4.CrewAi-Internet-Acesss

June 17, 2024

[1]:
```
import os
```

[2]:
```
%pwd
```

[2]:
```
'/mnt/d/Desktop/SuperteamsAI/News aggregator/Reseach'
```

[3]:
```
os.chdir("../")
```

[4]:
```
%pwd
```

[4]:
```
'/mnt/d/Desktop/SuperteamsAI/News aggregator'
```

[5]:
```
import logging
from pathlib import Path
logging.basicConfig(
    # filename='extract_data.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
```

[6]:
```
# from crewai import Agent, Task, Crew
# from langchain_openai import ChatOpenAI
# import os
# os.environ["OPENAI_API_KEY"] = "NA"

# llm = ChatOpenAI(
#     model = "crewai-llama3",
#     base_url = "http://localhost:11434/v1")
```

1 Content Planner Agent & Create planner task

[7]:
```
# import os
# from crewai import Agent, Task, Crew
# from langchain_openai import ChatOpenAI
# import requests
# from bs4 import BeautifulSoup
# import pandas as pd
# from newspaper import Article
# from pathlib import Path

# # Set up the OpenAI API key
# os.environ["OPENAI_API_KEY"] = "NA"

# # Set up the LLM for writer and editor
# llm = ChatOpenAI(
#     model="crewai-llama3",
#     base_url="http://localhost:11434/v1"
# )

# # Function to perform a Google search and return search results
# def google_search(query):
#     search_url = f"https://www.google.com/search?q={query}"
#     headers = {
#         "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"
#     }
#     response = requests.get(search_url, headers=headers)
#     response.raise_for_status()  # Raise an HTTPError for bad responses

#     soup = BeautifulSoup(response.text, 'html.parser')

#     search_results = []

#     for item in soup.select('div.g'):
#         title = item.select_one('h3')
#         link = item.select_one('a')['href']
#         if title and link:
#             search_results.append({
#                 "title": title.get_text(),
#                 "link": link
#             })

#     return search_results

# # Function to extract article attributes
# def extract_article_attributes(url):
#     try:
#         article = Article(url)
#         article.download()
#         article.parse()
#         return {
#             'authors': article.authors,
#             'text': article.text,
#             'title': article.title,
#             'link': url
#         }
#     except Exception as e:
#         print(f"Failed to process {url}: {e}")
#         return {
#             'authors': None,
#             'text': None,
#             'title': None,
#             'link': url
#         }

# # Function to perform a search, store results in a DataFrame, and extract␣
 ↪article attributes
# def search_and_store_to_dataframe(query, filename=None):
#     results = google_search(query)
#     df = pd.DataFrame(results)

#     if filename:
#         df.to_csv(filename, index=False)  # Save initial search results to␣
 ↪CSV file

#     # Extract article attributes
#     attributes_df = df['link'].apply(extract_article_attributes).apply(pd.
 ↪Series)
#     df = pd.concat([df, attributes_df], axis=1)

#     # Drop unnecessary columns and keep only 'link', 'authors', 'text', and␣
 ↪'title'
#     df = df[['link', 'authors', 'text', 'title']]

#     if filename:
#         filename = Path(filename).stem + "_with_attributes.csv"
#         df.to_csv(filename, index=False)  # Save DataFrame with attributes to␣
 ↪CSV file

#     print(df)

#     return df

# # Function to plan content
# def plan_content(topic):
#     query = topic
#     filename = "Dataset/search_results.csv"
#     df = search_and_store_to_dataframe(query, filename)
#     # Extract the necessary details for the content plan
#     latest_trends = df.head(5)  # Example of prioritizing the latest trends
#     target_audience = "General readers interested in the topic"
#     seo_keywords = ["example keyword1", "example keyword2"]  # These would be derived from the analysis
#     content_outline = {
#         "Introduction": "Brief introduction to the topic.",
#         "Key Points": latest_trends.to_dict('records'),
#         "Conclusion": "Summary and call to action."
#     }

#     content_plan = {
#         "Topic": topic,
#         "Target Audience": target_audience,
#         "SEO Keywords": seo_keywords,
#         "Content Outline": content_outline,
#         "Resources": latest_trends.to_dict('records')
#     }

#     return content_plan

# # Mock LLM class to simulate the planner agent's behavior
# class MockLLM:
#     def bind(self, *args, **kwargs):
#         def call(inputs):
#             return plan_content(inputs)
#         return call

# # Define the agents
# planner = Agent(
#     role="Content Planner",
#     goal="Plan engaging and factually accurate content on {topic}",
#     backstory="You're working on planning a blog article about the topic: {topic} on 'https://medium.com/'."
#               " You collect information that helps the audience learn something and make informed decisions."
#               " You have to prepare a detailed outline and the relevant topics and sub-topics that have to be part of the blog post."
#               " Your work is the basis for the Content Writer to write an article on this topic.",
#     llm=MockLLM(),
#     allow_delegation=False,
#     verbose=True
# )

# # Define the tasks for each agent
# plan_task = Task(
#     description=(
#         "1. Perform a Google search to find the latest trends, key players, and noteworthy news on {topic}.\n"
#         "2. Extract article attributes such as authors and main content.\n"
#         "3. Develop a detailed content outline including an introduction, key points, and a call to action.\n"
#         "4. Identify the target audience and include SEO keywords and relevant data or sources."
#     ),
#     expected_output="A comprehensive content plan document with an outline, audience analysis, SEO keywords, and resources.",
#     agent=planner,
#     action=lambda inputs: planner.llm.bind()(inputs["topic"])
# )

# writer = Agent(
#     role="Content Writer",
#     goal="Write insightful and factually accurate opinion piece about the topic: {topic}",
#     backstory="You're working on a writing a new opinion piece about the topic: {topic} in 'https://medium.com/'. "
#               "You base your writing on the work of the Content Planner, who provides an outline and relevant context about the topic. "
#               "You follow the main objectives and direction of the outline, as provide by the Content Planner. "
#               "You also provide objective and impartial insights and back them up with information provide by the Content Planner. "
#               "You acknowledge in your opinion piece when your statements are opinions as opposed to objective statements.",
#     allow_delegation=False,
#     llm=llm,
#     verbose=True
# )

# editor = Agent(
#     role="Editor",
#     goal="Edit and refine the article to ensure clarity, accuracy, and engagement",
#     backstory="You are an editor responsible for polishing the article to make it publish-ready.",
#     llm=llm,
#     allow_delegation=False,
#     verbose=True
# )

# write_task = Task(
#     description=(
#         "1. Use the content plan to craft a compelling blog post on {topic}.
↪\n"
#         "2. Incorporate SEO keywords naturally.\n"
#         "3. Sections/Subtitles are properly named in an engaging manner.\n"
#         "4. Ensure the post is structured with an engaging introduction,␣
↪insightful body, and a summarizing conclusion.\n"
#         "5. Proofread for grammatical errors and alignment with the brand's␣
↪voice.\n"
#     ),
#     expected_output="A well-written blog post in markdown format, ready for␣
↪publication, each section should have 2 or 3 paragraphs.",
#     agent=writer,
# )

# edit_task = Task(
#     description=("Proofread the given blog post for grammatical errors and␣
↪alignment with the brand's voice."),
#     expected_output="A well-written blog post in markdown format, ready for␣
↪publication, each section should have 2 or 3 paragraphs.",
#     agent=editor
# )

# # Crew setup with agents and tasks
# crew = Crew(
#     agents=[planner, writer, editor],
#     tasks=[plan_task, write_task, edit_task],
#     verbose=2
# )

# # Example function call for the content planner agent
# topic = "Modi takes office"
# inputs = {"topic": topic}
# result = crew.kickoff(inputs=inputs)

# print(result)
```

[8]:
```
# # Mock LLM class to simulate the planner agent's behavior
# class MockLLM:
#     def bind(self, *args, **kwargs):
#         def call(inputs):
#             return plan_content(inputs)
#         return call
```

[ ]:
```
import os
import time
```

```python
from crewai import Agent, Task, Crew
from langchain_openai import ChatOpenAI
import requests
from bs4 import BeautifulSoup
import pandas as pd
from newspaper import Article
from pathlib import Path

# Set up the OpenAI API key
os.environ["OPENAI_API_KEY"] = "NA"

# Set up the LLM for writer and editor
llm = ChatOpenAI(
    model="crewai-llama3",
    base_url="http://localhost:11434/v1"
)

# Function to perform a Google search and return search results
def google_search(query):
    search_url = f"https://www.google.com/search?q={query}"
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
↪537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"
    }
    search_results = []
    max_retries = 3
    retry_delay = 2

    retries = 0
    while retries < max_retries:
        try:
            response = requests.get(search_url, headers=headers)
            response.raise_for_status()  # Raise an HTTPError for bad responses

            soup = BeautifulSoup(response.text, 'html.parser')

            for item in soup.select('div.g'):
                title = item.select_one('h3')
                link = item.select_one('a')['href']
                if title and link:
                    search_results.append({
                        "title": title.get_text(),
                        "link": link
                    })
            break
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 429:  # Too Many Requests
```

```
                retries += 1
                print(f"Rate limit hit. Waiting for {retry_delay} seconds␣
↪before retrying... (Attempt {retries}/{max_retries})")
                time.sleep(retry_delay)
                retry_delay *= 2  # Exponential backoff
            else:
                raise e

    if retries == max_retries:
        print("Max retries reached. Failed to retrieve search results.")

    return search_results

# Function to extract article attributes
def extract_article_attributes(url):
    try:
        article = Article(url)
        article.download()
        article.parse()
        return {
            'authors': article.authors,
            'text': article.text,
            'title': article.title,
            'link': url
        }
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 403:
            print(f"403 Forbidden: Skipping URL {url}")
        else:
            print(f"Failed to process {url}: {e}")
        return None
    except Exception as e:
        print(f"Failed to process {url}: {e}")
        return None

# Function to perform a search, store results in a DataFrame, and extract␣
↪article attributes
def search_and_store_to_dataframe(query, filename=None):
    results = google_search(query)
    articles = [extract_article_attributes(result["link"]) for result in␣
↪results]
    articles = [article for article in articles if article is not None]  #␣
↪Filter out failed downloads
    df = pd.DataFrame(articles)

    if filename:
        df.to_csv(filename, index=False)  # Save DataFrame with attributes to␣
 ↪CSV file

    print(df)

    return df

# Function to plan content
def plan_content(topic):
    query = topic
    filename = "Dataset/search_results.csv"
    df = search_and_store_to_dataframe(query, filename)
    # Drop rows with any NaN values
    df_cleaned = df.dropna()

    # Display DataFrame after dropping NaN values
    print("\nDataFrame after dropping NaN values:")
    print(df_cleaned)
    # Extract the necessary details for the content plan
    latest_trends = df_cleaned.head(5)  # Example of prioritizing the latest␣
 ↪trends
    target_audience = "General readers interested in the topic"
    seo_keywords = ["example keyword1", "example keyword2"]  # These would be␣
 ↪derived from the analysis
    content_outline = {
        "Introduction": "Brief introduction to the topic.",
        "Key Points": latest_trends.to_dict('records'),
        "Conclusion": "Summary and call to action."
    }

    content_plan = {
        "Topic": topic,
        "Target Audience": target_audience,
        "SEO Keywords": seo_keywords,
        "Content Outline": content_outline,
        "Resources": latest_trends.to_dict('records')
    }

    # Convert content_plan to a string for return
    content_plan_str = (
        f"Topic: {content_plan['Topic']}\n"
        f"Target Audience: {content_plan['Target Audience']}\n"
        f"SEO Keywords: {', '.join(content_plan['SEO Keywords'])}\n"
        f"Content Outline: \n"
        f"  Introduction: {content_outline['Introduction']}\n"
        f"  Key Points: {content_outline['Key Points']}\n"
        f"  Conclusion: {content_outline['Conclusion']}\n"
        f"Resources: {content_plan['Resources']}"
    )

    return content_plan_str

# Mock LLM class to simulate the planner agent's behavior
class MockLLM:
    def bind(self, *args, **kwargs):
        def call(inputs):
            return plan_content(inputs)
        return call

# Define the agents
planner = Agent(
    role="Content Planner",
    goal="Plan engaging and factually accurate content on {topic}",
    backstory="You're working on planning a blog article about the topic:␣
  ↪{topic} on 'https://medium.com/'."
              " You collect information that helps the audience learn something␣
  ↪and make informed decisions."
              " You have to prepare a detailed outline and the relevant topics␣
  ↪and sub-topics that have to be part of the blog post."
              " Your work is the basis for the Content Writer to write an␣
  ↪article on this topic.",
    llm=MockLLM(),
    allow_delegation=False,
    verbose=True
)

# Define the tasks for each agent
plan_task = Task(
    description=(
        "1. Perform a Google search to find the latest trends, key players, and␣
  ↪noteworthy news on {topic}.\n"
        "2. Extract article attributes such as authors and main content.\n"
        "3. Develop a detailed content outline including an introduction, key␣
  ↪points, and a call to action.\n"
        "4. Identify the target audience and include SEO keywords and relevant␣
  ↪data or sources."
    ),
    expected_output="A comprehensive content plan document with an outline,␣
  ↪audience analysis, SEO keywords, and resources.",
    agent=planner,
    action=lambda inputs: planner.llm.bind()(inputs)
)
```

```
writer = Agent(
    role="Content Writer",
    goal="Write insightful and factually accurate opinion piece about the topic:
 ↪ {topic}",
    backstory="You're working on a writing a new opinion piece about the topic:␣
 ↪{topic} in 'https://medium.com/'. "
              "You base your writing on the work of the Content Planner, who␣
 ↪provides an outline and relevant context about the topic. "
              "You follow the main objectives and direction of the outline, as␣
 ↪provide by the Content Planner. "
              "You also provide objective and impartial insights and back them␣
 ↪up with information provide by the Content Planner. "
              "You acknowledge in your opinion piece when your statements are␣
 ↪opinions as opposed to objective statements.",
    allow_delegation=False,
    llm=llm,
    verbose=True
)

editor = Agent(
    role="Editor",
    goal="Edit and refine the article to ensure clarity, accuracy, and␣
 ↪engagement",
    backstory="You are an editor responsible for polishing the article to make␣
 ↪it publish-ready.",
    llm=llm,
    allow_delegation=False,
    verbose=True
)

write_task = Task(
    description=(
        "1. Use the content plan to craft a compelling blog post on {topic}.\n"
        "2. Incorporate SEO keywords naturally.\n"
        "3. Sections/Subtitles are properly named in an engaging manner.\n"
        "4. Ensure the post is structured with an engaging introduction,␣
 ↪insightful body, and a summarizing conclusion.\n"
        "5. Proofread for grammatical errors and alignment with the brand's␣
 ↪voice.\n"
    ),
    expected_output="A well-written blog post in markdown format, ready for␣
 ↪publication, each section should have 2 or 3 paragraphs.",
    agent=writer,
)

edit_task = Task(
    description=("Proofread the given blog post for grammatical errors and␣
↪alignment with the brand's voice."),
    expected_output="A well-written blog post in markdown format, ready for␣
↪publication, each section should have 2 or 3 paragraphs.",
    agent=editor
)

# Crew setup with agents and tasks
crew = Crew(
    agents=[planner, writer, editor],
    tasks=[plan_task, write_task, edit_task],
    verbose=2
)

# Example function call for the content planner agent
topic = "Modi takes office"
inputs = {"topic": topic}
result = crew.kickoff(inputs=inputs)

print(result)
```

[ ]:
```
from IPython.display import Markdown,display
display(Markdown(result))
```

[ ]:
```
```