Tools in Data Science - Project 1

Summary

- **Data Scraping**: Data was scraped from the GitHub API using a personal access token (PAT) to authenticate requests. Targeted users in Boston with over 100 followers, utilizing pagination to retrieve multiple pages and up to 500 repositories per user, collecting profile details and repository data. Error handling with retry logic ensured reliable data collection.

- **Interesting Fact from the Data Analysis**: GitHub was founded on February 8, 2008. User "evan," who signed up on February 13, 2008, was the first from Boston, but not the first to create a repository. That honor goes to "joshuaclayton," the fifth to join, who published the first repository, "joshuaclayton/jdclayton," on March 15, 2008.

- **Actionable Recommendation for Developers**: Given that a significant proportion of repositories utilize wikis (85.8%) and project features (97.9%), developers should ensure their repositories have comprehensive documentation. Clear, detailed documentation helps users understand how to use the project and encourages contributions from others.

Project Overview

For my TDS Project 1, I was tasked with scraping and analyzing data from GitHub users in Boston who have more than 100 followers. The objective was to gather insights about these influential users and their repositories to identify trends in programming languages, user engagement, and repository characteristics. I accomplished this using the GitHub API for data retrieval, Python for scripting and data manipulation, and the Pandas library for data organization. The final deliverables for the project included two CSV files (users.csv and repositories.csv), a README document, and the code used for data collection and cleaning.

Steps Involved in the Project

1. **Data Scraping**:
   Utilized the GitHub API with a personal access token to retrieve data on Boston users who had more than 100 followers. Collected comprehensive user details and repository information.

2. **Data Cleaning**:
   Cleaned the collected data to ensure consistency and accuracy. This included trimming whitespace, normalizing text fields such as company names, and formatting data according to the specifications in the problem statement.

3. **Data Analysis**:
   Analyzed the cleaned data to address the project assignment questions and provide specific answers based on the findings. Additionally, I observed some interesting insights.

4. **Documentation, Reporting, and Upload**
   Compiled the findings and insights into a README document that summarized the data scraping process, key findings, and actionable recommendations based on the analysis. Prepared all project files, including the Python scripts for scraping and cleaning, the CSV files, and the README, and uploaded them to a GitHub repository to ensure all components were accessible for review and further development.

Step 1: Data Scraping Process

The data was collected using the GitHub API, which followed a structured approach to ensure authentication, error handling, and efficient data retrieval. Here's a breakdown of what was done:

- **API Authentication**:
  - The script utilized a personal access token for authentication, which was included in the headers of each API request. This token ensured secure access to the GitHub API while respecting the set rate limits.
- **Rate Limit Monitoring**:
  - An initial authentication check was performed to retrieve the rate limit status. This involved checking how many requests remained and when the limit would reset. The reset time was adjusted to Indian Standard Time (IST) for convenience.
- **Data Retrieval**:
  - The main focus of data retrieval was on users located in Boston with more than 100 followers. The script employed a search endpoint to find these users, with pagination support to manage the number of results returned in each request. For each user found, detailed information was fetched from the user endpoint, which included their login, name, company, location, email, hireable status, bio, public repository count, followers, following count, and account creation date.
- **Repository Collection**:
  - For each user, the script fetched their public repositories, up to a maximum of 500. This was achieved by sending requests to the repositories endpoint, with pagination handling multiple pages of results.
- **Error Handling**:
  - The script implemented retry logic for API requests that might fail due to transient issues. If an error occurred, it logged the user or repository that could not be fetched for further investigation.
- **Data Storage**:
  - All collected data was stored in two separate lists, which were then converted into Pandas DataFrames and saved as CSV files (users.csv and repositories.csv). An error log was also created to document any failures during the scraping process.

Here is the code block

```
import requests
import pandas as pd
import time
from datetime import datetime, timezone, timedelta

# Set up headers for GitHub API authentication
access_token = "SECRET KEY"
headers = {
    "Authorization": f"Bearer {access_token}"
}

# Function to check GitHub API authentication and rate limit reset time in IST
def check_auth():
    try:
```

```
        auth_check = requests.get("https://api.github.com/rate_limit", headers=headers,
timeout=10)
        if auth_check.status_code == 200:
            rate_data = auth_check.json()["rate"]
            remaining_requests = rate_data['remaining']
            reset_timestamp = rate_data['reset']
            reset_time_utc = datetime.fromtimestamp(reset_timestamp, timezone.utc)
            reset_time_ist = reset_time_utc + timedelta(hours=5, minutes=30)
            print(f"Authenticated successfully. Requests remaining: {remaining_requests},
resets at: {reset_time_ist.strftime('%Y-%m-%d %H:%M:%S')} IST")
            return True
        else:
            print("Authentication failed. Check your access token.")
            return False
    except requests.exceptions.RequestException as e:
        print("Authentication check failed due to network error:", e)
        return False

# Helper function to make a request with retries
def make_request(url, headers, params=None, retries=3, timeout=10):
    for attempt in range(retries):
        try:
            response = requests.get(url, headers=headers, params=params, timeout=timeout)
            if response.status_code == 200:
                return response
            else:
                print(f"Request failed with status {response.status_code}. Attempt {attempt +
1} of {retries}.")
        except requests.exceptions.RequestException as e:
            print(f"Request error on attempt {attempt + 1} of {retries}: {e}")
        time.sleep(2)  # wait before retrying
    return None

# Proceed only if authenticated
if check_auth():
    search_url = "https://api.github.com/search/users"
    params = {
        "q": "location:Boston followers:>100",
        "per_page": 100,
        "page": 1
    }

    all_users_data = []         # To store user data for users.csv
    all_repositories_data = []  # To store repository data for repositories.csv
    error_log_data = []         # To log users whose data could not be fetched
    users_processed = 0         # Counter to track number of users processed

    while True:
```

```
        # Fetch the current page of users
        response = make_request(search_url, headers, params=params)
        if response and response.status_code == 200:
            data = response.json()
            users = data.get("items", [])
            print(f"Found {len(users)} users on page {params['page']}")

            if not users:
                break  # No more users to process

            for user in users:
                # Fetch full user details
                user_detail_url = f"https://api.github.com/users/{user['login']}"
                user_detail_response = make_request(user_detail_url, headers)

                if user_detail_response and user_detail_response.status_code == 200:
                    user_detail = user_detail_response.json()

                    # Collect user details for users.csv without any cleaning
                    user_data = {
                        "login": user_detail.get("login", ""),
                        "name": user_detail.get("name", ""),
                        "company": user_detail.get("company", ""),
                        "location": user_detail.get("location", ""),
                        "email": user_detail.get("email", ""),
                        "hireable": user_detail.get("hireable", ""),
                        "bio": user_detail.get("bio", ""),
                        "public_repos": user_detail.get("public_repos", 0),
                        "followers": user_detail.get("followers", 0),
                        "following": user_detail.get("following", 0),
                        "created_at": user_detail.get("created_at", "")
                    }
                    all_users_data.append(user_data)

                    # Fetch up to 500 repositories with pagination and track total for each
user
                    repos_fetched = 0
                    repo_page = 1
                    while repos_fetched < 500:
                        repos_url = f"https://api.github.com/users/{user['login']}/repos"
                        repo_response = make_request(repos_url, headers, params={"per_page":
100, "page": repo_page})

                        if repo_response and repo_response.status_code == 200:
                            repositories = repo_response.json()
                            if not repositories:
                                break  # No more repositories to fetch
```

```
                            for repo in repositories:
                                repo_data = {
                                    "login": user_detail.get("login", ""),
                                    "full_name": repo.get("full_name", ""),
                                    "created_at": repo.get("created_at", ""),
                                    "stargazers_count": repo.get("stargazers_count", 0),
                                    "watchers_count": repo.get("watchers_count", 0),
                                    "language": repo.get("language", ""),
                                    "has_projects": repo.get("has_projects", ""),
                                    "has_wiki": repo.get("has_wiki", ""),
                                    "license_name": repo.get("license", {}).get("name", "")
if repo.get("license") else ""
                                }
                                all_repositories_data.append(repo_data)
                                repos_fetched += 1

                            repo_page += 1  # Move to the next page of repositories

                            if repos_fetched >= 500:
                                break  # Stop fetching if 500 repositories are reached
                        else:
                            error_log_data.append({"login": user_detail.get("login", ""),
"error": "Failed to fetch repositories"})
                            break

                    print(f"Total repositories fetched for {user_detail.get('login', '')}:
{repos_fetched}")

                else:
                    error_log_data.append({"login": user.get("login", ""), "error": "Failed
to fetch user details"})

                users_processed += 1
                print(f"Total users processed: {users_processed}")

            # Check if there are more pages to fetch
            if 'next' in response.links:
                params["page"] += 1
            else:
                break
        else:
            print("Failed to fetch users data or no more users to fetch.")
            break

    # Create DataFrames and save to CSV without any data cleaning
    users_df = pd.DataFrame(all_users_data)
    repos_df = pd.DataFrame(all_repositories_data)
    users_df.to_csv("users.csv", index=False)
```

```
    repos_df.to_csv("repositories.csv", index=False)
    print("Data saved to users.csv and repositories.csv.")

    # Save error log if any
    if error_log_data:
        error_log_df = pd.DataFrame(error_log_data)
        error_log_df.to_csv("error_log.csv", index=False)
        print("Error log saved to error_log.csv.")
else:
    print("Authentication check failed. Exiting.")
```

Step 2: Data Cleaning

- **Data Cleaning for users.csv**
  - Company Field:
    - Removed any leading whitespace and the '@' character from the beginning of each company name.
    - Converted all company names to uppercase to ensure uniformity
  - Hireable Field:
    - Formatted the 'hireable' field to store only 'true', 'false', or an empty string if the value was null, providing a consistent format for boolean values.
  - Data Saving:
    - Saved the cleaned data to 'cleaned_users2.csv' and displayed a sample of the cleaned data to confirm the changes were applied correctly.

```
# Clean up the 'company' field
users_df['company'] = users_df['company'].str.strip()          # Remove whitespace
users_df['company'] = users_df['company'].str.lstrip('@')      # Strip leading '@'
users_df['company'] = users_df['company'].str.upper()          # Convert to uppercase

# Format the 'hireable' column specifically as 'true', 'false', or empty string if null
users_df['hireable'] = users_df['hireable'].apply(lambda x: 'true' if x is True else
('false' if x is False else ''))

# Save the cleaned file to confirm changes
users_df.to_csv('cleaned_users2.csv', index=False)

# Display a sample of the cleaned data to verify
users_df.head()
```

- **Data Cleaning for repositories.csv**
  - Boolean Fields:
    - Formatted the 'has_projects' and 'has_wiki' fields to 'true', 'false', or leave as an empty string if null. This standardized the presentation of boolean fields across the dataset.

- Data Saving:
  - Saved the cleaned repository data to 'cleaned_repositories2.csv' and displayed a sample to ensure the cleaning was executed as expected.

```
#  Format boolean fields to be 'true', 'false', or empty string for nulls
repositories_df['has_projects'] = repositories_df['has_projects'].apply(lambda x:
'true' if x is True else ('false' if x is False else ''))
repositories_df['has_wiki'] = repositories_df['has_wiki'].apply(lambda x: 'true' if x
is True else ('false' if x is False else ''))

# Save the cleaned file to confirm changes
repositories_df.to_csv('cleaned_repositories2.csv', index=False)

# Display a sample to confirm the output
repositories_df.head()
```

Step 3: Data Analysis

Analyzed the cleaned data to address the project assignment questions and provide specific answers based on the findings. Additionally, I observed some interesting insights.

I have attached the Google Colab code as a separate PDF, which includes both the data cleaning process and my solution code to all 16 questions. Please refer to it for full details.

Some insights from the data analysis:

1. **Top 5 Users by Followers:** The most-followed user, "brianyu28," has over 13,200 followers, followed by "PatrickAlphaC" with 9,670 and others close behind, showing strong influence.

2. **Top 5 Users by Public Repositories:** "JLLeitschuh" leads with an impressive 1,534 repositories, followed by users like "bahmutov" with 1,245. High repository counts hint at consistent productivity and contribution.

3. **Most Common Companies:** The top employers include major names like Northeastern University (16 users), Google (12), and Microsoft (10), indicating strong tech community support.

4. **Most Common Locations:** Most users are based in or around Boston, MA, with variations like "Boston, MA, USA" and "Boston, USA" being frequent among 267 recorded users.

5. **Top 5 Repositories by Stars:** The repository "rapid7/metasploit-framework" has an astounding 34,091 stars, making it a top project in Ruby and showcasing a global user interest in security tools.

6. **Most Common Programming Languages:** JavaScript, Python, and HTML are dominant, with JavaScript appearing in over 7,700 repositories, highlighting trends in development preferences.

7. **Repositories with Projects Enabled:** 97.9% of repositories have project features enabled, showing a widespread interest in organizing work within GitHub itself.

8. **Repositories with Wiki Enabled:** 85.8% of repositories use GitHub's wiki, indicating a strong inclination toward documentation and knowledge-sharing.

9. **Top 5 Licenses Used:** The MIT License is the most popular (9,748 repos), followed by Apache and GPL licenses, reflecting an open-source focus among developers.

10. **Oldest GitHub Account:** User "evan" holds the oldest account, created on February 13, 2008, showing early adoption and experience in the GitHub ecosystem.

Step 3: Documentation, Reporting and Upload

- **README Documentation**:
  Created a README document summarizing the entire project, including the data scraping process, data cleaning, and key insights from the analysis.

- **Project Files**: Organized all relevant project files within the repository, including:
  - The cleaned CSV files (users.csv and repositories.csv) for easy access to the processed data.
  - This README document containing the project overview, outline, and the Python code for data scraping and cleaning.
  - A Google Colab notebook (saved as a PDF) containing the complete code solutions for data cleaning and answers to all 16 project questions.

- **Final Upload to GitHub**: Uploaded the README, CSV files, and the PDF of the Google Colab code to a GitHub repository.