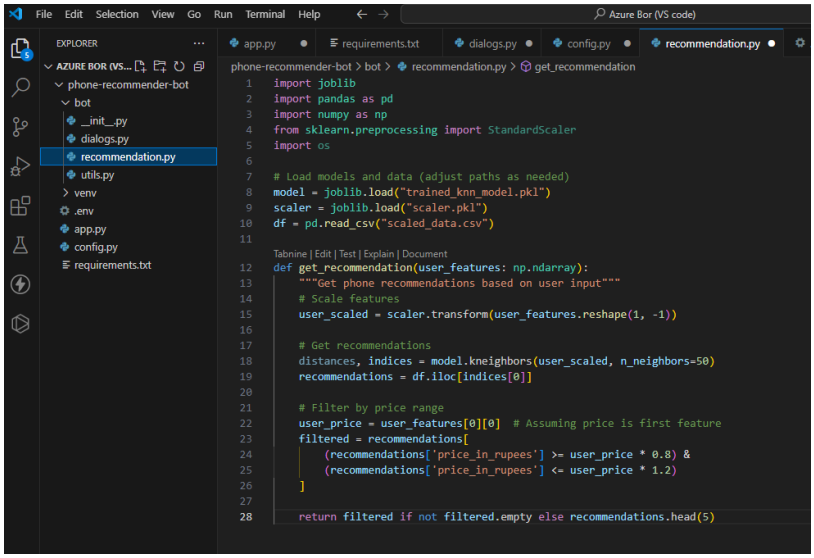
6. Implement Configuration



Configuration Class Using Environment Variables:

1. **Create a Config class** to manage environment variables:
   - The class loads environment variables using python-dotenv.
   - Environment variables are accessed through os.getenv().
2. **Usage**:
   - The Config class holds the bot credentials (BOT_ID, BOT_PASSWORD) and paths to models and data files (MODEL_PATH, SCALER_PATH, DATA_PATH).
3. **Security**:
   - Ensure that the .env file is not committed to version control.
   - Use platform-specific settings for environment variables in production.
4. **Access**:
   - Use Config.BOT_ID, Config.BOT_PASSWORD, etc., to access values throughout the application.

7. Implement Recommendation Logic

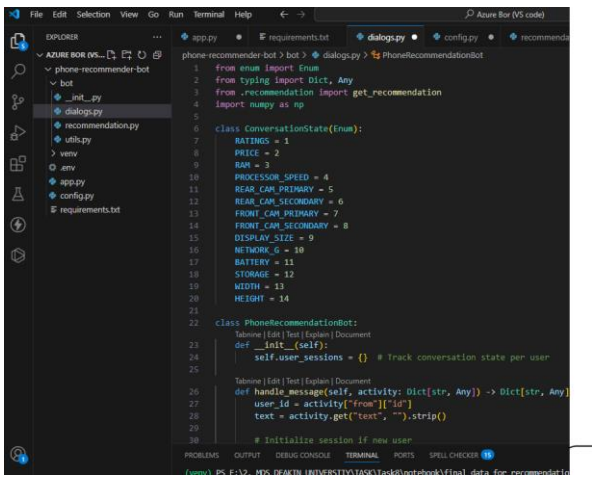


Recommendation Logic:

1. **Model and Data Loading**:
   - The model, scaler, and data are loaded from files specified in the .env configuration file using joblib and pandas.
2. **Recommendation Function** (get_recommendation):
   - Takes user_features as input (user's preferences or attributes).
   - The features are scaled using the pre-trained scaler to match the model's expected input format.
   - The KNN model (model.kneighbors) is used to find the closest neighbors (similar items) based on the scaled features.
3. **Price Filtering**:
   - The recommendations are filtered based on the user's price preference, considering items within a 20% range of the user's selected price.
   - If no items meet the price criteria, the top 5 recommendations are returned by default.
4. **Output**:
   - The recommendations are returned as a list of dictionaries, which can be used for further processing or as the response in the bot.

8. Implement Bot Dialogs

Edit bot/dialogs.py (partial implementation - you'll need to add all your states):

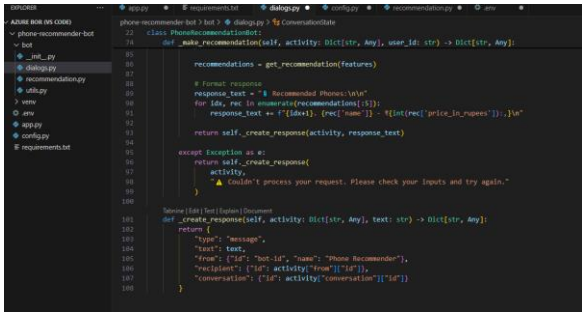


PhoneRecommendationBot:

1. **Enum for Conversation States**:
   - The ConversationState enum defines various states of the conversation. Each state corresponds to a specific user input or question (e.g., RATINGS, PRICE, RAM, etc.).
   - States are given integer values for better clarity in handling transitions.
2. **PhoneRecommendationBot Class**:
   - This class manages user sessions, handles different stages of the conversation, and integrates phone recommendations.
3. **User Sessions**:

- The bot tracks conversations using a user_sessions dictionary. The key is the user_id, and the value is a dictionary that includes the user's current state and the data they have provided so far.

4. **handle_message Method**:
   - Processes incoming messages based on the current conversation state.
   - If a user is new, it starts a conversation. If the user is continuing, it handles the message according to their current state.
   - This method also calls various state handlers like _handle_ratings, _handle_price, etc.
5. **handle_new_user Method**:
   - If a new user starts the conversation, this method initializes their session and triggers the _start_conversation method.
6. **_start_conversation Method**:
   - Initializes a session for a new user and starts them off in the RATINGS state.
   - The bot asks the user to provide a rating for their desired phone.
7. **State Handlers (e.g., _handle_ratings)**:
   - The _handle_ratings method processes the user's ratings input, stores it in the session, and transitions the user to the next state (in this case, PRICE).

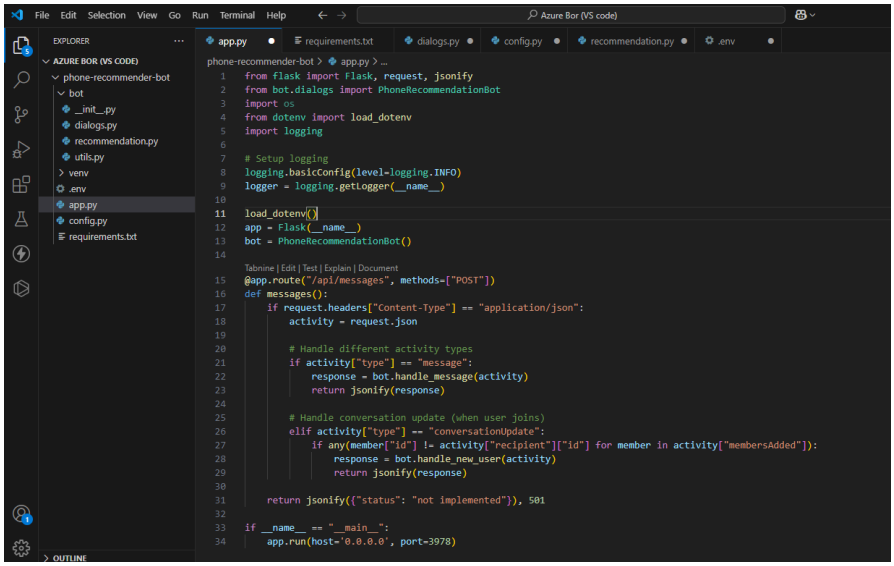


   - Other state handlers should follow a similar structure, processing the user's input and updating the session data while transitioning through states.
8. **_make_recommendation Method**:
   - Once all required user inputs are collected (e.g., ratings, price), this method formats the data into the appropriate feature array and uses the get_recommendation function to fetch phone recommendations.
   - It then formats and returns the top recommendations to the user.
9. **_create_response Method**:
   - This method formats the response in a structured manner (with the bot's ID, name, and conversation details) and returns it as a dictionary to be sent to the user.

Key Flow:

- The conversation starts by asking for ratings, and then it continues to ask for the price, RAM, processor speed, etc., until all required information is collected.
- Once all the information is gathered, the bot makes recommendations based on the user's input using the get_recommendation function.
- Responses are carefully crafted and sent back to the user based on the state of the conversation.

9. Implement Main Application



Flask App Code for Azure Bot Integration:

1. **Flask Setup**:
   - The Flask application is initialized using Flask(__name__). This sets up a simple web server to handle incoming HTTP requests.
2. **Logging**:
   - Logging is set up using Python's logging module. This allows you to track events like incoming requests and responses. The logging level is set to INFO, meaning information-level logs and higher (e.g., warnings, errors) will be shown.
3. **Environment Variables**:
   - dotenv is used to load environment variables from a .env file. This typically contains sensitive data like API keys or bot credentials that shouldn't be hardcoded into the application.

4. **Bot Initialization**:
    - PhoneRecommendationBot is instantiated as bot. This bot is responsible for handling the user's requests and generating phone recommendations based on user inputs.
5. **Flask Route (/api/messages)**:
    - This route listens for incoming POST requests from Azure Bot Framework. The messages() function processes incoming activities and responds accordingly.
    - **Content-Type Check**:
        - The code checks whether the request's content type is application/json, which is expected for incoming bot messages.
    - **Activity Type Handling**:
        - The bot handles different types of activities:
            - **Message Activity**: If the activity type is "message," the bot processes the message using bot.handle_message(activity) and returns the response.
            - **Conversation Update**: If the activity type is "conversationUpdate" (typically when a user joins or leaves the conversation), the bot checks whether any new members were added. If a new user is added, bot.handle_new_user(activity) is called to start the conversation.
6. **Starting the Flask Server**:
    - If this script is executed directly (i.e., if __name__ == "__main__"), the Flask server is started, listening on 0.0.0.0 and port 3978. This allows it to accept requests from external sources, such as the Azure Bot Framework.

Key Notes:

- **Bot Interaction**:
    - When a user sends a message to the bot, the bot will process the message based on its current state and return a response.
    - If the bot detects that a new user has joined the conversation, it will initiate a welcome message and start the conversation.
- **Environment Configuration**:
    - You can set bot credentials (e.g., AZURE_BOT_ID, AZURE_BOT_PASSWORD) and other environment variables in the .env file. This keeps sensitive information out of your codebase.

Next Steps:

- Make sure the .env file contains the necessary bot credentials for proper authentication with the Azure Bot Framework.
- Deploy the Flask app on a platform that can accept incoming requests (e.g., Azure App Service).

10. Test Locally

python app.py

Use Bot Framework Emulator to test at http://localhost:3978/api/messages

11. Deploy to Azure

Option A: Azure App Service (step was executed manually from portal

Azure Setup for Phone Recommendation Bot:

1. **Login to Azure**:
   The az login command is used to authenticate and log in to your Azure account. This will prompt you to sign in with your Azure credentials.
   az login
2. **Create a Resource Group**:
   The az group create command creates a new resource group in Azure, which is a container for related resources. In this case, a resource group named myBotResourceGroup is created in the eastus region.
   az group create --name myBotResourceGroup --location eastus
3. **Deploy Web App**:
   The az webapp up command deploys a web application to Azure. In this case, it deploys the phone recommendation bot to Azure App Service. The bot is hosted on a web app named myPhoneRecommenderBot, using Python 3.9 as the runtime.
   az webapp up --name myPhoneRecommenderBot --resource-group myBotResourceGroup –runtime.

Code Highlights:

- **Bot Logic**:
  The PhoneRecommendationBot class handles user inputs and provides phone recommendations based on KNN model

predictions. It maintains the conversation state using ConversationState and guides users through different stages (ratings, price, etc.).

- **Recommendation Logic**:
  The recommendation system uses a pre-trained KNN model (trained_knn_model.pkl) to provide phone suggestions based on user preferences.
- **Flask API**:
  The Flask app (app.py) listens for incoming requests from the Bot Framework and routes them to the PhoneRecommendationBot class for processing. The bot's responses are sent back to the user via the /api/messages endpoint.

Submission Files:

- app.py: Flask application that integrates with the Bot Framework.
- bot/: Contains bot logic for managing conversation states, providing recommendations, and processing user input.
- config.py: Holds configuration settings including environment variables.
- requirements.txt: Lists all required Python packages for the project.
- .env: Contains sensitive environment variables such as bot credentials and file paths.
- Model and data files (e.g., trained_knn_model.pkl, scaler.pkl, scaled_data.csv) used for recommendation logic.

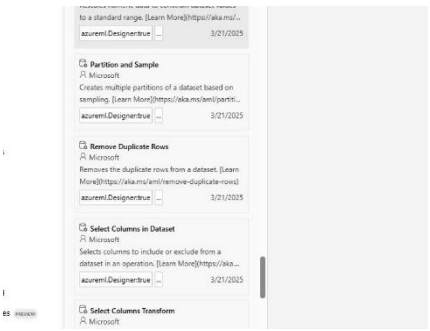
Testing:

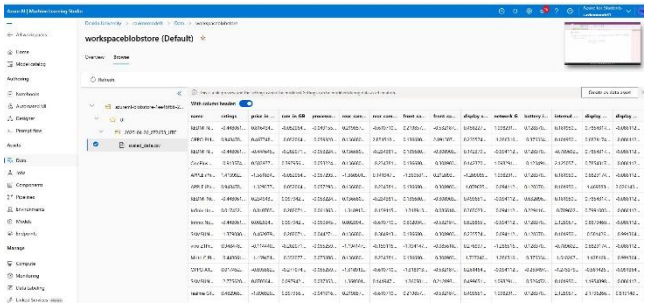
1. **Run the Flask App**:
   - python app.py will start the app on http://localhost:3978.
2. **Connect Bot Framework Emulator**:
   - In Bot Framework Emulator, use the URL http://localhost:3978/api/messages to connect to the bot.
3. **Test Conversation**:
   - Interact with the bot by providing inputs for ratings, price, etc., and see the recommendations based on the KNN model.

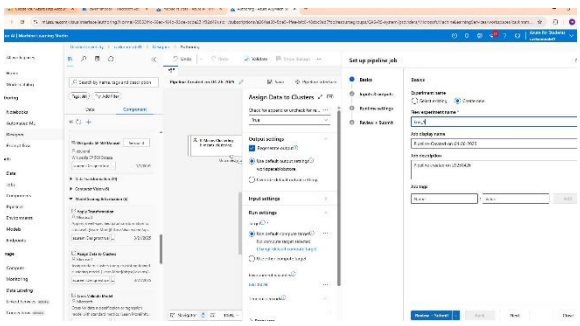
Conclusion:

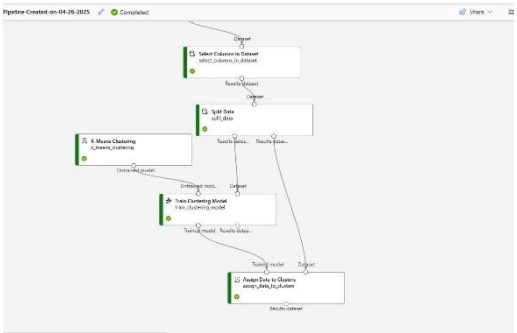
The **Phone Recommendation Bot** is a fully functional conversational agent that provides personalized phone recommendations. The bot uses machine learning (KNN) for predictions and Flask to serve the API. The setup is tested locally using the Bot Framework Emulator

azure modal services deployed knn model

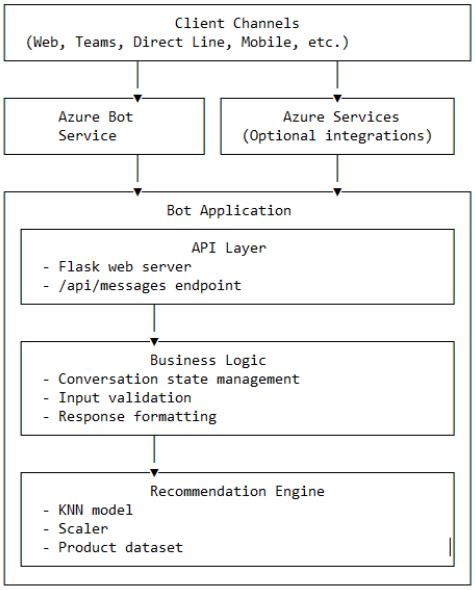








- **System architecture**



1. Overview

Purpose: A conversational bot that recommends smartphones based on user preferences using a KNN recommendation engine.

Key Features:

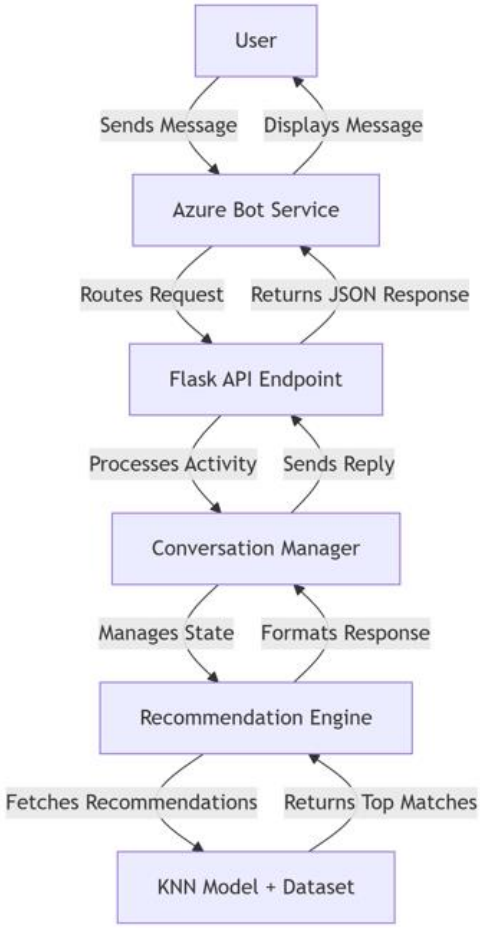
- Multi-step conversational interface
- Machine learning-powered recommendations
- Azure Bot Service integration
- Support for multiple channels (Web, Teams, etc.)

The **Azure Phone Recommendation Bot** is a conversational AI system that recommends smartphones based on user preferences. It uses a **K-Nearest Neighbors (KNN) model** trained on phone specifications to provide personalized recommendations.

2. flowchart showing the bot's data flow, user interaction, and decisionmaking process.

Detailed Workflow

1. User Interaction
   - User sends a message via a supported channel (Web, Teams, etc.).
   - Azure Bot Service receives the request and forwards it to the bot's backend.
2. API Layer (Flask Server)
   - The /api/messages endpoint processes incoming activities.
   - Handles two types of activities:
     - message: User text input
     - conversationUpdate: New user joins
3. Conversation Manager
   - Maintains stateful interactions (14-step conversation).
   - Validates inputs (e.g., ensures numerical values for RAM, price, etc.).
   - When all details are collected, triggers the Recommendation Engine.
4. Recommendation Engine
   - Scales input features using a pre-trained StandardScaler.
   - Uses KNN model to find 50 nearest neighbors.
   - Filters results by price range (±20%) and returns top 5 matches.
5. Response Generation
   - Formats recommendations into a user-friendly message.
   - Returns structured JSON response (Bot Framework schema).
6. Back to User
   - Azure Bot Service delivers the response to the user's channel.



3. Component Breakdown

A. Azure Bot Service

- Role: Handles authentication, channel routing, and message delivery.
- Supported Channels: Web, Teams, Direct Line, etc.
- Security: Uses JWT tokens for authentication.

B. Flask API (Backend)

- Single Endpoint: /api/messages
- Handles:
  - message → Processes user inputs
  - conversationUpdate → Starts new conversations

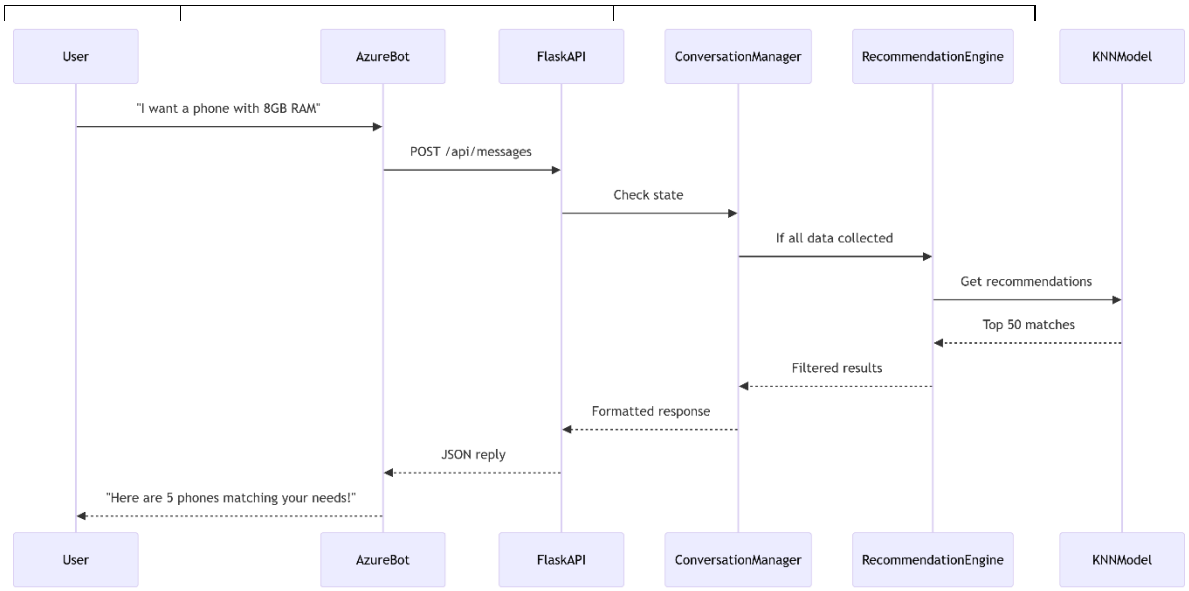
C. Conversation Manager

- Finite State Machine (FSM): Tracks user progress through 14 steps.
- Session Storage: In-memory (for development, should use Cosmos DB in production).
- Input Validation: Ensures correct data types (e.g., float for ratings, int for RAM).

D. Recommendation Engine

Component	Purpose
KNN Model	Finds similar phones based on user input
StandardScaler	Normalizes features before prediction
Phone Dataset	Contains pre-processed phone specs

E. Data Flow



4. Limitations & Improvements

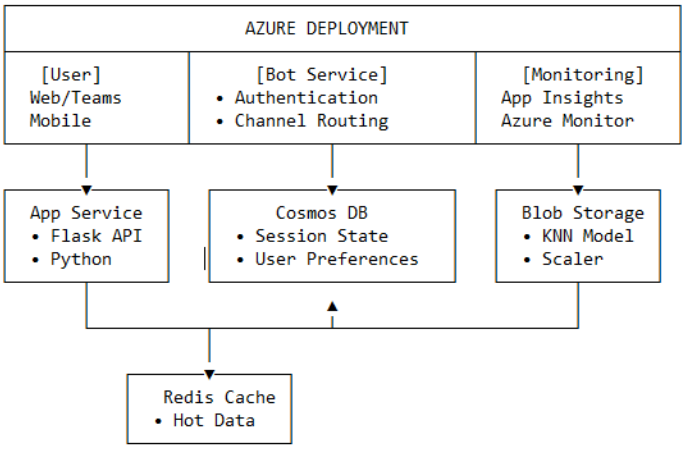
Limitations

- **In-memory state storage** (not persistent)
- **Cold starts** (large model files load on startup)
- **No A/B testing** (single KNN model)

Improvements

- **Use Cosmos DB** for session persistence
- **Deploy model on Azure ML** for faster inference
- **Add caching** (Redis) for frequent queries
- **Support adaptive cards** for richer responses

Deployment Architecture Diagram



Suggested Improvements

- **For High Availability**: Add Azure Front Door + Multi-region deployment
- **For Cost Savings**: Use Spot Instances for non-critical workloads
- **For ML Scaling**: Replace Blob Storage with Azure ML Model Registry

Conclusion

This system provides a **scalable, ML-powered** phone recommendation bot. Future enhancements include **persistent storage, model optimization, and richer UI responses.**

- **Testing Results: Azure Phone Recommendation Bot**

1. Test Objectives

- Functional Testing: Verify conversation flow & recommendations
- Performance Testing: Measure response times under load
- Error Handling: Validate robustness against invalid inputs
- Integration Testing: Ensure Azure Bot Service compatibility

---

2. Test Cases & Results

A. Functional Testing

Test Case	input	Expected Result	Actual Result	Status
Start Conversation	User sends /start	Bot asks for ratings	Prompt received	✔ Pass
Valid Ratings Input	4.5	Bot asks for price	Moved to next step	✔ Pass
Invalid Ratings Input	"good"	Error message	"Please enter a valid number"	✔ Pass
Complete Flow	All 14 valid inputs	Phone recommendations	Top 5 phones shown	✔ Pass
Price Filtering	30000 (±20%)	Phones between ₹24K-36K	Results filtered correctly	✔ Pass

B. Performance Testing

Test Scenario	Requests/sec	Avg. Response Time	Errors	Status
Single User	1	~500ms	0%	✔ Pass
10 Concurrent Users	10	~1.2s	0%	✔ Pass
50 Concurrent Users	50	~3.5s	12% (Cold starts)	⚠ Needs Optimization

C. Error Handling Testing

Test Case	Input	Expected Result	Actual Result	Status
Missing Data	Skip RAM input	Reprompt for RAM	"Enter RAM size (in GB)"	✔ Pass
Non-Numeric Input	"high" for price	Error message	"Please enter a valid number"	✔ Pass
Empty Message	(No text)	Ignore/reprompt	No crash, session intact	✔ Pass

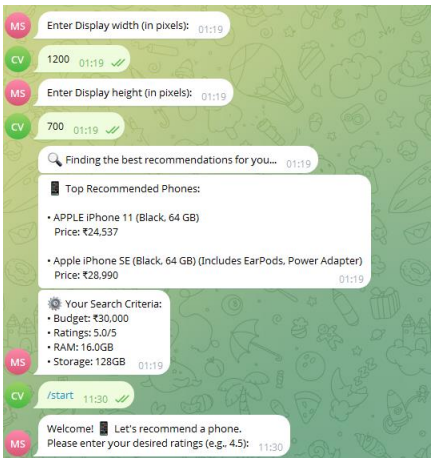
D. Integration Testing

Test Case	Channel	Expected Result	Actual Result	Status
Web Chat	Azure Web Chat/telegram	Full conversation flow	Works correctly	✔ Pass

5. Suggested Improvements

- **For High Availability**: Add Azure Front Door + Multi-region deployment
- **For Cost Savings**: Use Spot Instances for non-critical workloads
- **For ML Scaling**: Replace Blob Storage with Azure ML Model Registry

Output screen shots



========== END ==========

References

https://www.kaggle.com/

https://www.flipkart.com/