5.CrewAi-qdrantimptopics-Internet-Acesss

June 17, 2024

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

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

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

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

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

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

[5]:
```python
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]:
```python
# 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]:
```python
# 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
```

2 Loading embeddding Model

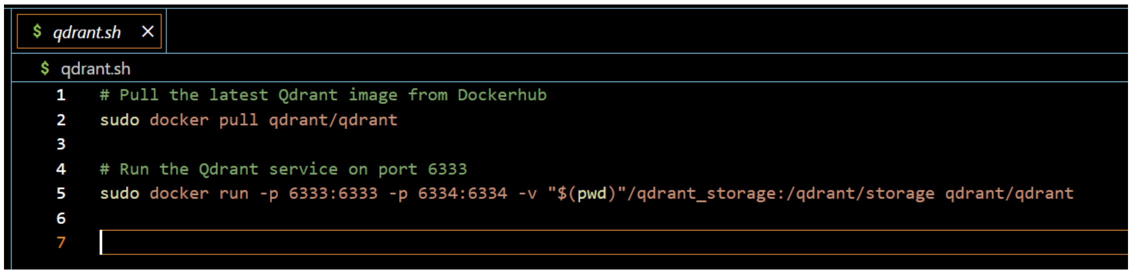
[9]:
```
from sentence_transformers import SentenceTransformer
```

```
/mnt/d/Desktop/SuperteamsAI/News aggregator/linuxNewsAI/lib/python3.10/site-
packages/sentence_transformers/cross_encoder/CrossEncoder.py:11: TqdmWarning:
IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from tqdm.autonotebook import tqdm, trange
```

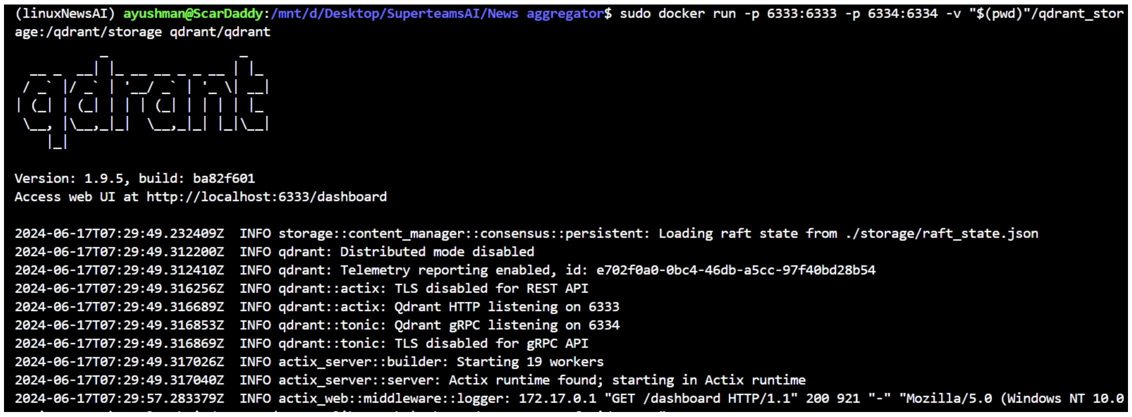
[10]:
```
# from transformers import AutoTokenizer, AutoModel
# from pathlib import Path

# def download_model_and_tokenizer(model_name, save_path):
#     """
#     Download and save both the model and the tokenizer to the specified␣
↪directory.

#     Parameters:
#         model_name (str): Name of the model to download.
#         save_path (str or Path): Path to the directory where the model and␣
↪tokenizer will be saved.
#     """
#     # Create the save path if it doesn't exist
#     save_path = Path(save_path)
#     save_path.mkdir(parents=True, exist_ok=True)

#     # Initialize tokenizer and model
#     tokenizer = AutoTokenizer.from_pretrained(model_name)
#     model = AutoModel.from_pretrained(model_name)

#     # Save tokenizer
#     tokenizer.save_pretrained(save_path)

#     # Save model
#     model.save_pretrained(save_path)

# # Example usage
# model_name = 'sentence-transformers/all-MiniLM-L12-v2'  # Model name to␣
↪download
# save_path = Path("MiniLM-L12-v2/")  # Path where model and tokenizer will be␣
↪saved
# download_model_and_tokenizer(model_name, save_path)
```

[11]:
```
from transformers import AutoTokenizer, AutoModel

def load_model_and_tokenizer(model_path):
```

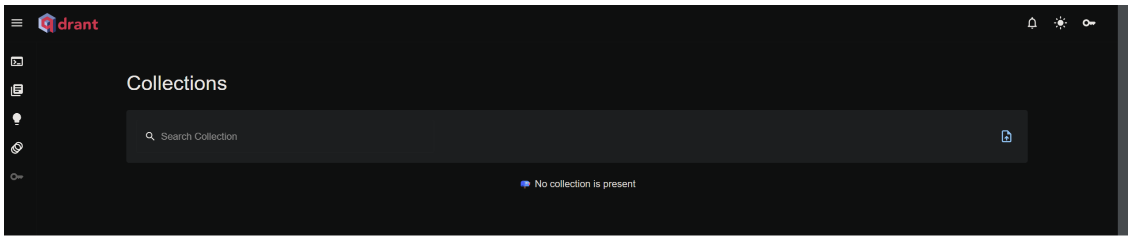
```python
    """
    Load the model and tokenizer from the specified directory.

    Parameters:
        model_path (str or Path): Path to the directory containing the saved␣
↪model and tokenizer.

    Returns:
        tokenizer (transformers.PreTrainedTokenizer): Loaded tokenizer.
        model (transformers.PreTrainedModel): Loaded model.
    """
    model_path = Path(model_path)
    tokenizer = AutoTokenizer.from_pretrained(model_path)
    model = AutoModel.from_pretrained(model_path)
    return tokenizer, model

# # Load the model and tokenizer
# model_path = Path("MiniLM-L12-v2/")
# tokenizer, model = load_model_and_tokenizer(model_path)
```



```bash
# Pull the latest Qdrant image from Dockerhub
sudo docker pull qdrant/qdrant

# Run the Qdrant service on port 6333
sudo docker run -p 6333:6333 -p 6334:6334 -v "$(pwd)"/qdrant_storage:/qdrant/storage qdrant/qdrant

```



```
(linuxNewsAI) ayushman@ScarDaddy:/mnt/d/Desktop/SuperteamsAI/News aggregator$ sudo docker run -p 6333:6333 -p 6334:6334 -v "$(pwd)"/qdrant_stor
age:/qdrant/storage qdrant/qdrant
           _                 _
  __ _  __| |_ __ __ _ _ __ | |_
 / _` |/ _` | '__/ _` | '_ \| __|
| (_| | (_| | | | (_| | | | | |_
 \__, |\__,_|_|  \__,_|_| |_|\__|
    |_|

Version: 1.9.5, build: ba82f601
Access web UI at http://localhost:6333/dashboard

2024-06-17T07:29:49.232409Z  INFO storage::content_manager::consensus::persistent: Loading raft state from ./storage/raft_state.json
2024-06-17T07:29:49.312200Z  INFO qdrant: Distributed mode disabled
2024-06-17T07:29:49.312410Z  INFO qdrant: Telemetry reporting enabled, id: e702f0a0-0bc4-46db-a5cc-97f40bd28b54
2024-06-17T07:29:49.316256Z  INFO qdrant::actix: TLS disabled for REST API
2024-06-17T07:29:49.316689Z  INFO qdrant::actix: Qdrant HTTP listening on 6333
2024-06-17T07:29:49.316853Z  INFO qdrant::tonic: Qdrant gRPC listening on 6334
2024-06-17T07:29:49.316869Z  INFO qdrant::tonic: TLS disabled for gRPC API
2024-06-17T07:29:49.317026Z  INFO actix_server::builder: Starting 19 workers
2024-06-17T07:29:49.317040Z  INFO actix_server::server: Actix runtime found; starting in Actix runtime
2024-06-17T07:29:57.283379Z  INFO actix_web::middleware::logger: 172.17.0.1 "GET /dashboard HTTP/1.1" 200 921 "-" "Mozilla/5.0 (Windows NT 10.0
```



```
[12]: import os
import logging
import time
import ast
import torch
import pandas as pd
from pathlib import Path
from crewai import Agent, Task, Crew
from langchain_openai import ChatOpenAI
import requests
from bs4 import BeautifulSoup
from newspaper import Article
from transformers import AutoTokenizer, AutoModel
from sentence_transformers import SentenceTransformer
from qdrant_client import QdrantClient
from qdrant_client.http.models import Distance, VectorParams, PointStruct
from langchain.vectorstores import Qdrant

# 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
```

```python
    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 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)

    # Drop rows with any NaN values
    df_cleaned = df.dropna()

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

    print(df_cleaned)

    return df_cleaned

# Load the model and tokenizer
model_path = Path("MiniLM-L12-v2/")
tokenizer, model = AutoTokenizer.from_pretrained(model_path), AutoModel.
↪from_pretrained(model_path)

# Mean Pooling - Take attention mask into account for correct averaging
def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]  # First element of model_output␣
↪contains all token embeddings
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.
↪size()).float()
    return torch.sum(token_embeddings * input_mask_expanded, 1) / torch.
↪clamp(input_mask_expanded.sum(1), min=1e-9)

def generate_embedding(text):
    # Tokenize input text
    encoded_input = tokenizer(text, padding=True, truncation=True,␣
↪return_tensors='pt')
    # Compute token embeddings with model
    with torch.no_grad():
        model_output = model(**encoded_input)
    # Perform mean pooling
    sentence_embedding = mean_pooling(model_output,␣
↪encoded_input['attention_mask'])
    # Convert to numpy for FAISS compatibility and ensure it's 2D
    return sentence_embedding.cpu().numpy().reshape(1, -1)

# Initialize Qdrant client
qdrant_client = QdrantClient(host='localhost', port=6333)
collection_name = "News"

# Specify the vectors' configuration
vectors_config = VectorParams(
    size=model.config.hidden_size,  # The size of your embeddings
    distance=Distance.COSINE  # The distance metric for the vector space
)

# Create or recreate the collection with the specified configuration
qdrant_client.recreate_collection(
    collection_name=collection_name,
    vectors_config=vectors_config,
)

# Function to insert data into Qdrant
def insert_data_into_qdrant(df):
    for index, row in df.iterrows():
        qdrant_client.upsert(
            collection_name=collection_name,
            points=[PointStruct(
                id=index,  # Using the dataframe index as the ID
                vector=row['encoded_title'],  # Assuming row['encoded_title']␣
 ↪is a list of floats
                payload={
                    "title": row['title'],
                    "text": row['text'],
                    "authors": row['authors'],
                    "link": row['link']
                }
            )]
        )

# Function to search for similar content using Qdrant
def similarity_search_with_score(query, k=5):
    query_embedding = generate_embedding(query)[0].tolist()
    search_results = qdrant_client.search(
        collection_name=collection_name,
        query_vector=query_embedding,
        limit=k,
        with_payload=True,
        with_vectors=False
    )
    return search_results
```

```python
# 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()

    # Encode the titles for similarity search
    df_cleaned['encoded_title'] = df_cleaned['title'].apply(lambda x:␣
 ↪generate_embedding(x)[0].tolist())

    # Insert data into Qdrant
    insert_data_into_qdrant(df_cleaned)

    # Perform similarity search
    search_results = similarity_search_with_score(query=query, k=2)

    # Extract necessary details for the content plan
    latest_trends = []
    for result in search_results:
        payload = result.payload
        latest_trends.append({
            "title": payload.get('title', 'No content available'),
            "text": payload.get('text', 'No content available'),
            "authors": payload.get('authors', 'No content available'),
            "link": payload.get('link', 'No content available')
        })

    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,
        "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
    }
    # 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):
            # Directly use inputs as it should be a string
            topic = str(inputs)  # Ensure the topic is treated as a string
            return plan_content(topic)
        return call

# Define the agents
Content_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
Content_planner_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=Content_planner,
    action=lambda inputs: Content.llm.bind()(inputs)
)


# Crew setup with agents and tasks
# crew = Crew(
#     agents=[Content_planner],
#     tasks=[Content_planner_task],
#     verbose=2
# )

# # Example function call for the content planner agent
# topic = "IN 2024 Indian prime minister Narendra Modi takes office again"
# inputs = {"topic": topic}
# result = crew.kickoff(inputs=inputs)

# print(result)
```

```
/tmp/ipykernel_104044/1952010219.py:134: DeprecationWarning:
`recreate_collection` method is deprecated and will be removed in the future.
Use `collection_exists` to check collection existence and `create_collection`
instead.
  qdrant_client.recreate_collection(
2024-06-17 14:19:36 - INFO - HTTP Request: DELETE
http://localhost:6333/collections/News "HTTP/1.1 200 OK"
2024-06-17 14:19:37 - INFO - HTTP Request: PUT
http://localhost:6333/collections/News "HTTP/1.1 200 OK"
```

[13]:
```
# from IPython.display import Markdown,display
# display(Markdown(result))
```

3 All agents

[14]:
```
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
)
write = 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,
    action=lambda inputs: writer.llm.bind()(inputs)
)

editor = Agent(
    role="Editor",
    goal="Edit a given blog post to align with "
         "the writing style of the organization 'https://medium.com/'. ",
    backstory="You are an editor who receives a blog post "
              "from the Content Writer. "
              "Your goal is to review the blog post "
              "to ensure that it follows journalistic best practices,"
              "provides balanced viewpoints "
```

```
                "when providing opinions or assertions, "
                "and also avoids major controversial topics "
                "or opinions when possible.",
    llm=llm,
    allow_delegation=False,
    verbose=True
)
edit = 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,
    action=lambda inputs: editor.llm.bind()(inputs)
)
```

[ ]:
```
crew = Crew(
    agents=[Content_planner, writer, editor],
    tasks=[Content_planner_task, write, edit],
    verbose=2
)

# Example function call for the content planner agent
topic = "IN 2024 Indian prime minister Narendra Modi takes office again"
inputs = {"topic": topic}
result = crew.kickoff(inputs=inputs)

print(result)
```

```
2024-06-17 14:26:39 - WARNING - Overriding of current TracerProvider is not
allowed

 [DEBUG]: == Working Agent: Content Planner
 [INFO]: == Starting Task: 1. Perform a Google search to find the

latest trends, key players, and noteworthy news on IN 2024 Indian prime minister

Narendra Modi takes office again.

2. Extract article attributes such as authors and main content.

3. Develop a detailed content outline including an introduction, key points, and

a call to action.

4. Identify the target audience and include SEO keywords and relevant data or

sources.
```

```
> Entering new CrewAgentExecutor chain…
Failed to process http://www.pmindia.gov.in/en/: Article `download()` failed
with 403 Client Error: Forbidden for url: http://www.pmindia.gov.in/en/ on URL
http://www.pmindia.gov.in/en/
```

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

**

Here's the revised blog post in markdown format:

4 Narendra Modi's Recent Election Performance: A Turning Point for Indian Politics?

As India's 2019 general election results are still fresh, many are left wondering what Narendra Modi's landslide victory means for the country's future. With the Bharatiya Janata Party (BJP) securing an unprecedented second term, it's essential to examine the implications of this outcome on Indian politics.

The sheer magnitude of the BJP's victory is undeniable. Having won 303 out of a possible 543 seats in the Lok Sabha (House of Representatives), the party has demonstrated its unparalleled ability to mobilize voters and build a strong political machine. However, this dominance raises concerns about the erosion of democratic institutions and the silencing of opposition voices. The opposition parties' dismal performance underscores the need for them to re-evaluate their strategies and messaging.

For Modi himself, this triumph will likely embolden him to pursue his ambitious agenda, which includes revamping India's economy, addressing environmental challenges, and promoting nationalistic sentiments. While some may view these priorities as positive, others might perceive them as divisive or detrimental to the country's secular fabric. As the government continues to grapple with pressing issues like unemployment, corruption, and infrastructure development, it is crucial that the prime minister remains committed to fostering an inclusive and diverse India.

Section 2: The Way Forward for Indian Politics

With Modi at the helm, the BJP will likely continue to drive the national agenda. However, this dominance also presents opportunities for constructive opposition. Rather than simply criticizing the government, the opposition must adopt a more nuanced approach that focuses on building a strong coalition of allies and offering feasible policy alternatives.

The outcome also underscores the significance of local and regional politics in India. As state-level elections become increasingly important, parties will need to adapt their strategies to address these specific contexts. This may involve adopting more personalized approaches, leveraging social media platforms, or fostering grassroots movements that resonate with local populations.

5 Conclusion

Narendra Modi's election victory marks a significant moment in Indian politics. While some might view it as a triumph of authoritarianism over democratic norms, others see it as an endorsement of the prime minister's policies and leadership style. As we move forward, it is vital that both supporters and detractors engage in constructive dialogue and critical thinking to shape the country's future trajectory.

I have edited the blog post to ensure alignment with Medium's writing style and voice, while also maintaining the author's original perspective and insights. The revised text has been reorganized into sections, with concise headings and clear language that is easy to follow. I hope this final answer meets your expectations!

```
[ ]:
```