# Search Tool Documentation

the purpose of the **searchtool** is to provide a user-friendly interface for users to search for offers and retrieve relevant results based on the similarity of their search queries to the data stored in the dataframes provided. The script uses various data preprocessing techniques, fuzzy string matching, and vectorization methods to achieve this functionality.

## Step 1: Importing Required Libraries

```
import numpy as np
import pandas as pd
import spacy
import string
import gensim
import operator
import re
from fuzzywuzzy import fuzz
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import CountVectorizer
from flask import Flask, request, jsonify, render_template
```

This step involves importing necessary libraries such as NumPy, pandas, spacy, string, gensim, operator, regular expression (re), fuzzy string matching from fuzzywuzzy, cosine similarity from scikit-learn (sklearn), CountVectorizer, and Flask.

## Step 2: Flask App Initialization

```
app = Flask(__name__)
```

Here, a Flask web application named app is initialized.

## Step 3: Reading Data

```
# URLs of data files
brands_url =
'https://drive.google.com/file/d/1hPCAp4xx4PaDG_OLTJhYlhICc4WdgbDy/view?usp=sharing'
categories_url =
'https://drive.google.com/file/d/1SR6oGRnB4ULk9umnxZztHYKiHDI-VSag/view?usp=sharing'
offers_url =
'https://drive.google.com/file/d/18krLxNoBx9wFgtkXZiyL9Oh8B6w1hN40/view?usp=sharing'

# Constructing data file URLs
```

# Search Tool Documentation

```
brands_url = 'https://drive.google.com/uc?id=' +
brands_url.split('/')[-2]
categories_url = 'https://drive.google.com/uc?id=' +
categories_url.split('/')[-2]
offers_url = 'https://drive.google.com/uc?id=' +
offers_url.split('/')[-2]

# Reading data from URLs
brands_df = pd.read_csv(brands_url)
categories_df = pd.read_csv(categories_url)
offers_df = pd.read_csv(offers_url)
```

The script uses three main data sources:

- Brand Data: A CSV file containing brand information, including brand names and associated product categories.
- Category Data: A CSV file with category data, including parent-child relationships between categories.
- Offers Data: A CSV file containing offer details, including the offer text and retailer information.

## Step 4: Data Cleaning and Processing

```
# Renaming columns for consistency
brands_df.rename(columns={'BRAND_BELONGS_TO_CATEGORY':
'PRODUCT_CATEGORY'}, inplace=True)

# Dropping the RECEIPTS column as it is not needed for the tool
brands_df.drop("RECEIPTS", axis=1, inplace=True)

# Removing duplicate rows based on Brand and Product Category
brands_df.drop_duplicates(subset=['BRAND', 'PRODUCT_CATEGORY'],
inplace=True)

# Dropping rows with missing values in the 'BRAND' column
brands_df = brands_df.dropna()
```

Here, several data cleaning and processing steps are performed on the brands_df DataFrame:

- The column 'BRAND_BELONGS_TO_CATEGORY' is renamed to 'PRODUCT_CATEGORY' for consistency.
- The 'RECEIPTS' column is dropped from the DataFrame since it's not required for the tool. *(Assumptions: Receipt number are not usually required for offer search)*
- Duplicate rows with the same combination of 'BRAND' and 'PRODUCT_CATEGORY' are removed to ensure data uniqueness.

# Search Tool Documentation

- Rows with missing values in the 'BRAND' column are dropped (only 1 row dropped, not significant effect on dataset).

These steps help clean and prepare the brand data for further processing.

## Step 5: Retailer Data Handling

```
# Creating a list of unique non-null retailers
retailer_list = offers_df['RETAILER'].dropna().unique()

# Mapping retailers based on partial matching in offer text
def map_retailer(offer_text):
    for retailer in retailer_list:
        if retailer.lower() in offer_text.lower():
            return retailer
    return 'Unknown'

# Using fuzzy string matching to handle retailer mapping
def get_matching_string(str1, str2):
    similarity_score = fuzz.token_set_ratio(str1, str2)
    return str2 if similarity_score >= similarity_threshold else
'Unknown'

# ... (other functions)
```

Here, the retailer data is processed in several steps:

- A list of unique non-null retailers is created from the 'RETAILER' column of the offers_df DataFrame.
- The map_retailer function is defined to map retailers based on the presence of retailer keywords in offer text.
- The get_matching_string function is defined to handle partial matching of retailer names using fuzzy string matching.
- These steps address the issue of mapping retailers and handling variations in retailer names.

## Step 6: Defining Helper Functions

```
def map_retailer(offer_text):
    # ... (function implementation)
```

# Search Tool Documentation

```
def get_matching_string(str1, str2):
    # ... (function implementation)

def vectorize_data_based_on_metadata(product_input):
    # ... (function implementation)
```

Here, three helper functions are defined: map_retailer, get_matching_string, and vectorize_data_based_on_metadata. These functions handle retailer mapping, string similarity matching, and text vectorization, respectively.

## Step 7: Offer Retailer Mapping and Similarity Matching

```
# Finding similarity in keywords and mapping retailers
similarity_threshold = 50
offers_df['Match_Retailer'] = offers_df.apply(lambda row:
get_matching_string(row['OFFER'], row['RETAILER']), axis=1)

# Dropping unnecessary columns and renaming the column
columns_to_drop = ['Mapped Retailer', "RETAILER"]
offers_df.drop(columns_to_drop, axis=1, inplace=True)
offers_df.rename(columns={'Match_Retailer': 'RETAILER'},
inplace=True)
```

In this section, the retailer mapping process is further refined:

The get_matching_string function is used to find similarity in keywords between offer text and retailer names, and a threshold of 50 is used to determine matches.

Unnecessary columns (such as 'Mapped Retailer') are dropped, and the column 'Match_Retailer' is renamed to 'RETAILER'. These steps enhance the accuracy of mapping retailers and ensure consistent column names.

## Step 8: Creating Metadata and Text Vectorization

```
# Creating a metadata column by concatenating relevant columns
all_merge_df['metadata'] = all_merge_df.apply(lambda x:
x['BRAND'] + ' ' + x['PRODUCT_CATEGORY'] + ' ' +
x['IS_CHILD_CATEGORY_TO'] + ' ' + x['OFFER'], axis=1)

# Using CountVectorizer for text vectorization
```

# Search Tool Documentation

```
count_vec = CountVectorizer(stop_words='english')
count_vec_matrix =
count_vec.fit_transform(all_merge_df['metadata'])
```

## Step 9: Flask Routes for Index and Search

```
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')

@app.route('/search', methods=['GET','POST'])
def search_offers():
    # ... (function implementation)
```

These Flask route functions handle the main page ("/") and search page ("/search") endpoints. index() renders the main search interface, and search_offers() processes search queries and displays results.

**Index()**

This function is a route handler for the root ("/") endpoint in the Flask web application. It is responsible for rendering the main interface of the search tool. Here's what this function does:

- **Rendering the Template:** When a user accesses the root URL (the main page), the function renders an HTML template called "index.html." This template contains the user interface components, including a search bar and a search button.
- **User Interaction:** The template allows users to input their search queries and submit them using the search button. The form sends a POST request to the /search endpoint when the user submits a query.
- **Initial Display:** When the main page is accessed initially (not as a result of a search), the function renders the template with the search bar but without any search results.

**Search()**
Flask route function for the search page ("/search"). Handles POST requests containing user queries. Displays the search results in an HTML table.

This function is a route handler for the "/search" endpoint in the Flask web application. It handles both GET and POST requests related to searching for offers based on user queries. Here's a breakdown of its functionality:

# Search Tool Documentation

- **Request Handling:** This function checks if the incoming request is a POST request, which means the user has submitted a search query through the search form on the web interface.
- **Processing the Query:** If it's a POST request, the function retrieves the user's search query from the form data. It then calls the vectorize_data_based_on_metadata() function, passing the search query as input.
- **Rendering Search Results:** The function receives a DataFrame of similar offers and their corresponding similarity scores from the vectorize_data_based_on_metadata() function. It adds the similarity scores as a new column to the DataFrame and resets the index for proper display.
- **Rendering Templates:** If offers are found, the function renders an HTML template called "result.html" and passes the DataFrame as a variable to be displayed in an HTML table. If no offers are found, it displays an error message indicating that no similar products were found.
- **Error Handling:** If any errors occur during the search process or rendering of the template, the function displays a generic error message.

## Step 10: Running the Application

Loading the Requirements Run the script using the command

```
pip install -r requirements.txt.
```

This will start installing the required packages for Flask web application.

2) Starting the Application Run the script using the command

```
python searchtool_wsgi.py.
```

This will start the Flask web application.

3) Stop the Application Run the script using the command-on-command line.

```
Crtl + C
```

This will stop the Flask web application. Accessing the User Interface Open a web browser and navigate to access the search tool's user interface. Enter your search query and submit it to retrieve matching offers.

```
http://localhost:5000
```