gemini_reddit_analyzer.py

First

```
cd Desktop
```

Second

```
export API_KEY='AIzaSyDpvOoXDinBegtRkSMFjCbRE_iYGiDH32M'
export REDDIT_CLIENT_ID='QPAItQzJ0x7H3MOpj7BCcA'
export REDDIT_CLIENT_SECRET='fFW1Dgik95qmfhzD9rCzkPrdXuHfpg'
export REDDIT_USER_AGENT='Supersnu'
```

Run

```
python3 gemini_reddit_analyzer.py "Starlink" 500
```

Or

```
python3 gemini_reddit_analyzer.py "Another Keyword" 100
```

Code

```
import sys
import time
import csv
import praw
import os
from datetime import datetime, timezone
from typing import List, Dict, Any
import json
import google.generativeai as genai

# === Environment Variable Names ===
ENV_GEMINI_API_KEY = 'AIzaSyDpvOoXDinBegtRkSMFjCbRE_iYGiDH32M' # This is the required name for Gemir
REDDIT_CLIENT_ID = 'QPAItQzJ0x7H3MOpj7BCcA'  # Your Client ID
REDDIT_CLIENT_SECRET = 'fFW1Dgik95qmfhzD9rCzkPrdXuHfpg'  # Your Client Secret
REDDIT_USER_AGENT = 'Destroying_Angel_Snuferno'

# === Gemini API settings ===
GEMINI_MODEL_NAME = 'gemini-2.5-flash-preview-04-17' # Recommended model

# Batch size for Gemini API calls (adjust based on testing and rate limits)
BATCH_SIZE = 10 # Reduced batch size for Gemini to manage prompt length better
API_RETRY_DELAY = 5 # Seconds to wait before retrying API call

def get_env_variable(var_name: str) -> str:
    """Gets an environment variable or exits if not found."""
    value = os.environ.get(var_name)
    if not value:
        print(f"Error: Environment variable {var_name} not set.")
        sys.exit(1)
    return value

def authenticate_reddit() -> praw.Reddit:
    """Authenticate and return a Reddit instance."""
    client_id = get_env_variable(ENV_REDDIT_CLIENT_ID)
    client_secret = get_env_variable(ENV_REDDIT_CLIENT_SECRET)
```

```
    user_agent = get_env_variable(ENV_REDDIT_USER_AGENT)

    reddit = praw.Reddit(
        client_id=client_id,
        client_secret=client_secret,
        user_agent=user_agent
    )
    print("Successfully authenticated with Reddit.")
    return reddit

def fetch_reddit_data(reddit: praw.Reddit, keyword: str, limit: int = 500) -> List[Dict]:
    """
    Fetch Reddit posts and comments matching the keyword.
    Returns a list of dicts with required fields.
    """
    results = []
    count = 0

    print(f"Searching for submissions with keyword '{keyword}'...")
# Search submissions with the keyword in all subreddits
try:
    submissions = reddit.subreddit('all').search(keyword, limit=None) # Fetch more initially if needed

    for submission in submissions:
        if count >= limit:
            break

        # Add submission as a post
        results.append({
            'id': count + 1,
            'topic': keyword,
            'text': submission.title + ("\n" + submission.selftext if submission.selftext else ""),
            'timestamp': datetime.fromtimestamp(submission.created_utc, timezone.utc).isoformat(),
            'source': 'reddit_post', # More specific source
            'notes': '',
            'username': str(submission.author) if submission.author else 'unknown',
            'upvotes': submission.score,
            'type': 'post'
        })
        count += 1
        if count % 50 == 0:
            print(f"Fetched {count} items so far...")

        # Fetch comments for this submission
        if count < limit :
            submission.comments.replace_more(limit=0) # Get top-level comments
            for comment in submission.comments.list():
                if count >= limit:
                    break
                results.append({
                    'id': count + 1,
                    'topic': keyword,
                    'text': comment.body,
                    'timestamp': datetime.fromtimestamp(comment.created_utc, timezone.utc).isoformat(),
                    'source': 'reddit_comment', # More specific source
                    'notes': '',
                    'username': str(comment.author) if comment.author else 'unknown',
                    'upvotes': comment.score,
                    'type': 'comment'
                })
```

```
                count += 1
                if count % 50 == 0:
                    print(f"Fetched {count} items so far...")
    except praw.exceptions.PrawcoreException as e:
        print(f"Reddit API error: {e}")
        print("Please check your Reddit API credentials and network connection.")
        sys.exit(1)
    except Exception as e:
        print(f"An unexpected error occurred during Reddit data fetching: {e}")
        sys.exit(1)

    return results[:limit]

def build_gemini_batch_prompt(batch: List[Dict]) -> str:
    """Build the prompt for Gemini API batch sentiment analysis."""
    system_instruction = (
        "You are a sentiment analysis assistant. I will provide you with a list of texts. "
        "For each text, classify the sentiment as 'Positive', 'Negative', or 'Neutral', "
        "and provide a confidence 'score' between 0.0 and 1.0. "
        "Consider upvotes as minor context if available, but base your classification primarily on the text content. "
        "Respond ONLY with a valid JSON array of objects. Each object in the array must correspond "
        "to one input text, in the exact same order, and have two keys: \"sentiment\" (string) and "
        "\"score\" (float).\n"
        "Example for two texts:\n"
        "[\n"
        "  {\"sentiment\": \"Positive\", \"score\": 0.95},\n"
        "  {\"sentiment\": \"Negative\", \"score\": 0.88}\n"
        "]\n\n"
        "Here are the texts to analyze:\n"
    )

    user_texts_parts = []
    for i, item in enumerate(batch):
        text_content = item['text'].replace('\n', ' ').strip() # Clean up text a bit
        if not text_content: # Handle empty text, assign neutral
            text_content = "No content provided."
        upvotes = item.get('upvotes', 0)
        user_texts_parts.append(
            f"Text {i+1}:\n"
            f"Content: \"{text_content}\"\n"
            f"Upvotes: {upvotes}\n"
        )

    return system_instruction + "\n".join(user_texts_parts) + "\nPlease provide the JSON array as specified."


def call_gemini_api_batch(batch: List[Dict], gemini_api_key: str) -> List[Dict[str, Any]]:
    """
    Call Gemini API for a batch of texts and get sentiment classifications.
    Returns a list of dicts with 'sentiment' and 'score' for each text.
    """
    if not batch:
        return []

    genai.configure(api_key=gemini_api_key)
    model = genai.GenerativeModel(GEMINI_MODEL_NAME)

    prompt = build_gemini_batch_prompt(batch)

    # print(f"\n--- Gemini Prompt for Batch (first 200 chars) ---\n{prompt[:200]}...\n-------------------------")
```

```
try:
    response = model.generate_content(
        prompt,
        generation_config=genai.types.GenerationConfig(
            response_mime_type="application/json",
            temperature=0.1 # Low temperature for more deterministic sentiment
        )
    )

    json_str = response.text.strip()
    # print(f"Raw Gemini response: {json_str}") # Debug: raw response

    # Advanced JSON cleaning (handles potential markdown and leading/trailing text)
    fence_regex = r"```(?:json)?\s*\n?(.*?)\n?\s*```"
    match = re.search(fence_regex, json_str, re.DOTALL)
    if match:
        json_str = match.group(1).strip()
    else:
        # Try to find the start of a JSON array if no fence is present
        array_start_index = json_str.find('[')
        array_end_index = json_str.rfind(']')
        if array_start_index != -1 and array_end_index != -1 and array_end_index > array_start_index:
            json_str = json_str[array_start_index : array_end_index + 1]


sentiment_results = json.loads(json_str)

if not isinstance(sentiment_results, list) or len(sentiment_results) != len(batch):
    print(f"Error: Gemini response is not a list or length mismatch. Expected {len(batch)}, got {len(sentiment_
    print(f"Problematic JSON string: {json_str}")
    return [{"sentiment": "Neutral", "score": 0.5, "error": "Response format error"} for _ in batch]

# Validate individual items
validated_results = []
for i, res in enumerate(sentiment_results):
    if isinstance(res, dict) and "sentiment" in res and "score" in res:
        # Ensure sentiment is one of the expected values
        valid_sentiments = ["Positive", "Negative", "Neutral"]
        if res["sentiment"] not in valid_sentiments:
            print(f"Warning: Item {i} in batch has invalid sentiment '{res['sentiment']}'. Defaulting to Neutral.")
            res["sentiment"] = "Neutral"
            res["score"] = 0.5 # Reset score for invalid sentiment
        validated_results.append({"sentiment": str(res["sentiment"]), "score": float(res["score"])})
    else:
        print(f"Warning: Item {i} in batch has invalid structure: {res}. Defaulting to Neutral.")
        validated_results.append({"sentiment": "Neutral", "score": 0.5, "error": "Invalid item structure"})
    return validated_results

except json.JSONDecodeError as e:
    print(f"JSON decode error from Gemini response: {e}")
    print(f"Problematic Gemini response text: {response.text if 'response' in locals() and hasattr(response, 'text
    return [{"sentiment": "Neutral", "score": 0.5, "error": "JSON decode error"} for _ in batch]
except Exception as e:
    # Catch other google.api_core.exceptions.GoogleAPIError or genai specific errors
    print(f"Error calling Gemini API: {e}")
    return [{"sentiment": "Neutral", "score": 0.5, "error": str(e)} for _ in batch]


def batch_process_sentiment_with_gemini(data: List[Dict], gemini_api_key: str) -> List[Dict]:
```

```
    """
    Process data in batches through Gemini API and add sentiment results.
    """
    all_results_with_sentiment = []
    total_items = len(data)
    total_batches = (total_items + BATCH_SIZE - 1) // BATCH_SIZE

    for i in range(0, total_items, BATCH_SIZE):
        batch_num = i // BATCH_SIZE + 1
        batch_data = data[i:i+BATCH_SIZE]

        print(f"Processing batch {batch_num} of {total_batches} with {len(batch_data)} items...")

        # Retry logic for API calls
        max_retries = 3
        retries = 0
        sentiments = []
        success = False
        while retries < max_retries and not success:
            sentiments = call_gemini_api_batch(batch_data, gemini_api_key)
            # Check if all items in sentiments have a valid sentiment (not just error placeholders)
            if all("error" not in s for s in sentiments) and len(sentiments) == len(batch_data):
                success = True
            else:
                retries += 1
                print(f"  Batch {batch_num} failed. Retrying ({retries}/{max_retries}) in {API_RETRY_DELAY}s...")
                time.sleep(API_RETRY_DELAY)

        if not success:
            print(f"  Batch {batch_num} failed after {max_retries} retries. Assigning Neutral sentiment.")
            sentiments = [{"sentiment": "Neutral", "score": 0.5, "error": "API call failed after retries"} for _ in batch_da


        current_time = datetime.now(timezone.utc).isoformat()
        for item_idx, (original_item, sentiment_info) in enumerate(zip(batch_data, sentiments)):
            # Merge sentiment info into the original item
            updated_item = original_item.copy()
            updated_item['sentiment'] = sentiment_info.get('sentiment', 'Neutral')
            updated_item['sentiment_score'] = sentiment_info.get('score', 0.5) # Use get for score
            if "error" in sentiment_info: # Log if there was an error for this specific item
                updated_item['notes'] = f"{updated_item.get('notes', '')} API_Error: {sentiment_info['error']}".strip()

                # The timestamp in the original item is from Reddit (item creation)
                # If you want to add a processing timestamp:
                # updated_item['processing_timestamp'] = current_time

                all_results_with_sentiment.append(updated_item)
                print(f"  Processed item {(i + item_idx + 1)}/{total_items}: Sentiment={updated_item['sentiment']}, Score=

            # Respect API rate limits if any are hit, or just be polite
            time.sleep(1) # Small delay between batches

        return all_results_with_sentiment


    def generate_output_filename(keyword: str) -> str:
        """
        Generate a dynamic filename like supersnu(keywordXXX).csv where XXX is a 3-digit incremental ID.
        """
        base_name = f"supersnu({keyword.replace(' ', '_').replace('/', '_')})" # Sanitize keyword for filename
```

```
    output_dir = "output_csvs"
    os.makedirs(output_dir, exist_ok=True)

    existing_files = [f for f in os.listdir(output_dir) if f.startswith(base_name) and f.endswith('.csv')]
    existing_ids = []
    for f_name in existing_files:
        try:
            # Extract number: supersnu(keyword)001.csv → 001
            num_str = f_name[len(base_name):-4]
            if num_str: # Ensure there's something to parse
                existing_ids.append(int(num_str))
        except ValueError:
            # Handle cases where the format might be different or no number exists
            pass # print(f"Could not parse ID from filename: {f_name}")

    next_id = max(existing_ids) + 1 if existing_ids else 1
    return os.path.join(output_dir, f"{base_name}{str(next_id).zfill(3)}.csv")


def save_to_csv(data: List[Dict], filename: str):
    """
    Save the processed data to a CSV file with specified columns.
    """
    if not data:
        print("No data to save.")
        return

    # Columns based on your provided image and original script
    fieldnames = [
        'id', 'topic', 'text', 'sentiment', 'sentiment_score',
        'timestamp', # This is the Reddit item's creation timestamp
        'source', 'notes', 'username', 'upvotes', 'type'
    ]

    print(f"Attempting to save {len(data)} rows to {filename}...")
    try:
        with open(filename, mode='w', encoding='utf-8', newline='') as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames, extrasaction='ignore') # extrasaction='ignore' will skip f
            writer.writeheader()
            for row_num, row_data in enumerate(data):
                # Ensure all fields exist, provide defaults if not
                # This is more robust if some items are missing expected keys
                row_to_write = {field: row_data.get(field) for field in fieldnames}
                writer.writerow(row_to_write)
        print(f"Successfully saved data to {filename}")
    except IOError as e:
        print(f"Error saving CSV file: {e}")
    except Exception as e:
        print(f"An unexpected error occurred during CSV saving: {e}")


def main():
    if len(sys.argv) < 2:
        print("Usage: python3 script_name.py \"KEYWORD\" [limit]")
        sys.exit(1)

    keyword = sys.argv[1]
    limit = 500
    if len(sys.argv) > 2:
        try:
```

```
            limit = int(sys.argv[2])
            if limit <= 0:
                print("Limit must be a positive integer.")
                sys.exit(1)
        except ValueError:
            print("Invalid limit provided. Must be an integer.")
            sys.exit(1)

    print(f"Fetching up to {limit} Reddit data for keyword: {keyword}")

    gemini_api_key = get_env_variable(ENV_GEMINI_API_KEY)

    reddit_instance = authenticate_reddit()
    reddit_data = fetch_reddit_data(reddit_instance, keyword, limit=limit)

    if not reddit_data:
        print("No Reddit data fetched. Exiting.")
        sys.exit(0)

    print(f"Fetched {len(reddit_data)} posts/comments from Reddit.")

    print("Analyzing sentiment with Gemini API in batches...")
    analyzed_data = batch_process_sentiment_with_gemini(reddit_data, gemini_api_key)

    if not analyzed_data:
        print("No data was analyzed. Exiting.")
        sys.exit(0)

    output_file = generate_output_filename(keyword)
    save_to_csv(analyzed_data, output_file)
    # print(f"Sentiment analysis results saved to {output_file}") # Already printed in save_to_csv on success

if __name__ == "__main__":
    # For Gemini prompt debugging, especially JSON extraction.
    # Needs to be imported if you uncomment the regex usage.
    import re
    main()
```