



UNIVERSITÀ
DELLA
CALABRIA

DIPARTIMENTO DI INGEGNERIA
INFORMATICA, MODELLISTICA,
ELETTRONICA E SISTEMISTICA

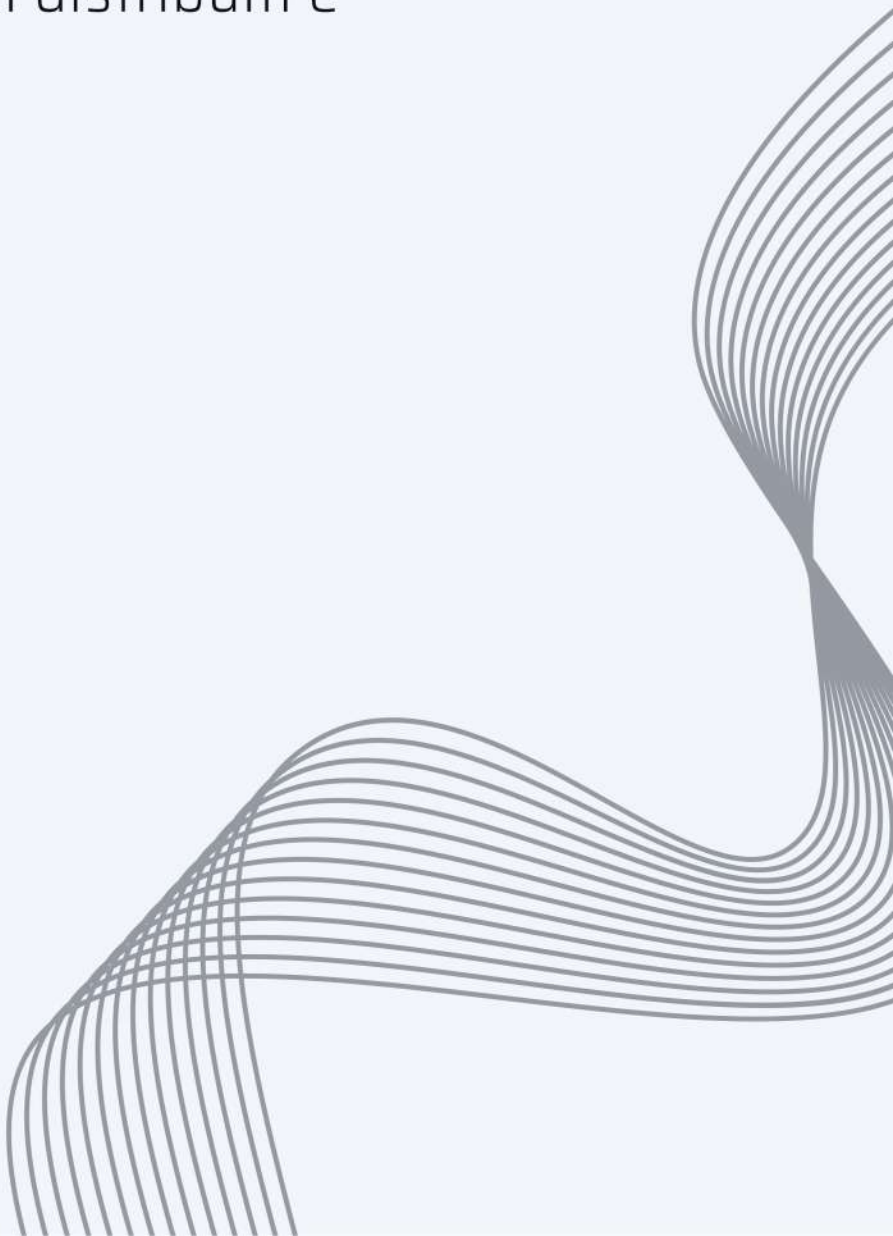
DIMES

FREQUENT ITEMSET MINING DOCKERIZED APP

Progetto di Sistemi distribuiti e
cloud computing

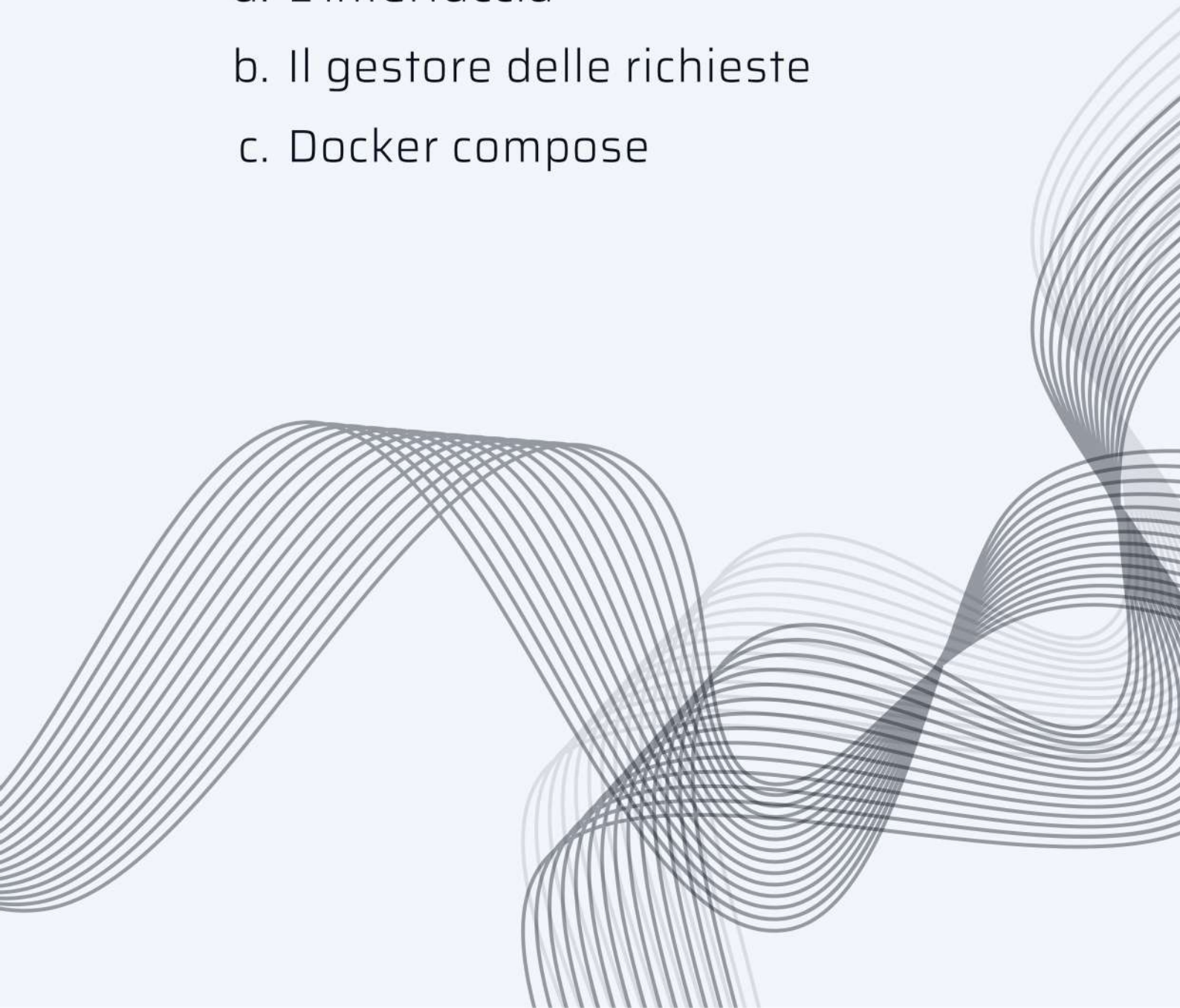
A.A. 2022/2023

Studentessa
GAIA ASSUNTA
BERTOLINO
MAT. 242590



INDICE

1. Introduzione al progetto
2. La tecnologia Docker
3. Gli algoritmi di frequent mining
4. L'implementazione
 - a. L'interfaccia
 - b. Il gestore delle richieste
 - c. Docker compose



1. INTRODUZIONE AL PROGETTO

Il progetto assegnato ha richiesto l'implementazione di un sistema distribuito per l'esecuzione di **algoritmi di frequent itemset mining** su dataset forniti in ingresso dall'utente.

Il sistema realizzato consente l'esecuzione di più job in parallelo grazie al processo di **containerizzazione** separata delle funzionalità della web app e notifica all'utente in maniera **asincrona** tramite mail l'esito di ciascuna operazione, con la possibilità di scaricare i risultati dell'analisi.

Il progetto è stato integrato con un **frontend** per l'inserimento dei dati richiesti e con un **database** per il mantenimento dello storico delle richieste effettuate dall'avvio dell'applicazione.

La tecnologia richiesta per l'implementazione fa riferimento ai tools messi a disposizione da Docker Engine.

Nell'ambito del progetto, si è scelti di ricorrere ai seguenti strumenti:

- **Visual Studio Code** per la programmazione del frontend e del backend, opportunamente integrato con l'estensione ufficiale di Docker
- **Docker Engine** per la containerizzazione della web app
- **Docker Desktop** per la gestione delle immagini e dei container
- **Postman** per operazioni di testing
- **DB Browser for SQLite** per visualizzare il contenuto del database locale creato per mantenere lo storico delle analisi richieste
- **Google Drive** per la condivisione del progetto

2. LA TECNOLOGIA DOCKER

Docker è un software libero progettato per eseguire applicazioni in ambienti isolabili e facilmente distribuibili, denominato **container Linux** o più semplicemente **container**, che si basa su un'architettura client-server.

La **containerizzazione** è un processo in cui tutte le informazioni necessarie al funzionamento di un'app (come dipendenze e configurazioni) vengono raggruppate insieme al codice sorgente dell'app in un oggetto definito **immagine**.

Tale processo si differenzia dalla funzionalità rappresentata dalla **macchina virtuale** in quanto essa opera invece a livello hardware, astraendo dunque le risorse messe a disposizione dal terminale. Ne consegue che la **virtualizzazione** permette di avere su un singolo computer più sistemi operativi, agendo tramite l'uso di software come hypervisor e virtual machine monitor, i quali svolgono funzioni di partizionamento delle risorse fra macchine, di isolamento delle risorse, dell'incapsulamento dei dati e dell'indipendenza fra hardware e software. Esempi di virtualizzazione sono VMWare e Virtualbox. Dunque, ogni macchina virtuale esegue un sistema operativo a se stante e spesso i file ad essa afferenti presentano dimensioni notevoli a causa delle informazioni necessarie alla sua esecuzione.

La creazione e l'esecuzione di una immagine, invece, prescinde dal sistema operativo; infatti, ciascuna immagine può essere istanziata indipendentemente dall'OS presente sul terminale in quanto viene eseguita tramite il Docker Engine.

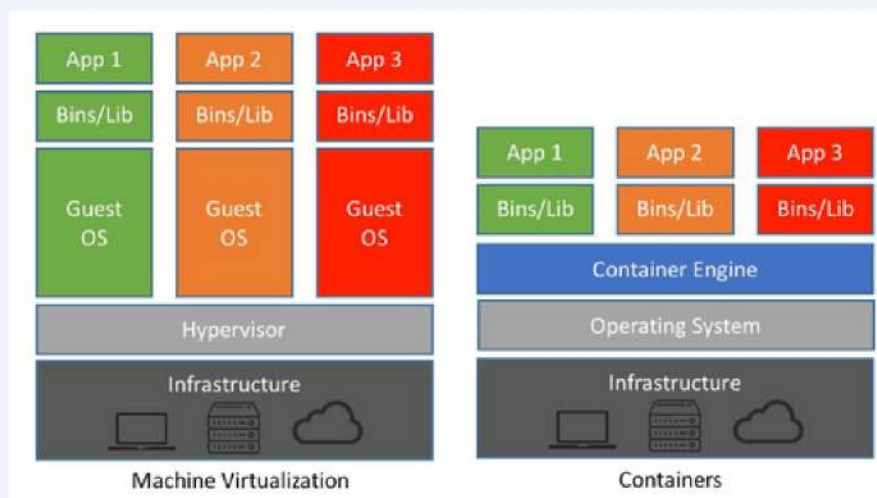


Fig. 1: Confronto della gerarchia di funzionamento fra macchine virtuali e containers

Docker inizialmente faceva uso esclusivo di LXC (acronimo per Linux Containers) il quale è un ambiente di virtualizzazione in grado di eseguire diversi ambienti Linux isolati tra loro su un terminale dotato di kernel Linux. LXC, mediante l'uso della funzionalità **cgroups** fornita dal kernel Linux, è in grado di considerare, limitare e isolare l'utilizzo delle risorse da parte di un gruppo di processi.

Successivamente, Docker ha iniziato a basarsi sulla libreria **libcontainer**, scritta in Go e in grado di gestire l'intera vita di un container. Tramite la sua evoluzione, ora Docker è in grado di fornire un servizio che astrae dal sistema operativo.

La containerizzazione consiste dunque nella creazione di una immagine, un oggetto contenente tutte le dipendenze e le configurazioni necessarie a replicare l'ambiente di esecuzione richiesto da un determinato software. Ad una immagine è possibile applicare dei cambiamenti (producendo dunque delle immagini intermedie) che vengono rappresentati tramite dei **layer**. La layerizzazione, inoltre, semplifica il processo di building dell'immagine.

Il cosiddetto container è l'istanza di una immagine quindi una immagine in esecuzione tramite un client Docker. L'immagine viene identificata da un **tag** che permette di poterla richiamare ed eseguire.

L'architettura di Docker è di tipo **client-server**:

- il cosiddetto **Docker Daemon** (dockerd) è il componente principale e si occupa di coordinare gli oggetti docker come immagini e container
- il **Docker Client** è una interfaccia che permette l'interazione con uno o più Docker Daemon, ad esempio per eseguire la build di una immagine
- il **Docker Registry** altro non è che una o più repository per la gestione delle immagini (ad esempio Docker Hub).

In particolare, il Docker Client e il Docker Daemon possono trovarsi sulla stessa macchina o il demone può essere posto in remoto.

In aggiunta, tramite i **volume** è possibile scambiare dati fra container o renderli persistenti in zone del filesystem host gestite da Docker; essi si differenziano dai bind mount che archiviano invece in qualsiasi parte del sistema host e dai mount di tmpfs che invece possono essere archiviati solo nella memoria e non nel filesystem dell'host.

Tramite la funzionalità del **Docker Network** è possibile creare delle reti di comunicazione fra docker. La rete di default è la cosiddetta bridge che può gestire fino ad un massimo di $2^{16} = 65536$ container.

Per la creazione di una immagine è necessaria la presenza di un **Dockerfile** (caratterizzato da domain-specific language (DSL)) ovvero un insieme di istruzioni definite in un linguaggio specifico relativo al dominio Docker.

Tramite il client **Docker Compose**, è invece possibile far funzionare applicazioni composte da più immagini. Esso si realizza tramite un file YAML in grado di definire le interazioni fra container, eventuali reti di networking e porte di esecuzione.

I container possono poi essere orchestrati tramite **software di distribuzione** e gestione come Kubernetes e Docker Swarm che permettono di gestire funzionalità come load balancing e scalabilità.

In conclusione, l'utilizzo di Docker per creare e gestire i container può semplificare la creazione di sistemi distribuiti, permettendo a diverse applicazioni o processi di lavorare in modo autonomo sulla stessa macchina fisica o su diverse macchine virtuali.

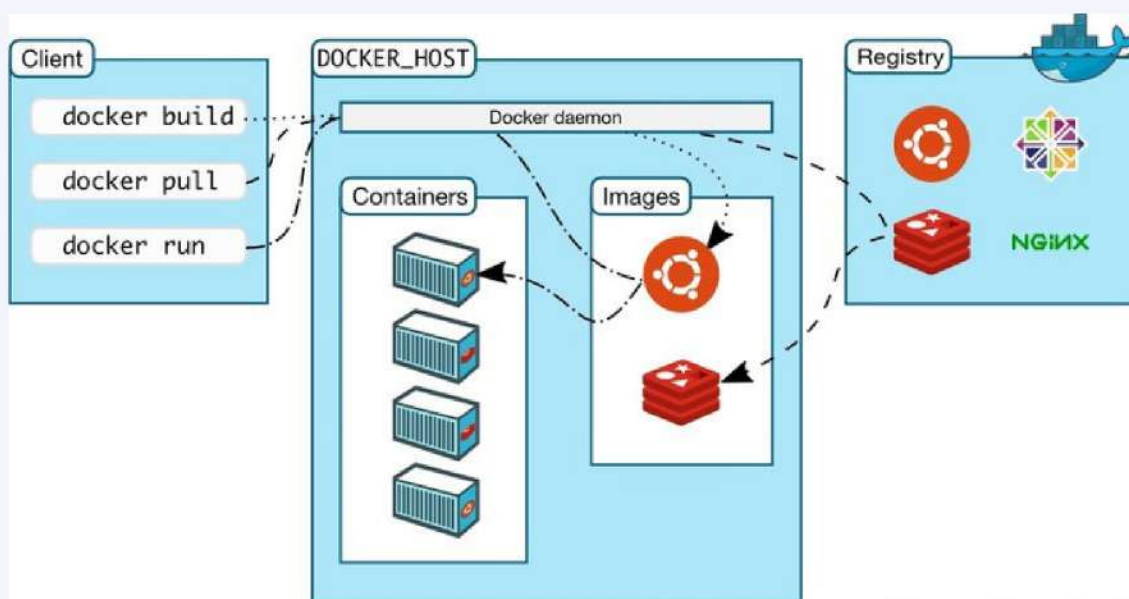


Fig. 2: Architettura dell'environment Docker

3. GLI ALGORITMI DI FREQUENT MINING

Il **Frequent Pattern Mining** è un meccanismo appartenente al campo del Data Mining che consente di identificare, all'interno di un dataset, schemi e associazioni tra i dati che ricorrono con una certa frequenza. Questo genere di informazioni si rivela utile in ambito commerciale, tecnologico e scientifico e si applica per lo più a dati transazionali e sequenziali.

Una delle tecniche più note, ad esempio, è la **Market Basket Analysis (MBA)**, la quale prevede di indicare la presenza (1) o meno (0) dei prodotti nei carrelli della spesa dei clienti, che vengono quindi descritti come sequenze di 1 e 0, dette transazioni. Da un insieme di transazioni, la MBA deduce quali prodotti tendono a essere acquistati insieme, un'informazione di grande utilità per progettare ad esempio strategie di cross-selling in un supermercato.

Nell'ambito di questo progetto sono stati resi disponibili all'utente due algoritmi di estrapolazione delle regole di associazione:

- **L'algoritmo Apriori** : esso è utilizzato per la generazione degli itemset frequenti tramite approssimazioni successive a partire dagli itemset con un solo elemento. In sintesi, il presupposto teorico su cui si basa l'algoritmo parte dalla considerazione che se un insieme di oggetti (itemset) è frequente, allora anche tutti i suoi sottoinsiemi sono frequenti, ma se un itemset non è frequente, allora neanche gli insiemi che lo contengono sono frequenti (principio di anti monotonicità).

```
Apriori ( $T, \epsilon$ )  
   $L_1 \leftarrow \{ \text{large 1-itemsets} \}$   
   $k \leftarrow 2$   
  while  $L_{k-1} \neq \emptyset$   
     $C_k \leftarrow \text{Generate}(L_{k-1})$   
    for transactions  $t \in T$   
       $C_t \leftarrow \text{Subset}(C_k, t)$   
      for candidates  $c \in C_t$   
         $\text{count}[c] \leftarrow \text{count}[c] + 1$   
       $L_k \leftarrow \{ c \in C_k \mid \text{count}[c] \geq \epsilon \}$   
       $k \leftarrow k + 1$   
  return  $\bigcup_k L_k$ 
```

Fig. 3: Pseudocodice dell'algoritmo apriori

- **L'algoritmo FP-Growth:** esso è un miglioramento del metodo Apriori. L'algoritmo prevede che venga generato un pattern frequente senza la necessità della generazione di candidati. L'algoritmo di crescita FP rappresenta infatti il database sotto forma di un albero chiamato albero pattern frequente o albero FP che mantiene l'associazione tra i set di elementi e dove ogni nodo rappresenta un elemento dell'insieme di elementi. Il database viene dunque frammentato utilizzando un elemento frequente; questa parte frammentata viene chiamata "frammento di pattern" e vengono analizzati i set di elementi di questi modelli frammentati. Con questo metodo, dunque, la ricerca di set di elementi frequenti viene ridotta in modo comparativo.

```
Procedure FPgrowth*(T)  
Input: A conditional FP-tree T  
Output: The complete set of all FI's corresponding to T.  
Method:  
1. if T only contains a single branch B  
2.   for each subset Y of the set of items in B  
3.     output itemset  $Y \cup T.base$  with count = smallest count of nodes in Y;  
4. else for each i in T.header do begin  
5.   output  $Y = T.base \cup \{i\}$  with i.count;  
6.   if T.FP-array is defined  
7.     construct a new header table for Y's FP-tree from T.FP-array  
8.   else construct a new header table from T;  
9.   construct Y's conditional FP-tree TY and possibly its FP-array AY;  
10.  if TY ≠ ∅  
11.    call FPgrowth*(TY);  
12. end
```

Fig. 4: Pseudocodice dell'algoritmo FP-Growth

4. L'IMPLEMENTAZIONE

Le funzionalità della web app realizzata risultano essere suddivise in due macroaree di gestione:

- Una interfaccia in grado di ricevere i parametri e il dataset da analizzare con funzionalità di salvataggio in database locale dello storico delle richieste. Di seguito tale area sarà definita col termine **interfaccia**
- Il software di applicazione dell'algoritmo scelto al dataset e di invio dei risultati tramite mail. Di seguito tale area sarà definita col termine **gestore della richiesta**

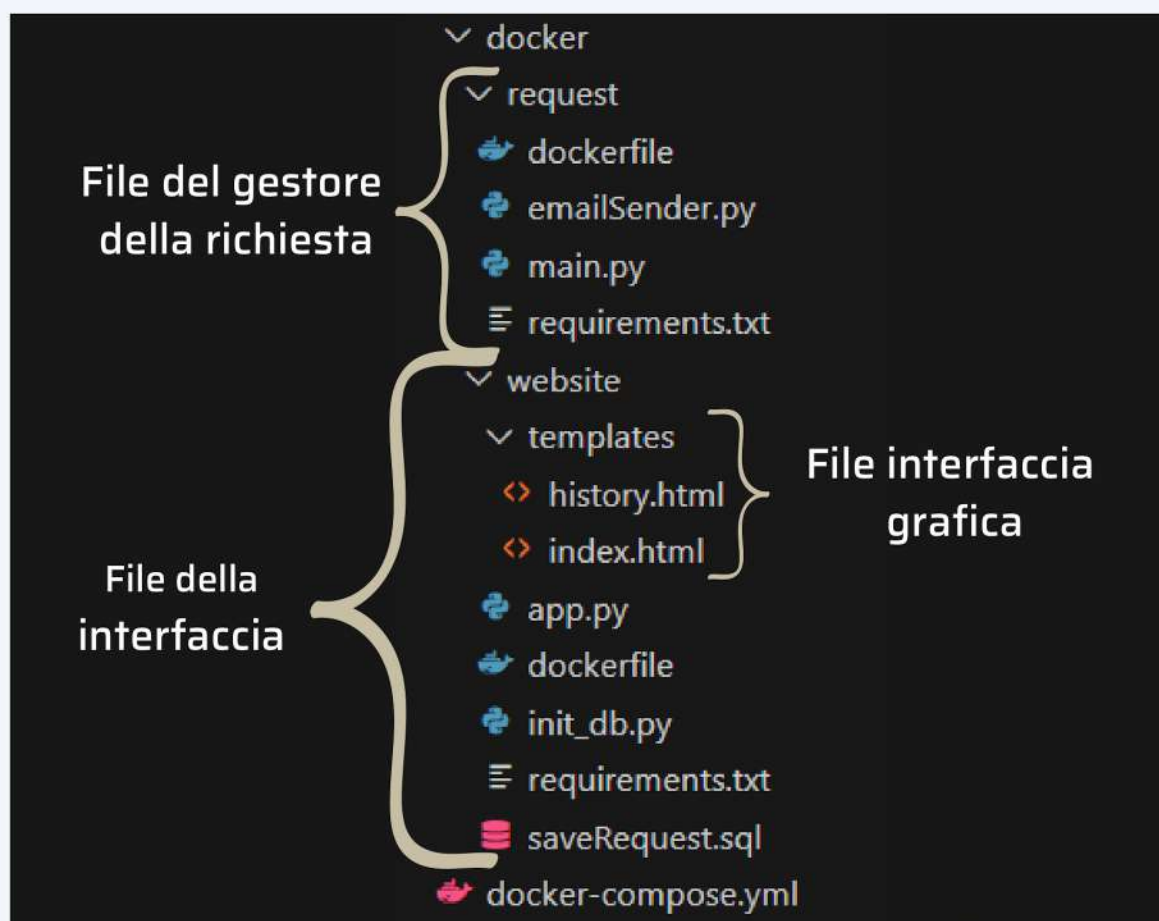


Fig. 5: Gerarchia dei file costitutivi della web app

4.1 L'INTERFACCIA

L'interfaccia di interazione con l'utente è stata realizzata attraverso il linguaggio di markup **HTML**, utilizzato per la formattazione e l'impaginazione di documenti ipertestuali. Per linguaggio di markup si intende un linguaggio realizzato per la formattazione e l'impaginazione di documenti ipertestuali di un testo nonché per la realizzazione del template di base di una web app.

Per ottenere una rappresentazione che fosse gradevole ed user-friendly si è ricorso all'utilizzo del framework open source e orientato allo sviluppo web **Bootstrap** che fornisce script basati su HTML, CSS E Javascript per varie funzioni e componenti relativi al web design.

L'homepage si consta di una barra di navigazione che permette di accedere alle due pagine finestre realizzate in HTML:

- la **Home**: la pagina principale che contiene un form per permettere al client di sottomettere i parametri necessari all'esecuzione dell'algoritmo di frequent mining in maniera intuitiva
- la **History**: la pagina secondaria che contiene l'elenco delle richieste effettuate con le informazioni relative, memorizzate a partire dall'avvio dell'applicazione

Frequent Itemset Mining Home History

Analyze your dataset

Choose the algorithm and submit a dataset. Results will be send on your email

Email address

We'll never share your email with anyone else.

Choose the algorithm
Apriori

Upload your dataset
Sfoglia... Nessun file selezionato.
We accept .csv format

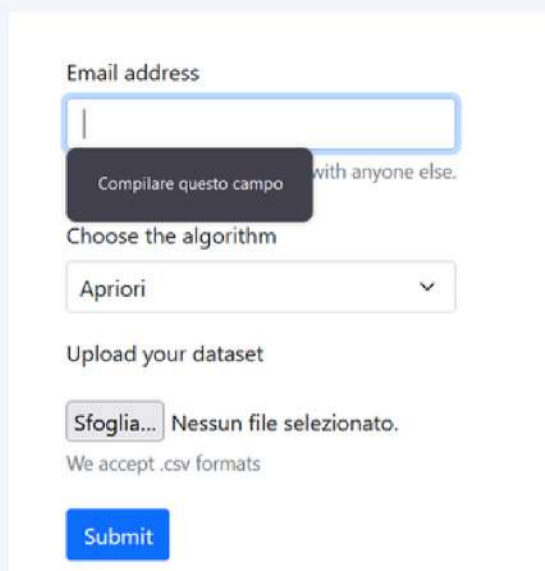
Submit

Gaia Bertolino

Fig. 6: Homepage della web app

Tramite i parametri HTML, i campi del form dell'homepage presentano delle restrizioni riguardo ciò che può essere sottoposto dall'utente:

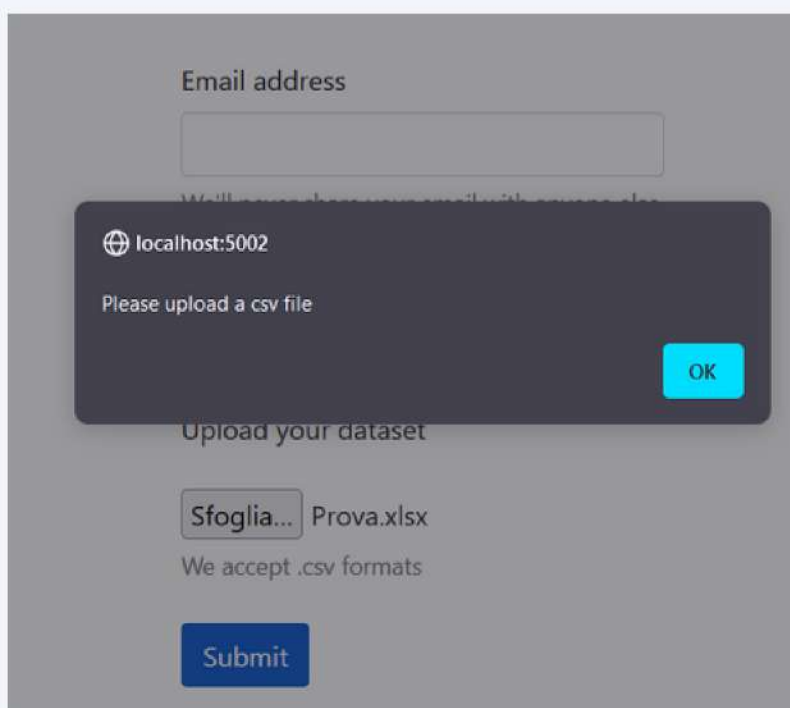
- I campi Email address e di file uploading sono **required**. Se l'utente prova a sottoporre una richiesta senza aver compilato tutti i campi riceve una opportuna segnalazione automatica da parte del sistema



The screenshot shows a web form with the following elements: an 'Email address' label above an empty text input field; a dark grey tooltip with the text 'Compilare questo campo' (Fill this field) and 'with anyone else.'; a 'Choose the algorithm' label above a dropdown menu showing 'Apriori'; an 'Upload your dataset' label above a file upload area; a 'Sfoglia...' (Browse...) button next to the text 'Nessun file selezionato.' (No file selected.); a note 'We accept .csv formats'; and a blue 'Submit' button at the bottom.

Fig. 7: Notifica in caso di presenza di campi non compilati

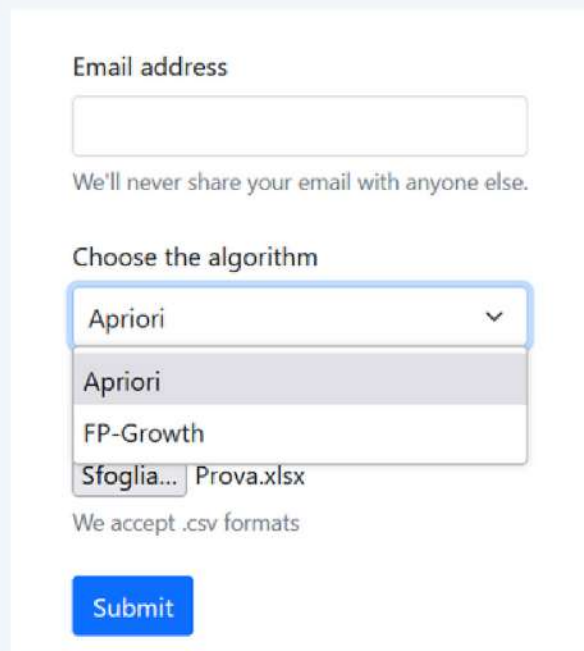
- Il file sottoposto deve necessariamente avere **estensione .csv**. Tuttavia, il sistema si predispone a future integrazioni per la lettura di file txt, Excel, ecc.



The screenshot shows the same web form as before, but with a dark grey modal dialog box in the center. The dialog contains a globe icon, the text 'localhost:5002', and the message 'Please upload a csv file'. There is a blue 'OK' button in the bottom right corner of the dialog. The background form is dimmed.

Fig. 7: Notifica in caso di sottomissione di un file con estensione diversa da .csv

- Il campo di **scelta dell'algoritmo** ha come valore di default "Apriori" che può essere modificato tramite l'apposito menu a tendina. Il sistema si predispone per l'aggiunta di altri algoritmi.

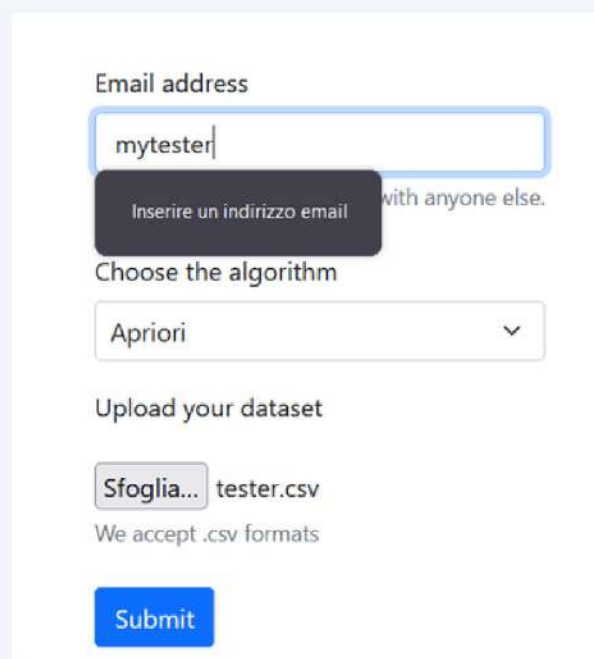


The screenshot shows a web form with the following elements:

- Email address**: A text input field.
- We'll never share your email with anyone else.**: A line of text below the email field.
- Choose the algorithm**: A dropdown menu with a blue border.
- Apriori**: The selected option in the dropdown menu.
- Apriori**: The first option in the dropdown list.
- FP-Growth**: The second option in the dropdown list.
- Sfoggia...**: A button to view more options.
- Prova.xlsx**: A file upload button.
- We accept .csv formats**: A line of text below the file upload button.
- Submit**: A blue button at the bottom.

Fig. 8: Menu a tendina per la scelta dell'algoritmo

- Il campo **Email address** è specifico per cui la sottomissione di un indirizzo non riconosciuto come **plausibile mail** viene rigettato



The screenshot shows the same web form as in Fig. 8, but with an error notification:

- Email address**: The text input field contains "mytester".
- Inserire un indirizzo email**: A dark grey notification box with white text, indicating an invalid email address.
- Choose the algorithm**: The dropdown menu is still set to "Apriori".
- Upload your dataset**: A section header.
- Sfoggia...**: A button to view more options.
- tester.csv**: A file upload button.
- We accept .csv formats**: A line of text below the file upload button.
- Submit**: A blue button at the bottom.

Fig. 9: Notifica in caso di inserimento di un valore non riconducibile ad un indirizzo mail

- Una sottomissione che rispetta i requisiti sopra affrontati genera una finestra di avviso del **corretto** avvio dell'elaborazione con conseguente notifica tramite mail del risultato

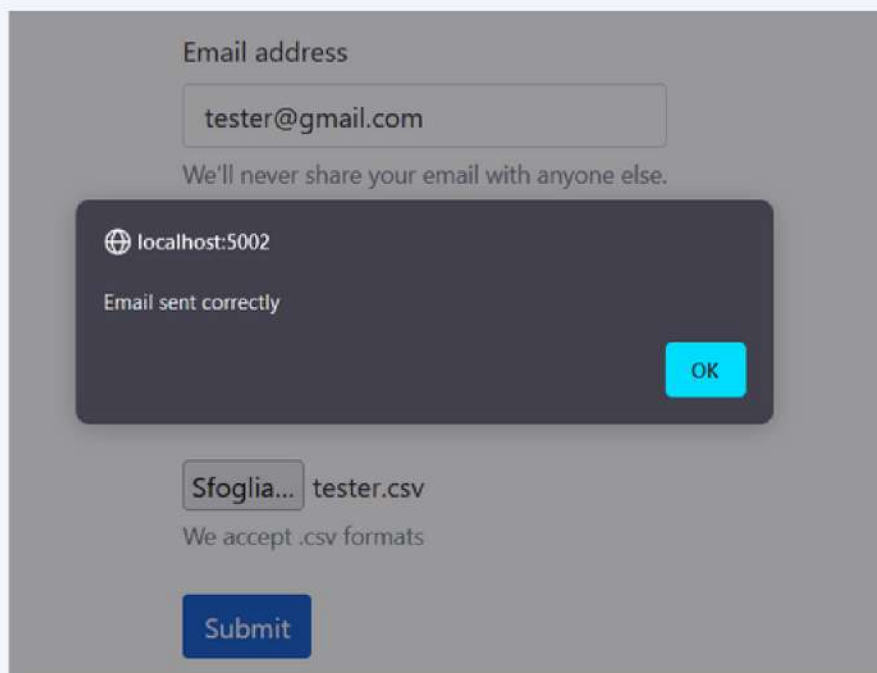


Fig. 10: Notifica di avvenuto avvio dell'elaborazione

Al corretto invio della richiesta, si viene **reindirizzati** sulla pagina History che riporta tutte le richieste effettuate dall'avvio dell'applicazione in ordine di sottomissione

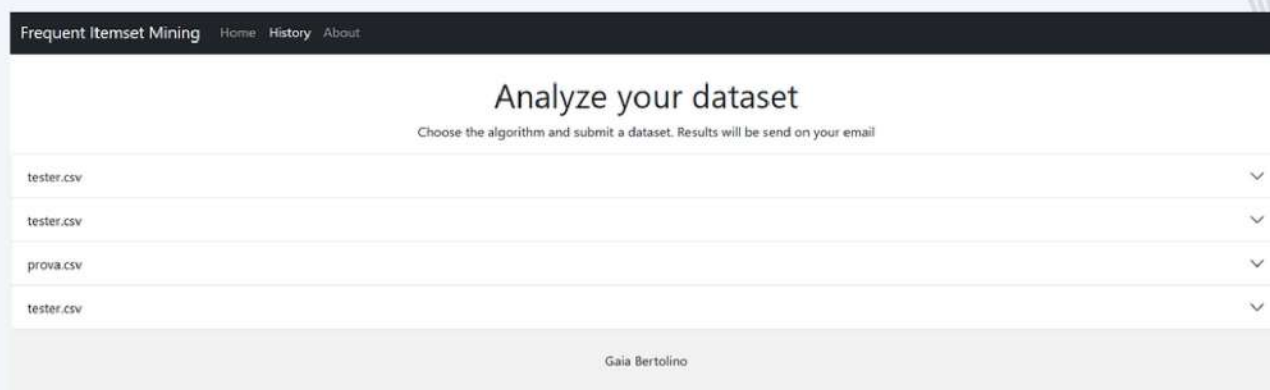


Fig. 11: Esempio di pagina History che riporta la sottomissione di quattro richieste differenti

Cliccando sulle richieste è possibile visualizzare le informazioni ad esse associate quali:

- Id: valore che identifica la richiesta, in questo caso generato come ordinale semplice
- Contatto mail inserito a cui è stato inviato il risultato
- Algoritmo scelto per l'analisi

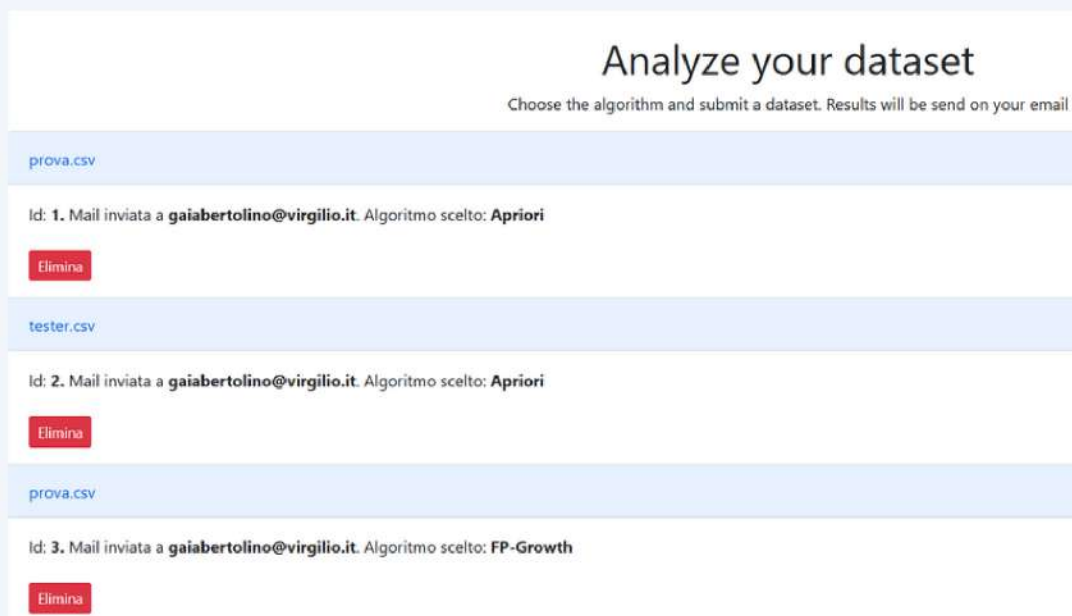


Fig. 12: Esempio della pagina History rappresentante il contenuto di tre richieste sottoposte

La gestione della cronologia è implementata tramite un database locale. Il database è generato attraverso l'uso di tre file:

- Un database, realizzato tramite un file con estensione db
- Un file Python di creazione della connessione fra l'applicazione e il database creato
- Un file con estensione sql che crea una table "requests" all'interno del database

```
docker > website > init_db.py > init
1 import sqlite3
2
3 def init():
4     connection = sqlite3.connect('database.db')
5     with open('saveRequest.sql') as f:
6         connection.executescript(f.read())
7     connection.commit()
8     connection.close()
9
```

Fig. 13: File di gestione della comunicazione con il database

```
1 DROP TABLE IF EXISTS requests;
2
3 CREATE TABLE requests (
4     id INTEGER PRIMARY KEY,
5     FileName FILESTREAM,
6     Mail TEXT,
7     Algorithm TEXT
8 );
9
```

Fig. 14: File di creazione della table contenuta all'interno del database

	id	FileName	Mail	Algorithm
1	1	prova.csv	gaiabertolino@virgilio.it	Apriori
2	2	tester.csv	gaiabertolino@virgilio.it	Apriori
3	3	prova.csv	gaiabertolino@virgilio.it	FP-Growth

Fig. 15: Esempio di contenuto del database visualizzato attraverso il software DB Browser for SQLite

In aggiunta, è stato predisposto un pulsante di **eliminazione** della richiesta dalla cronologia generata. Esso interviene direttamente sui dati salvati nel database eliminando la riga di afferenza, per poi riportare l'utente all'homepage.

La gestione delle chiamate HTTP all'applicazione è stata effettuata ricorrendo al **micro-framework Flask** realizzato per lo sviluppo di applicazioni web. Flask permette di personalizzare e semplificare aspetti dell'applicazione, tra cui l'integrazione del database e la gestione delle chiamate HTTP.

L'utilizzo di questo framework rende possibile inoltre renderizzare facilmente le pagine HTML e la gestione del comportamento dell'applicazione in caso di diverse chiamate.

```
@app.route('/', methods=('GET', 'POST'))
def index():
    if request.method == 'POST':
```

Fig. 16: Intestazione del metodo responsabile della gestione delle chiamate all'url base ('/')

```
@app.route('/history')
def history():
```

Fig. 17: Intestazione del metodo responsabile della gestione delle chiamate all'url della pagina History ('/history')

```
if __name__ == "__main__":
    port = int(os.environ.get('PORT', 5000))
    app.run(debug=True, host='0.0.0.0', port=port)
```

Fig. 18: Metodo di gestione delle chiamate alla web app con specificazione della porta di ascolto

```
return redirect('/history')
return render_template('index.html')
```

Fig. 19: Utilizzo del reindirizzamento alle pagine della web app

Come accennato nel capitolo 2, il processo di creazione dell'immagine relativa ad una applicazione richiede la scrittura di un **dockerfile**, un file di testo che contiene una serie di istruzioni, ciascuna su una riga separata, in un DSL. Le istruzioni vengono eseguite una dopo l'altra per creare un'immagine Docker aggiungendo ad ogni passaggio dei layer, solitamente ad una immagine di partenza. Il processo di costruzione viene avviato eseguendo il comando "docker build".

Per non includere tutti i file presenti nella directory di origine locale nel contesto di costruzione si può ricorrere all'introduzione del file **.dockerignore**, usato per escludere file e directory dal contesto di costruzione, il cui nome prende spunto dal file **.gitignore** di Git. Il punto posto all'inizio del nome del file indica che si tratta di un file nascosto. Nell'ambito di questo progetto, non è stato necessario ricorrere ad alcun file **.dockerignore**.

Il docker file deve gestire l'installazione di tutte le componenti utilizzate nell'applicazione, rispettandone le versioni per ovviare a problemi di compatibilità e per ricreare lo stesso ambiente dello sviluppo. A tal proposito si sono esplicitate tutte le dipendenze ricorrendo ad un file **requirements.txt** opportunamente generato attraverso la funzione pip freeze, la quale è in grado di enumerare tutte le dipendenze e le librerie del proprio workspace.

```
docker > website > requirements.txt
1  Flask==2.3.2
2  gitdb==4.0.10
3  GitPython==3.1.30
4  mlxtend==0.22.0
5  mysql-connector==2.2.9
6  mysql-connector-python==8.0.27
7  npx==0.1.1
8  numpy==1.24.1
9  packaging==23.0
10 pandas==1.5.3
11 PySimpleGUI==4.60.5
12 python-dateutil==2.8.2
13 requests==2.28.2
14 scikit-learn==1.2.2
```

Fig. 20: Contenuto del file requirements.txt utilizzato

```
docker > website > dockerfile > ...
1  FROM python:latest
2
3  COPY . .
4  RUN pip install --upgrade pip
5  RUN pip install --upgrade setuptools
6  RUN pip3 install -r requirements.txt
7
8
9  CMD ["python3", "app.py"]
10
```

Fig. 21: Contenuto del dockerfile realizzato per la containerizzazione dell'interfaccia

Come accennato, Docker permette di generare una immagine personale partendo da una base già diffusa nello spazio pubblico di Docker Hub, uno strumento che permette di rendere fruibili i propri software e da cui poter attingere per i propri progetti.

Nell'ambito di questo progetto, sia per la containerizzazione dell'interfaccia che della gestione dell'elaborazione più avanti descritta, si è fatto ricorso all'immagine ufficiale python.

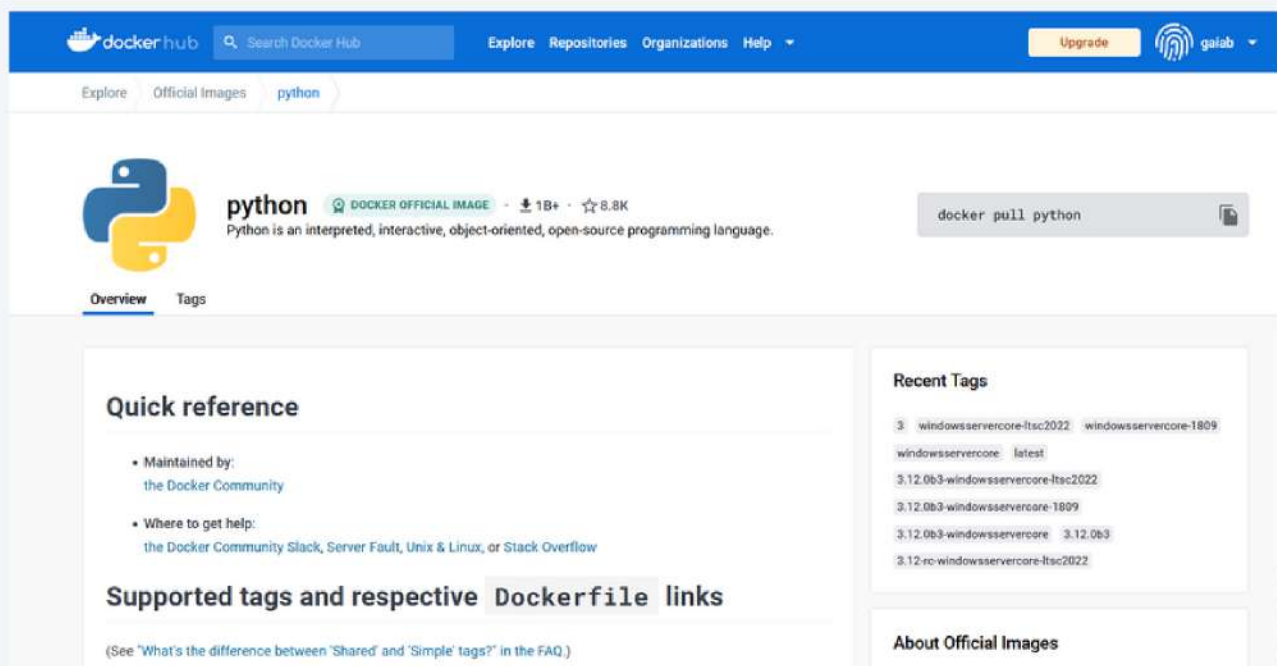


Fig. 22: Pagina di visualizzazione dell'immagine python presente su Docker Hub

4.B IL GESTORE DELLA RICHIESTA

Il gestore della richiesta si compone di due file Python demandati a gestire le diverse funzionalità dell'elaborazione:

- Il file **principale** che si occupa di gestire le chiamate POST ricevute da parte, nel caso di questo progetto, dall'interfaccia di interazione con l'utente sopra descritta, in cui vengono inviate le informazioni necessarie all'elaborazione del dataset
- Il file **secondario** che si occupa di gestire l'invio della notifica tramite mail attraverso il protocollo standard per la trasmissione di email SMTP (acronimo per Simple Mail Transfer Protocol).

In particolare, il file principale si occupa di leggere i dati da file e di valutare se la formattazione è tale da poter costruire un dataframe Pandas. Successivamente si occupa dell'attività di **one-hot encoding** dei dati ovvero di eseguire il processo per cui i dati categorici ricevuti vengono trasformati in numerici binari al fine di potervi applicare algoritmi vari. Tale azione si rende necessaria per applicazioni d esempio di machine learning o frequent itemset mining. Se l'esecuzione avviene correttamente, viene infine applicato l'algoritmo richiesto tramite l'utilizzo delle apposite funzioni fornite dalla libreria **Mlxtend** e il risultato ottenuto viene salvato in un file txt.

L'applicazione si predispone all'introduzione di ulteriori algoritmi nonché ad una formattazione tabellare del risultato.

```
18 def algo(data, algorithm):
19     # Transforming in array
20     dataset = []
21     for i in range(0, len(data)):
22         dataset.append([str(data.values[i,j]) for j in range(0, len(data.iloc[0]))])
23
24     # Onehotencoding of data
25     encoder = TransactionEncoder()
26     oneHotEncoder = encoder.fit(dataset).transform(dataset)
27     enc = pd.DataFrame(oneHotEncoder, columns=encoder.columns_)
28
29     # Extracting the most frequent itemsets via Mlxtend
30     if algorithm == "Apriori":
31         frequent_itemsets = apriori(enc, min_support=0.01, use_colnames=True)
32     else:
33         frequent_itemsets = fpgrowth(enc, min_support=0.01, use_colnames=True)
34     frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda x: len(x))
35
36     # Saving the result in a txt
37     np.savetxt(r'result.txt', frequent_itemsets.values, fmt='%s', header=' '.join(frequent_itemsets.columns))
```

Fig. 23: Codice di gestione dell'encoding del dataset e di applicazione dell'algoritmo richiesto

Come accennato, il file secondario fornisce invece una funzione generica di invio mail che permette di specificare il mittente, l'oggetto, l'eventuale file da mandare e il corpo del messaggio. In particolare, per gestire opportunamente casi di errore, è previsto che al passaggio di un campo vuoto nel campo relativo al path del file non venga generato alcun allegato.

```
14 def send_email(user, pwd, recipient, subject, fileToSend, body):
15     body = body
16     FROM = user
17     TO = recipient
18     SUBJECT = subject
19     TEXT = body
20
21     # Prepare actual message
22     msg = MIMEText(body, 'plain')
23     msg['From'] = FROM
24     msg['To'] = TO
25     msg['Subject'] = SUBJECT
26
27     msgText = MIMEText(body, 'plain')
28     msg.attach(msgText)
29
30     # Attach the file
31     if fileToSend != "":
32         file = fileToSend
33         attachment = MIMEBase('application', 'octet-stream')
34         attachment.set_payload(open(file, 'rb').read())
35         encoders.encode_base64(attachment)
36         attachment.add_header('Content-Disposition', 'attachment; filename="%s"' % os.path.basename(file))
37         msg.attach(attachment)
38
39     try:
40         server = smtplib.SMTP("smtp.gmail.com", 587)
41         server.ehlo()
42         server.starttls()
43         server.login(user, pwd)
44         message = msg.as_string()
45         server.sendmail(FROM, TO, message)
46         server.close()
47         print('Successfully sent the mail')
48     except:
49         print("Failed to send mail")
```

Fig. 24: Codice di generazione ed invio della mail di notifica all'utente

E' dunque lo script principale in carica di popolare opportunamente i campi di invio della mail. Infatti, nel caso in cui si sottometta un file non correttamente formattato, l'applicazione non può generare alcun risultato e l'utente viene notificato tramite mail della non corretta esecuzione dell'algoritmo, invitandolo a rivedere la correttezza del dataset. In caso contrario, l'utente riceverà una mail contenente i risultati dell'elaborazione.

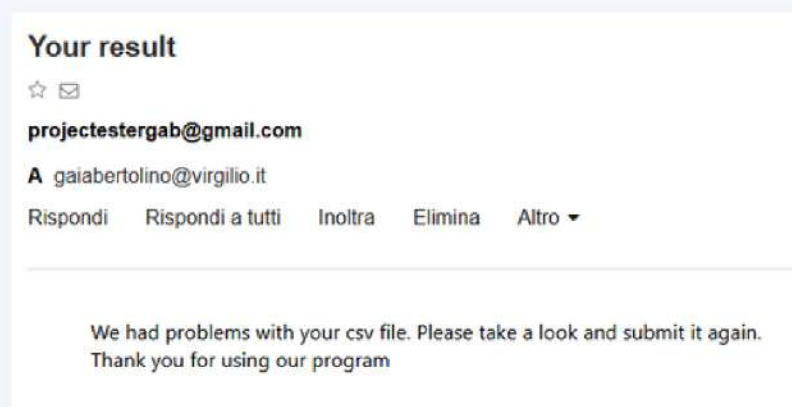


Fig. 25: Esempio di mail contenente una notifica di errore

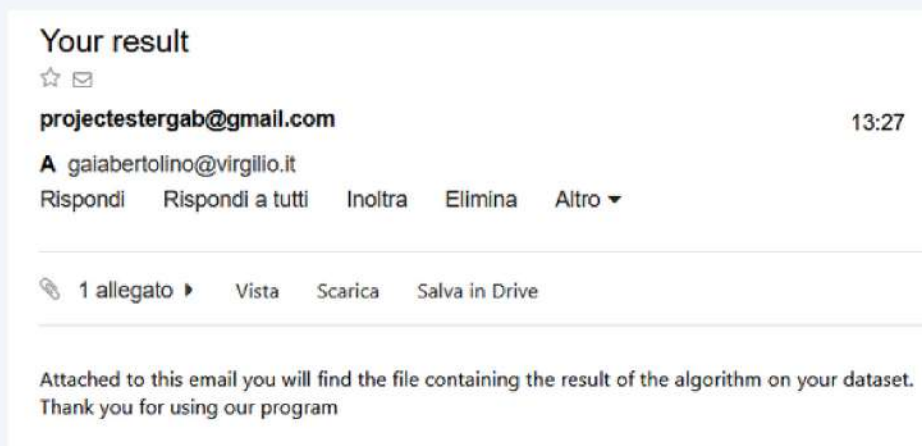


Fig. 26: Esempio di mail contenente il risultato della corretta elaborazione

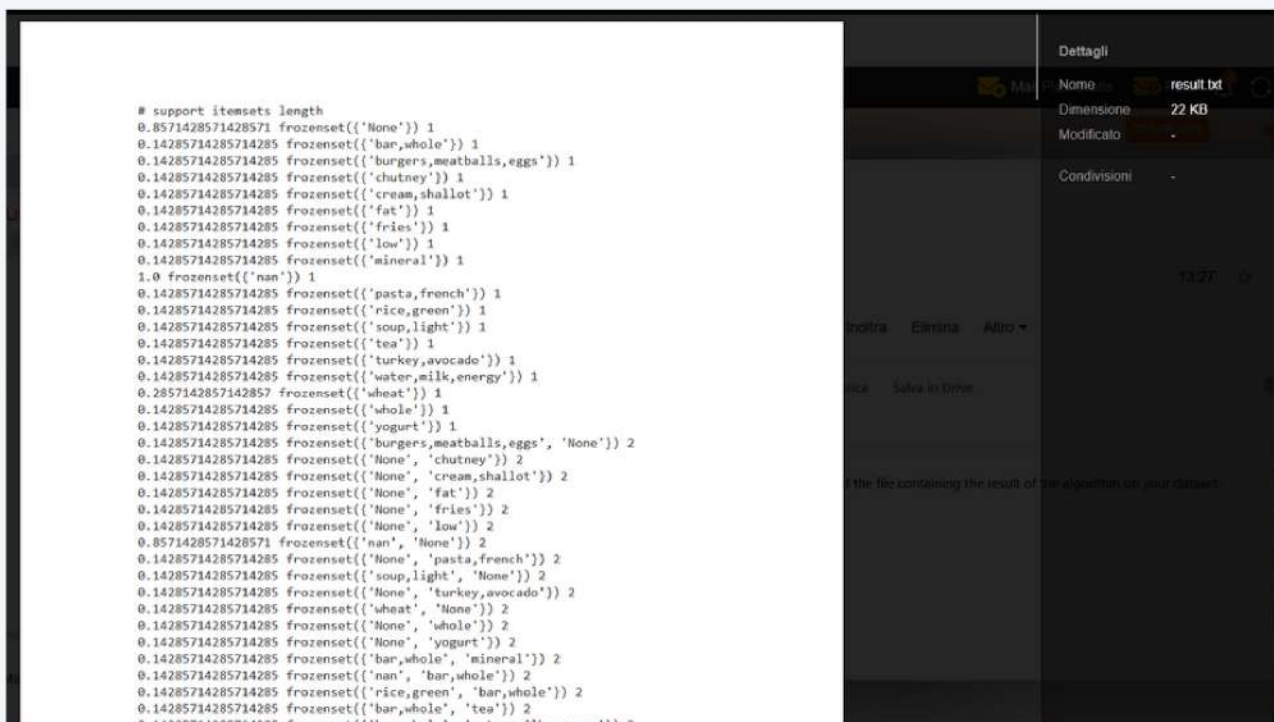


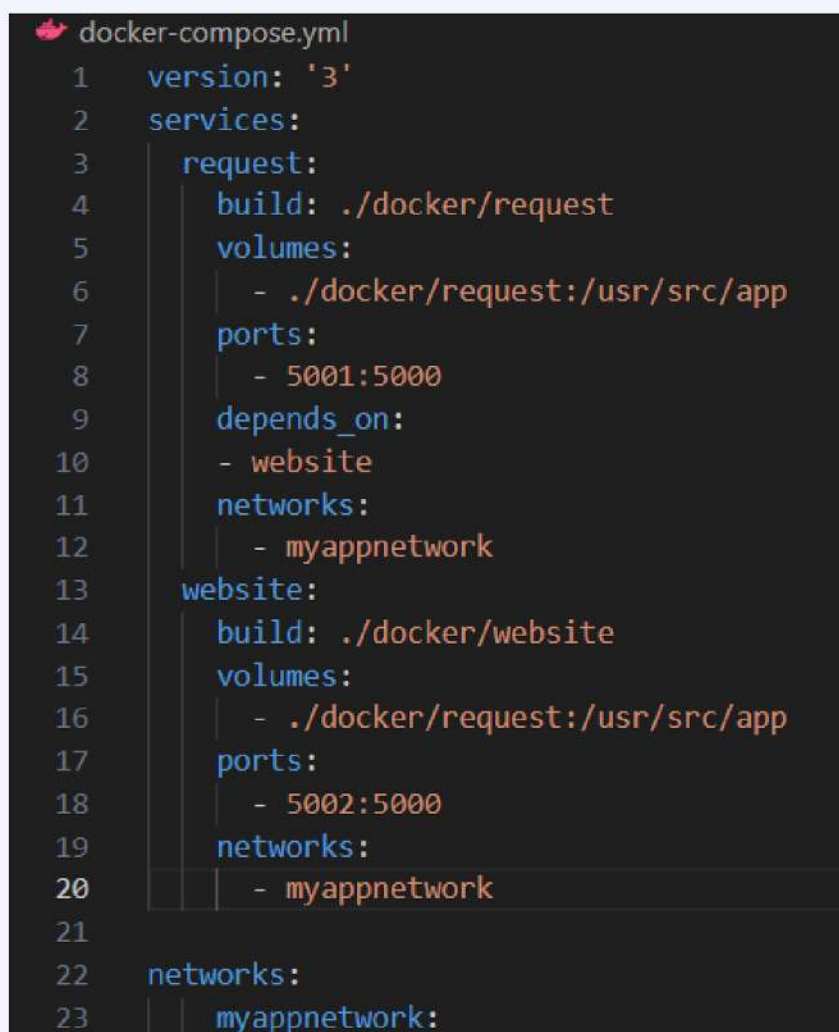
Fig. 27: Esempio di file contenente i risultati ricevuto tramite mail

Come accennato precedentemente, per il gestore delle richieste è stata realizzata una immagine a parte dall'interfaccia al fine di isolare la sua specifica funzionalità e di rendere possibile una futura orchestrazione più efficiente. Come visto per la parte relativa all'interfaccia, per ottenere la containerizzazione è stato realizzato un `dockerfile` facente sempre uso dell'immagine ufficiale di python e un `requirements.txt` file ottenuto nuovamente col comando `pip freeze`.

4.C DOCKER COMPOSE

Visti i capitoli precedenti, è possibile concludere come la web app sia composta da ben due macroaree, ciascuna containerizzata separatamente dall'altra. A tal proposito, per istanziare un'applicazione i cui file sono organizzati in immagini diverse bisogna fare ricorso ad organo di orchestrazione della comunicazione fra i container, il tool **Docker Compose**. Esso consente di gestire le proprie app multi-container salvandone le modalità in un unico file di configurazione scritto in formato **YAML**.

In particolare, esso consente di specificare parametri rilevanti (descritti brevemente nel paragrafo 2) quali i network di comunicazione, i tag identificativi di ciascun servizio, i relativi volumes e le porte da cui saranno accessibili i servizi.



```
docker-compose.yml
1  version: '3'
2  services:
3    request:
4      build: ./docker/request
5      volumes:
6        - ./docker/request:/usr/src/app
7      ports:
8        - 5001:5000
9      depends_on:
10       - website
11     networks:
12       - myappnetwork
13   website:
14     build: ./docker/website
15     volumes:
16       - ./docker/request:/usr/src/app
17     ports:
18       - 5002:5000
19     networks:
20       - myappnetwork
21
22   networks:
23     myappnetwork:
```

Fig. 28: Contenuto del file di configurazione della composition

Più dettagliatamente, ciascun container è riportato come oggetto di tipo `service` e ne vengono specificate la porta di reindirizzamento rispetto alla porta di afferenza specificata nel proprio codice, permettendo dunque che si possano eseguire più applicazioni con la stessa porta, la locazione dei propri volumes e la directory in cui si trova il rispettivo dockerfile.

Infine, il file di composition esegue più container in contemporanea e permette a ciascuno di riferirsi agli altri service presenti sulla rete tramite il proprio tag e la porta di afferenza.

```
10 app = Flask(__name__)
11 init_db.init()
12
13 @app.route('/', methods=('GET', 'POST'))
14 def index():
15     if request.method == 'POST':
16
17         # Saving the file
18         path = str(os.getcwd() + "\\") + request.files['file'].filename
19         request.files['file'].save(path)
20
21         # Verification of data
22         if request.files['file'].filename.endswith('.csv'):
23
24             # Saving in the database
25             connection = sqlite3.connect('database.db')
26             connection.row_factory = sqlite3.Row
27             connection.execute('INSERT INTO requests (FileName, Mail, Algorithm) VALUES (?, ?, ?)',
28                               (request.files['file'].filename, request.form['mail'], request.form['algorithm']))
29             connection.commit()
30             connection.close()
31
32             url = r'http://request:5000'
33             with open(path, 'rb') as f:
34                 r = requests.post(url, files={'file': f}, data = {'algorithm': request.form['algorithm'],
35                                                                'mail': request.form['mail']})
36
37     return redirect('/history')
38     return render_template('index.html')
```

```
url = r'http://request:5000'
with open(path, 'rb') as f:
    r = requests.post(url, files={'file': f}, data = {'algorithm': request.form['algorithm'],
                                                    'mail': request.form['mail']})
```

Fig. 29: Esempio di uso del tag di un service all'interno di un altro service

Studentessa
GAIA ASSUNTA
BERTOLINO
MAT. 242590

