Mondelli Federico 20044357

Prerequisiti:

- **Installazione librerie necesssarie:**
  - requests: *pip install requests*
  - pandas: *pip install pandas*
  - pm4py: *pip install pm4py*
  - matplotlib: *pip install matplotlib*
- **Aggiornamento ('trending.csv'):**
  - Aggiornare le repository virali.

Fase 1 – Raccolta dei Dati da GitHub Archive (data_collection.py)

Questa prima fase si suddivide in **tre sottofasi principali**, ognuna delle quali ha un ruolo cruciale nell'acquisizione e nella preparazione dei dati per l'analisi.
Di seguito viene descritta la fase nel dettaglio per comprendere meglio il flusso del processo:

1. Download dei file da GitHub Archive

GitHub Archive pubblica file compressi '.json.gz'.
Ogni file rappresenta un'ora di eventi denominato attraverso un formato: 'YYYY-MM-DD-HH.json.gz', dove HH indica l'ora specifica (dalle 00 alle 23 in UTC).

Funzionamento:

- **Determinazione dell'intervallo temporale**: il codice si basa sull'orario di esecuzione per determinare quale intervallo temporale scaricare.
  GitHub aggiorna la sezione "Trending" ogni giorno alle 11:00 UTC, quindi:
  - Se l'orario attuale è dopo le 11:03 UTC:
    il codice scarica i file relativi all'intervallo dalle 11:00 UTC del giorno precedente fino alle 11:00 UTC del giorno corrente.
  - Se l'orario è prima delle 11:03 UTC:
    GitHub potrebbe non aver ancora aggiornato la lista "Trending", quindi il codice scarica i file dell'intervallo precedente, ovvero dalle 11:00 UTC di due giorni prima fino alle 11:00 UTC del giorno precedente.
- **Evita il download ridondante**: prima di scaricare nuovi file, il codice verifica se sono già presenti nella directory di destinazione (data/github_data). Se il file esiste già, viene ignorato, evitando quindi il download ripetuto di file già disponibili.
- **Salvataggio dei file (*'data/github_data'*)**: i file scaricati vengono memorizzati nella directory menzionata (che viene creata automaticamente se non esiste già). Ogni file contiene gli eventi relativi a un'ora di attività su GitHub, e viene scaricato tramite una chiamata HTTP a https://data.gharchive.org/ utilizzando il nome del file.
- **Rimozione file obsoleti**: al termine del processo, eventuali file scaricati (in esecuzioni precedenti) non più utili all'elaborazione attuale non più necessario, vengono rimossi per mantenere la directory ordinata e ridurre lo spazio occupato inutilmente (essendo che i file sono di grandi dimensioni).
- **Monitoraggio delle prestazioni**: momentaneamente è presente anche il tempo impiegato per scaricare i file. Questo dato può essere utile per future ottimizzazioni del processo, soprattutto se i tempi di download si rivelano troppo lunghi (magari scaricare parallelamente i file...).

2. Parsing dei file JSON e conversione in CSV

Una volta scaricati e puliti i file, la fase successiva consiste nell'elaborazione dei file .json.gz per convertirli in un formato più facilmente manipolabile, ovvero il formato CSV.

Funzionamento:

- **Lettura dei file JSON**: il codice scorre tutti i file .json.gz presenti nella cartella data/github_data. Ogni file viene letto e decompresso utilizzando la libreria gzip. Ogni riga di ciascun file JSON rappresenta un evento GitHub.
- **Evita il parsing ridondante**: prima di processare ogni file, il codice verifica se sono già presenti nella directory di destinazione (data/github_csv). Se il file esiste già, viene ignorato, evitando quindi il parsing ripetuto di file già convertiti in esecuzioni precedenti.
- **Estrazione dei dati rilevanti**: Dato che i file JSON contengono molte informazioni, il codice estrae solo i dati di interesse, che corrispondono agli eventi di GitHub. Per ogni evento vengono estratti i seguenti campi:
    - Tipi di evento: 'type_event'
    - ID dell'attore: 'id_actor'
    - URL dell'attore: 'url_actor'
    - ID repository: 'name_repo'
    - Azione del payload dell'evento: 'action_payload'
    - Timestamp di creazione dell'evento: 'created_at'
- **Salvataggio dei file in formato CSV ('data_github_csv')**: dopo l'estrazione, i dati vengono scritti in un file CSV con le colonne definite in CSV_HEADERS. Questo passaggio rende i dati più facili da manipolare e analizzare nelle fasi successive.
- **Gestione degli errori**: Vengono gestiti vari tipi di errori, come ad esempio la presenza di file corrotti o malformati, o la mancanza di alcune chiavi nei dati JSON. Se si verificano questi errori, il codice segnala il problema e continua con gli altri file (saltando il file problematico).
- **Rimozione file obsoleti**: al termine del processo, eventuali file .csv di che contengono eventi di intervalli di tempo precedenti (es: aggiornamento eventi di due giorni fa…) non più utili all'elaborazione attuale non più necessari, vengono rimossi per mantenere la directory ordinata e ridurre lo spazio occupato inutilmente (essendo che i file sono di grandi dimensioni).

3. Pulizia e normalizzazione dei file CSV

Una volta convertiti i file .json.gz → .csv, è necessario effettuare un'ulteriore elaborazione per ripulire e uniformare i dati. Questa sottofase si occupa di preparare i CSV per la generazione dei modelli di processo, rimuovendo anomalie, calcolando metriche temporali e rendendo più chiari alcuni nomi di eventi

Funzionamento:

- **Lettura e verifica:** ogni file viene letto e controllato: se manca colonna 'type_event', viene scartato.
- **Pulizia dei file:** per ogni file valido:
    - Viene convertito il campo 'created_at' in formato data/ora leggibile
    - Vengono ordinato cronologicamente gli eventi per repository
    - Viene calcolato il tempo tra un evento e il successivo (per ogni repository: 'time_diff'))
    - Alcuni eventi vengono rinominati: *'WatchEvent'* → *StarEvent* (se l'azione nel payload = 'started')
- **Evita la pulizia ridondante**: prima di processare ogni file, il codice verifica se sono già presenti nella directory di destinazione (data/github_cleaned_csv). Se il file esiste già, viene ignorato, evitando quindi la pulizia per file già processati in esecuzioni precedenti.

- **Salvataggio dei file (‘data/github_cleaned_csv’):** i file puliti e normalizzati vengono salvati nella directory menzionata
- **Rimozione file obsoleti**: al termine del processo, eventuali file .csv di che contengono eventi di intervalli di tempo precedenti (es: aggiornamento eventi di due giorni fa…) non più utili all’elaborazione attuale non più necessari, vengono rimossi per mantenere la directory ordinata e ridurre lo spazio occupato inutilmente (essendo che i file sono di grandi dimensioni).

Fase 2 – Process Mining (Generazione dei modelli di processo e calcolo delle feature) (process_mining.py)

Questa fase ha l’obiettivo di trasformare i dati raccolti e puliti in rappresentazioni grafiche e numeriche utili per il Machine Learning (fase 3).
Il processo è composto da diverse sottofasi che includono l’estrazione dei sottoinsiemi di eventi per repository, la generazione dei modelli di processo (DFG) e il calcolo delle caratteristiche di ogni repository. Questa seconda fase viene eseguita anche se il suo risultato finale è già stato generato in esecuzioni precedenti, per potervi mostrare meglio il flusso del processo durante l’esecuzione del programma. Di seguito viene descritto il funzionamento di questa fase nel dettaglio:

1. Preparazione dei dati

Funzionamento:

- **Caricamento CSV puliti:** vengono letti tutti i file presenti nella directory *‘data/github_cleaned_csv’* e uniti in un unico DataFrame che rappresenta tutti gli eventi di tutte le 24 ore scaricate.
- **Caricamento repository virali (‘data/trending/trending.csv’):** viene caricato il file ‘*trending.csv*’, contenente la lista aggiornata delle repository considerate virali.
  E’ un **requisito indispensabile** che questo file venga aggiornato manualmente, prima dell’esecuzione del programma, visitando questo link: ‘https://github.com/trending’ e bisogna copiare manualmente, sotto la colonna ‘repository’ del file, tutti i nomi delle repo attualmente trending (Spoken Language: Any, Language: Any, Data Range: **Today**), altrimenti se il file non è presente, è vuoto o malformato (es: senza colonna ‘repository’) il programma viene interrotto. Questo perché senza questo file corretto, non ha la possibilità di poter addestrare il modello essendo che non ha “etichette” per distinguere le repo virali da quelle non virali.
- **Selezione repository non virali:** per creare un dataset bilanciato, viene costruita una lista casuale di repository **non virali** scegliendole tra quelle presenti nel DataFrame, **escludendo** quelle già classificate come virali: ne sceglie un numero N (N = numero di repository virali) al fine di poter avere un addestramento bilanciato.

2. Generazione dei modelli di processo (DFG)

- **Estrazione del sottoinsieme eventi:** vengono selezionati solo gli eventi relativi alla repository in esame (N repo virali + N repo non virali scelte in automatico).
- **Creazione del log PM4Py:** il sottoinsieme viene trasformato in un log compatibile con PM4Py

- **Costruzione DFG:** viene generato il Directly-Follows Graph (DFG), che rappresenta la sequenza diretta tra le attività (tipi di eventi) eseguite sulla repository.
- **Salvataggio dei grafi ('data/process_model'):** il DFG vengono salvati come '*.png'* nella cartella menzionata, con nome *'<repo>.png'*.
  Se il file esiste già, viene sovrascritto per i motivi descritti nell'introduzione della Fase 2 (Process Mining) e inoltre attualmente non ho bisogno di memorizzarmi a lungo termine queste immagini, servono solo come resoconto dopo il termine dell'esecuzione del programma.

3. Estrazione delle feature (caratteristiche)

Per ogni DFG generato vengono calcolate le seguenti feature numeriche, utili per l'addestramento dei modelli predittivi:

- num_nodi: numero di nodi (tipi di eventi unici) nel DFG.
- num_archi: numero totale di archi (transizioni tra eventi).
- densità_dfg: densità del grafo, calcolata come rapporto tra archi esistenti e archi teoricamente possibili (n * (n-1)).
- percentuale_StarEvent: percentuale di eventi StarEvent (originati da WatchEvent con payload.action == "started") rispetto al totale degli eventi nella repository.
- tempo_medio_eventi: tempo medio (in secondi) tra un evento e il successivo, calcolato solo all'interno della stessa repository.

4. Salvataggio dei risultati ('data/ process_model_features.csv')

- **Scrittura delle feature:** tutte le feature vengono raccolte in un DataFrame, dove ogni riga rappresenta una repository e ogni colonna una "metrica". Viene aggiunta anche una colonna target con valore 1 se la repository è virale, 0 altrimenti (questo viene fatto grazie al file trending.csv che abbiamo aggiornato manualmente prima dell'esecuzione dell'intero programma).
  Questo è un punto cruciale su cui discutere meglio: attualmente ad ogni aggiornamento disponibile, questo file viene aggiornato con le nuove repo selezionate (virali+non virali) + le relative feature descritte sopra. Quindi attualmente sto preparando i dati per un addestramento nel fare previsioni se una nuova repo "può andare virale" in base agli eventi delle sue ultime 24 ore (senza tracciare una storia della repo, anche perché aggiornamenti eseguiti con diversi giorni di distanza, non avrebbe la storia completa della repo, ma ce l'avrebbe con buchi di 24 ore. Infatti salvo nel file *'process_model_features.csv'* le repo aggiungendo al nome della repo la data del giorno in cui aggiungo questo record al file. Questa modifica al nome penso possa essere necessario per far capire al modello che ogni repository sia univoca...
  Altrimenti dovremmo discutere per altre tipologie su come impostare la preparazione dei dati per l'addestramento.