

UNIVERSITÀ<sup>2</sup> DEGLI STUDI DI  
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di laurea in Ingegneria Informatica

TESINA  
**BIG DATA ENGINEERING**

Iole Morabito M63001448

Doriana Traetto M63001416

# Indice

<b>1. Introduzione.....</b>	<b>3</b>
1.1    Contesto e motivazioni .....	3
1.2    Obiettivi del progetto.....	3
<b>2. Risorse .....</b>	<b>4</b>
2.1    Dataset .....	4
2.2    Strumenti utilizzati .....	4
<b>3. Modellazione del problema .....</b>	<b>6</b>
3.1    Divisione in anamnesi e diagnosi .....	6
3.2    Workflow del progetto .....	7
3.3    Scelta del modello .....	7
<b>4. Estrazione e Analisi dei dati .....</b>	<b>8</b>
4.1    Pre-processing dei dati .....	8
4.2    NER .....	11
4.3    RE .....	13
<b>5. Knowledge Graph .....</b>	<b>18</b>
5.1    Processo di costruzione .....	18
5.2    Esempio di grafi creati.....	19
<b>6. Dashboard .....</b>	<b>20</b>
6.1    Progettazione dell’interfaccia utente .....	20
6.2    Implementazione delle funzionalità .....	21
6.3    Visualizzazione dei dati .....	36
<b>7. Requisito opzionale .....</b>	<b>39</b>
7.1    Scelte progettuali .....	39
7.2    Implementazione delle funzionalità .....	39
7.3    Visualizzazione dei dati .....	44
<b>8. Risultati .....</b>	<b>45</b>
8.1    Benefici ottenuti .....	45
8.2    Limiti riscontrati .....	45
8.3    Futuri sviluppi e ricerche.....	46

# 1. Introduzione

## 1.1 Contesto e motivazioni

Negli ultimi anni, il settore medico ha visto una crescita esponenziale nella quantità di dati prodotti quotidianamente. Ogni caso clinico genera una mole significativa di informazioni che, se opportunamente analizzate, possono offrire insight preziosi per migliorare la cura dei pazienti. Tuttavia, gran parte di questi dati sono non strutturati, provenienti da note cliniche scritte da medici e infermieri, che descrivono dettagliatamente le condizioni dei pazienti, i trattamenti prescritti e i risultati delle diagnosi.

Il contesto dei big data nel settore sanitario richiede strumenti avanzati per l'estrazione e l'analisi di queste informazioni. Le tecnologie tradizionali di gestione dei dati spesso non sono sufficienti per trattare la complessità e la varietà dei dati clinici non strutturati. È qui che entrano in gioco tecniche avanzate di Natural Language Processing (NLP) e di intelligenza artificiale, che permettono di trasformare il testo non strutturato in dati strutturati e significativi. Utilizzando modelli di linguaggio avanzati (LLM) e costruendo grafi della conoscenza (Knowledge Graph), è possibile creare strumenti potenti per supportare i medici nelle loro decisioni, migliorare la precisione diagnostica e facilitare la comunicazione tra i professionisti della salute.

## 1.2 Obiettivi del progetto

L'obiettivo principale di questo progetto è sviluppare un sistema che estragga e analizzi dati testuali non strutturati provenienti da note cliniche riportate in italiano, al fine di costruire un Knowledge Graph utile per la creazione di analytics, report e visualizzazione dei dati. Questo sistema mira a:

- Automatizzare l'estrazione delle informazioni cliniche: tramite tecniche di Named Entity Recognition (NER) e Relation Extraction (RE), il sistema sarà in grado di identificare e classificare concetti medici chiave, come malattie, sintomi, farmaci, eventuali esami e procedure effettuate o da effettuare.
- Costruire un Knowledge Graph: organizzare le informazioni estratte in un grafo della conoscenza, che rappresenti in modo strutturato le relazioni tra i diversi concetti medici. Questo grafo faciliterà l'accesso e la visualizzazione delle informazioni cliniche, migliorando la comprensione complessiva delle condizioni dei pazienti e delle loro storie cliniche.
- Sviluppare una dashboard interattiva: creare un'interfaccia utente intuitiva e personalizzabile utilizzando Streamlit, che permetta ai medici e agli infermieri di visualizzare le informazioni cliniche in modo chiaro e conciso e di accedere facilmente al Knowledge Graph. La dashboard consentirà anche di personalizzare le visualizzazioni e le analisi secondo le specifiche esigenze degli utenti.

Attraverso il raggiungimento di questi obiettivi, il progetto si propone di fornire un contributo significativo al miglioramento delle pratiche mediche e alla promozione di una sanità più efficiente e data driven, sfruttando appieno il potenziale delle tecnologie emergenti nel campo dei big data.

# 2. Risorse

## 2.1 Dataset

Il dataset utilizzato per questo progetto è stato ottenuto dal **MultiCardioNER Corpus**, un'iniziativa finalizzata all'adattamento di sistemi di riconoscimento delle entità nominate (NER) nel dominio della cardiologia. Questo corpus si basa su una combinazione di due dataset esistenti, ovvero DisTEMIST per le malattie e DrugTEMIST per i farmaci. Questo dataset è stato creato dal Barcelona Supercomputing Center's NLP for Biomedical Information Analysis ed è stato utilizzato come parte di BioASQ 2024.

Ai fini del nostro elaborato abbiamo scelto di utilizzare **DisTEMIST**, composto da 250 documenti clinici accuratamente annotati per includere menzioni di varie condizioni mediche. Questi documenti provengono da casi clinici reali e contengono dettagli significativi riguardanti la storia clinica del paziente, l'evoluzione dello stato di salute, gli esami diagnostici eseguiti, i trattamenti ricevuti e le conclusioni diagnostiche.

## 2.2 Strumenti utilizzati

In Figura 1 è possibile osservare un diagramma di deployment che offre una rappresentazione visiva ad alto livello e che illustra la distribuzione fisica dei componenti del sistema software su hardware o ambienti di esecuzione, senza entrare nei dettagli interni del funzionamento dei singoli componenti:

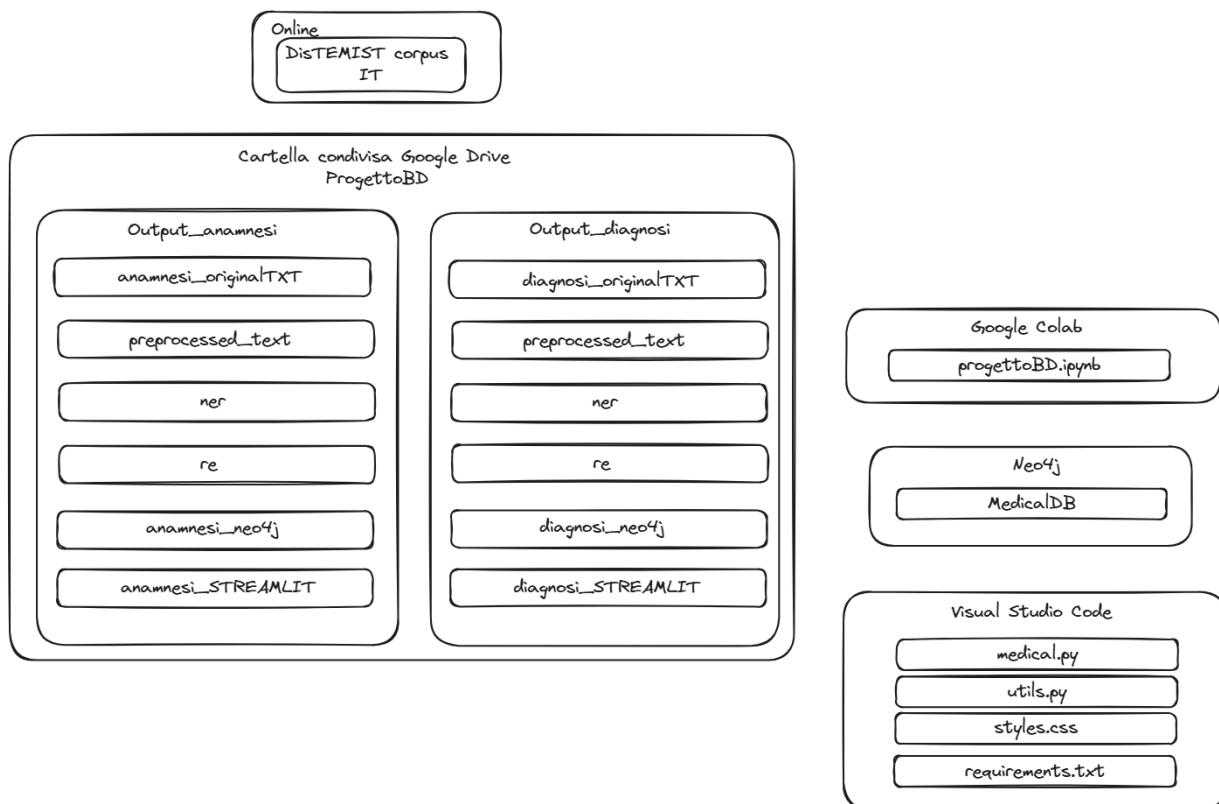


Figura 1: Diagramma di deployment

Nel corso del progetto sono stati utilizzati diversi strumenti software e librerie per supportare le varie fasi di elaborazione e analisi dei dati.

**Google Colab** è un ambiente di sviluppo basato su cloud che fornisce una piattaforma integrata per eseguire analisi e operazioni su grandi volumi di dati. Grazie alla sua capacità di utilizzare risorse di calcolo scalabili, come le unità di elaborazione grafica (GPU) e le unità di elaborazione tensoriale (TPU), Colab consente agli utenti di elaborare e analizzare grandi dataset in modo efficiente e veloce. Inoltre, l'integrazione con Google Drive consente di accedere e gestire i dati direttamente dallo stesso ambiente di sviluppo, semplificando il processo di caricamento, archiviazione e condivisione dei dati. Questo rende Google Colab una scelta popolare per gli analisti e gli scienziati dei dati che lavorano con grandi dataset nel contesto dei big data.

**Google Drive** è stato utilizzato per archiviare i dataset, i file pre-elaborati e i risultati intermedi e finali del progetto. È stato montato su Google Colab per permettere l'accesso ai file direttamente dallo script.

**Hugging Face** è una piattaforma leader nel campo dei modelli di linguaggio (LLM) e delle tecnologie di intelligenza artificiale. Fornisce una vasta gamma di modelli pre-addestrati per task come il Named Entity Recognition (NER) e il Relation Extraction (RE), fondamentali per il progetto. La scelta del LLM tramite Hugging Face permette di sfruttare tecniche avanzate di NLP per estrarre informazioni significative da grandi volumi di testo non strutturato. In un contesto di big data, Hugging Face offre strumenti scalabili e facilmente integrabili che migliorano significativamente l'accuratezza e l'efficienza del processo di analisi dei dati clinici, supportando la creazione di un Knowledge Graph accurato e dettagliato.

**Neo4j** è un database a grafo estremamente potente e versatile, particolarmente adatto per gestire e analizzare dati complessi con relazioni intricate, come quelli presenti nel settore medico. Utilizzando il linguaggio di query Cypher, Neo4j permette di estrarre e visualizzare informazioni dettagliate e connesse tra loro in modo intuitivo e rapido. Nel contesto dei big data, Neo4j e Cypher si rivelano strumenti indispensabili per la costruzione di un Knowledge Graph, in quanto consentono di modellare, memorizzare e interrogare grandi quantità di dati non strutturati provenienti dalle note cliniche, facilitando l'identificazione di pattern e la generazione di insight utili per i professionisti della salute.

**Visual Studio Code** è un ambiente di sviluppo integrato (IDE) ampiamente utilizzato da sviluppatori e ingegneri software per la creazione, la gestione e la manutenzione di applicazioni e soluzioni software complesse. È stato utilizzato per implementare in linguaggio Python il codice per la dashboard Streamlit.

**Streamlit** è una piattaforma open-source che facilita la creazione di applicazioni web interattive per la visualizzazione e l'analisi dei dati. Nel contesto del progetto, Streamlit è utilizzato per sviluppare una dashboard user-friendly che permetta al personale sanitario di accedere facilmente alle informazioni estratte dal Knowledge Graph. Questa piattaforma consente di integrare rapidamente vari componenti visuali, come grafici e tavole, e di creare interfacce personalizzabili che rispondono alle specifiche esigenze degli utenti finali. Streamlit si distingue per la sua semplicità d'uso e la capacità di trasformare script Python in applicazioni web complete, rendendolo ideale per il contesto dei big data in cui la presentazione chiara e immediata delle informazioni è cruciale.

# 3. Modellazione del problema

## 3.1 Divisione in anamnesi e diagnosi

Dopo aver consultato un esperto di dominio, si è scelto di dividere il processo in sezioni di anamnesi e diagnosi è stata presa per ottimizzare l'accuratezza e la pertinenza dell'estrazione delle entità e delle relazioni dai testi clinici. In figura 2 è possibile osservare il **grafo ad alto livello** che ci ha fornito una panoramica generale e una comprensione iniziale della struttura dei dati clinici, consentendoci di identificare i concetti chiave e le interconnessioni principali:

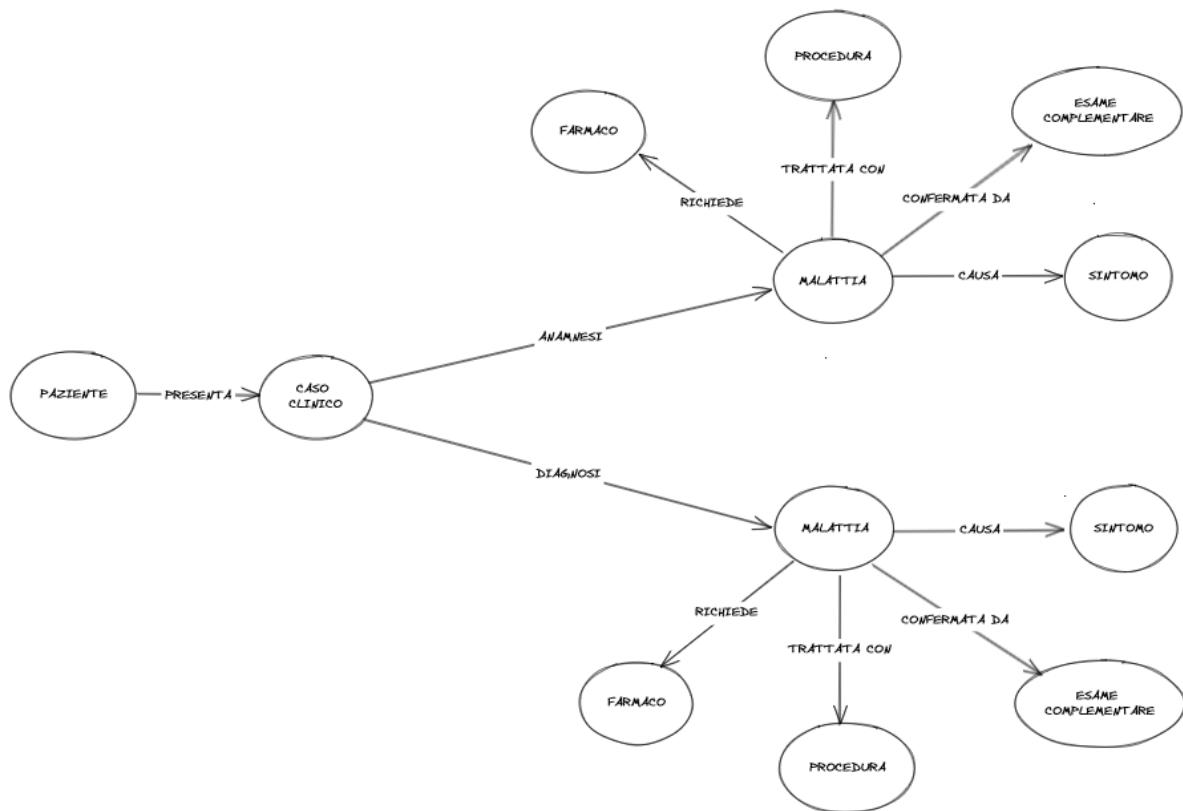


Figura 2: Grafo ad alto livello.

Questa suddivisione permette di trattare separatamente le informazioni relative alla storia clinica del paziente (anamnesi) e le informazioni diagnostiche (diagnosi), migliorando la precisione dell'identificazione delle entità specifiche per ciascuna categoria.

Nella fase di anamnesi, ci si concentra principalmente sull'estrazione di entità come sintomi, malattie passate, procedure ed esami già effettuati e farmaci precedentemente assunti, che sono cruciali per comprendere la storia clinica del paziente. Nella fase di diagnosi, invece, l'attenzione è rivolta ai sintomi, alle malattie diagnosticate, alle procedure e agli esami complementari da eseguire e ai farmaci da assumere, che sono essenziali per il trattamento attuale del paziente.

## 3.2 Workflow del progetto

In Figura 3 si mostra il workflow completo seguito nel progetto:

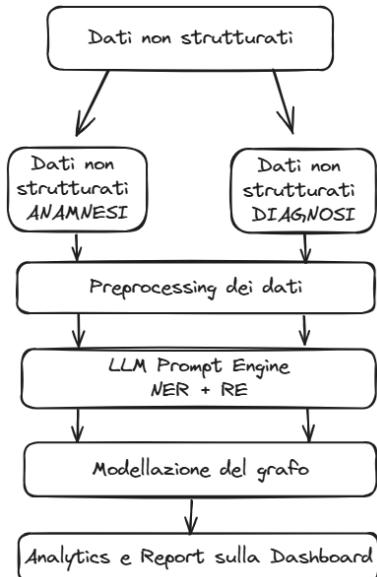


Figura 3: Workflow del progetto

## 3.3 Scelta del modello

In questa sezione, descriviamo il processo di addestramento del modello di linguaggio utilizzato per l'analisi dei testi clinici. L'addestramento del modello di linguaggio è una fase cruciale per garantire che il modello possa comprendere e processare correttamente il linguaggio naturale specifico del dominio medico.

La scelta del modello di linguaggio (LLM) è stata guidata da diversi criteri, quali le prestazioni, il supporto della lingua italiana, le capacità di adattamento del modello e la disponibilità di risorse pre-addestrate. Abbiamo valutato i modelli basati sulle loro prestazioni nei benchmark di comprensione del linguaggio naturale, con particolare attenzione ai dataset medici. È stato fondamentale selezionare un modello che supportasse efficacemente la lingua italiana, poiché i casi clinici sui quali abbiamo lavorato si presentavano in italiano. Il modello scelto doveva essere facilmente adattabile e personalizzabile per il nostro specifico compito di estrazione di entità e relazioni nel dominio medico. Abbiamo considerato modelli per i quali erano disponibili risorse e modelli pre-addestrati su dati simili, come il modello GLiNER per il riconoscimento delle entità in testi clinici italiani.

Abbiamo scelto di utilizzare il modello **DeepMount00/universal\_ner\_ita**<sup>1</sup> per il riconoscimento delle entità. Questo modello è stato selezionato per le sue elevate prestazioni nel riconoscimento delle entità nominative in testi clinici italiani.

<sup>1</sup> [https://huggingface.co/DeepMount00/universal\\_ner\\_ita](https://huggingface.co/DeepMount00/universal_ner_ita)

# 4. Estrazione e Analisi dei dati

## 4.1 Pre-processing dei dati

In questa fase verrà descritto il processo di estrazione di concetti e relazioni dai testi clinici. Questo processo è fondamentale per trasformare il linguaggio naturale in dati strutturati, che possono essere successivamente analizzati e utilizzati per varie applicazioni nel dominio medico.

Il processo di suddivisione è stato implementato attraverso uno script Python che, in base ai percorsi relativi ai file di input (file di testo originali), consente di ottenere i file suddivisi in anamnesi e diagnosi in due percorsi di output separati.

```
import os

# Percorsi di input e output

input_path = '/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/it/brat/'

output_path1 = '/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/anamnesi/'

output_path2 = '/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/diagnosi/'

# Paragrafi target per ogni testo

testo1_sections = ['ANAMNESI', 'MALATTIA ATTUALE', 'CONTESTO', 'EVOLUZIONE
CLINICA', 'DECORSO CLINICO', 'EVOLUZIONE']

testo2_sections = ['MALATTIA IN CORSO', 'ESAME FISICO', 'ESAMI COMPLEMENTARI',
'DIAGNOSI', 'FARMACI']

# Funzione per selezionare paragrafi specifici

def extract_sections(paragraphs, sections):
    extracted = []
    for para in paragraphs:
        lower_para = para.strip().lower()
        for section in sections:
            if lower_para.startswith(section.lower()):
                extracted.append(para)
                break
    return '\n\n'.join(extracted)
```

```

# Itera su tutti i file di testo nel percorso di input
for file_name in os.listdir(input_path):
    if file_name.endswith('.txt'):
        # Leggi il contenuto del file
        with open(os.path.join(input_path, file_name), 'r') as file:
            content = file.read()

        # Dividi il contenuto in paragrafi
        paragraphs = content.split('\n\n')

        # Estrai i paragrafi per Testo 1 e Testo 2
        testo1 = extract_sections(paragraphs, testo1_sections)
        testo2 = extract_sections(paragraphs, testo2_sections)

        # Scrivi i due nuovi testi nei percorsi di output
        with open(os.path.join(output_path1, file_name), 'w') as file:
            file.write(testo1)

        with open(os.path.join(output_path2, file_name), 'w') as file:
            file.write(testo2)

```

Il seguente comando indica che stiamo utilizzando l'interprete Python per eseguire il modulo "spacy" e richiedere il download del modello "it\_core\_news\_sm". Questo modello contiene una pipeline NLP già addestrata specifica per l'italiano.

```
!python -m spacy download it_core_news_sm
```

Il seguente comando consente di eseguire il preprocessing dei testi clinici relativi all'anamnesi, al fine di renderli adatti per l'elaborazione successiva. È analogo per i testi relativi alla diagnosi. La funzione *preprocess\_text(text)* tokenizza il testo in frasi, in parole, trasforma le parole in minuscolo, rimuove la punteggiatura e restituisce il testo preprocessato. Ogni file di testo viene elaborato individualmente e salvato nella stessa struttura della cartella di input, ma in una directory diversa per i testi preprocessati dell'anamnesi e della diagnosi.

```

import os
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
import spacy
import string

```

```

import pandas as pd

# Scarica le risorse necessarie per NLTK
nltk.download('punkt')
nltk.download('stopwords')

# Carica un modello di lingua spacy preaddestrato
nlp = spacy.load("it_core_news_sm")

# Definisci il percorso della cartella contenente i file di testo su Google
# Drive
folder_path =
"/content/drive/MyDrive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multi
cardioner_train+dev+test+bg_240502/track2/cardioccc_dev/anamnesi/"

# Lista dei file nella cartella
files = os.listdir(folder_path)

# Percorso della cartella per salvare i risultati
output_folder_path = "/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/output_anamnesi/preprocessed_text"
os.makedirs(output_folder_path, exist_ok=True)

# Funzione per il preprocessing del testo
def preprocess_text(text):

    # Tokenizzazione del testo in frasi
    sentences = sent_tokenize(text)

    # Tokenizzazione del testo in parole e rimozione di punteggiatura e parole
    stop
    preprocessed_sentences = []
    for sentence in sentences:
        tokens = word_tokenize(sentence)
        tokens = [token.lower() for token in tokens if token not in
string.punctuation]
        tokens = [token for token in tokens if token not in
stopwords.words('italian')]
        preprocessed_sentence = ' '.join(tokens)
        preprocessed_sentences.append(preprocessed_sentence)

```

```

    return '\n'.join(preprocessed_sentences)

# Elabora ogni file nella cartella
for file_name in files:
    # Leggi il contenuto del file
    with open(os.path.join(folder_path, file_name), "r", encoding="utf-8") as file:
        text = file.read()

    # Preprocessing del testo
    preprocessed_text = preprocess_text(text)

    # Salva i risultati
    output_file_path = os.path.join(output_folder_path, file_name)
    with open(output_file_path, "w", encoding="utf-8") as output_file:
        output_file.write(preprocessed_text)

```

## 4.2 NER

Il Named Entity Recognition (NER) è il processo di identificazione e classificazione delle entità nominate presenti in un testo. Nel contesto dei testi clinici, le entità di interesse possono includere pazienti, casi clinici, malattie, farmaci, sintomi, procedure, esami complementari e altre informazioni rilevanti. Nel nostro progetto, abbiamo utilizzato il modello pre-addestrato **DeepMount00/universal\_ner\_ita** per eseguire il NER sui testi clinici.

I seguenti comandi sono stati lanciati per autenticarsi e accedere ai modelli pre-addestrati disponibili sulla piattaforma Hugging Face:

```

!pip install huggingface-hub
!huggingface-cli login

```

La scelta del modello pre-addestrato ha implicato l'installazione della libreria **GLiNER** (Generalized Language Interface for Named Entity Recognition):

```

!pip install gliner

```

Nel codice seguente vengono definiti i percorsi delle cartelle, caricato il modello pre-addestrato per il riconoscimento delle entità nominate, specifico per l'italiano. Viene definita una funzione per elaborare singoli file di testo che carica il testo, lo suddivide in parti più piccole, predice le entità nominate per ciascuna parte e salva le entità filtrate nel file di output. Questo processo è analogo per il riconoscimento delle entità nelle diagnosi.

```

import os
from gliner import GLiNER

# Percorsi delle cartelle
input_folder = "/content/drive/MyDrive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+dev+test+bg_240502/track2/cardioccc_dev/output_anamnesi/preprocessed_text"
output_folder = "/content/drive/MyDrive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+dev+test+bg_240502/track2/cardioccc_dev/output_anamnesi/ner"

# Carica il modello
model = GLiNER.from_pretrained("DeepMount00/universal_ner_ita")

# Etichette per NER
labels = ["paziente", "malattia", "procedura", "sintomo", "esame complementare", "farmaco"]

# Assicurati che la cartella di output esista
os.makedirs(output_folder, exist_ok=True)

# Funzione per processare un singolo file
def process_file(input_file_path, output_file_path):
    with open(input_file_path, 'r', encoding='utf-8') as file:
        text = file.read()

    # Dividi il testo in parti più piccole se necessario
    text_parts = [text[i:i + 500] for i in range(0, len(text), 500)]

    # Predici le entità per ogni parte
    all_entities = []
    for part in text_parts:
        entities = model.predict_entities(part, labels)
        all_entities.extend(entities)

    # Rimuovi duplicati e filtra entità non utili
    seen_texts = set()
    filtered_entities = []
    for entity in all_entities:
        if entity.text not in seen_texts:
            seen_texts.add(entity.text)
            filtered_entities.append(entity)

    # Scrivi le entità nel file di output
    with open(output_file_path, 'w', encoding='utf-8') as file:
        for entity in filtered_entities:
            file.write(f'{entity.text}\t{entity.type}\n')

```

```

text = entity["text"].strip()
label = entity["label"]

if text and text.lower() != label.lower() and text not in seen_texts:
    seen_texts.add(text)
    filtered_entities.append(entity)

# Salva le entità filtrate nel file di output
with open(output_file_path, 'w', encoding='utf-8') as file:
    for entity in filtered_entities:
        file.write(f"{entity['text']} => {entity['label']}\n")

# Elabora tutti i file nella cartella di input
for filename in os.listdir(input_folder):
    if filename.endswith(".txt"):
        input_file_path = os.path.join(input_folder, filename)
        output_file_path = os.path.join(output_folder, filename)
        process_file(input_file_path, output_file_path)

```

## 4.3 RE

L'estrazione delle relazioni (RE) è il processo di identificazione delle relazioni semantiche tra le entità riconosciute in un testo.

La funzione *extract\_relations(entities, case\_name)* estrae le relazioni tra le entità in base alle loro etichette. Per esempio, se trova un'entità "paziente" e un'entità "caso clinico", aggiunge una relazione "PRESENTA" tra di loro.

La funzione *parse\_entities\_from\_text(text)* analizza il testo di input e estrae le entità nominate insieme alle loro posizioni di inizio e fine nel testo.

In generale, il seguente codice itera attraverso tutti i file nella directory di input e per ogni file:

1. Estraie il nome del caso clinico basato sul nome del file e ne legge il contenuto.
2. Analizza il testo per estrarre le entità.
3. Aggiunge un'entità "caso clinico" al testo estratto.
4. Estraie le relazioni dalle entità.
5. Scrive il risultato nel file di output, contenente sia le entità che le relazioni estratte.

Il processo è analogo per il riconoscimento delle diagnosi.

```
import os
```

```

import re

# Funzione per estrarre le relazioni basate sulle etichette delle entità

def extract_relations(entities, case_name):

    relations = []

    # Aggiungi la relazione "ANAMNESI" tra "caso clinico" e "malattia"

    for entity in entities:

        if entity["label"] == "malattia":

            relations.append((case_name, "ANAMNESI", entity["text"]))

    for i in range(len(entities)):

        for j in range(i + 1, len(entities)):

            entity1 = entities[i]

            entity2 = entities[j]

            # Estrarre relazioni basate sulle etichette delle entità

            if entity1["label"] == "paziente" and entity2["label"] == "caso clinico":

                relations.append((entity1["text"], "PRESENTA", entity2["text"]))

            elif entity1["label"] == "malattia" and entity2["label"] == "farmaco":

                relations.append((entity1["text"], "RICHIEDE", entity2["text"]))

            elif entity1["label"] == "malattia" and entity2["label"] == "esame complementare":

                relations.append((entity1["text"], "CONFERMATA_DA", entity2["text"]))

            elif entity1["label"] == "malattia" and entity2["label"] == "sintomo":

                relations.append((entity1["text"], "CAUSA", entity2["text"]))

            elif entity1["label"] == "malattia" and entity2["label"] == "procedura":
```

```

        relations.append((entity1["text"], "TRATTATA_CON",
entity2["text"]))

return relations


# Funzione per convertire il testo in una lista di entità con posizioni di
inizio e fine

def parse_entities_from_text(text):

    entities = []

    entity_pattern = re.compile(r"(.*?) =>
(paziente|malattia|procedura|sintomo|esame complementare|farmaco)")

    for match in entity_pattern.finditer(text):

        entity_text = match.group(1).strip()

        entity_label = match.group(2).strip()

        start_pos = match.start(1)

        end_pos = match.end(1)

        entities.append({

            "text": entity_text,

            "label": entity_label,

            "start": start_pos,

            "end": end_pos

        })

    return entities


# Percorsi delle directory di input e output

input_dir = "/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/output_anamnesi/ner"

output_dir = "/content/drive/My
Drive/ProgettoBD/multicardioner_train+dev+test+bg_240502/multicardioner_train+de
v+test+bg_240502/track2/cardioccc_dev/output_anamnesi/re"

```

```
# Crea la directory di output se non esiste
os.makedirs(output_dir, exist_ok=True)

# Processa ogni file nella directory di input
for filename in os.listdir(input_dir):

    if filename.endswith(".txt"):

        input_file_path = os.path.join(input_dir, filename)
        output_file_path = os.path.join(output_dir, filename)

        # Nome del caso clinico basato sul nome del file
        case_name = filename.replace(".txt", "").replace("_", " ")

        # Leggi il contenuto del file di input
        with open(input_file_path, "r", encoding="utf-8") as input_file:
            input_text = input_file.read()

        # Parsing delle entità dal testo di input
        entities = parse_entities_from_text(input_text)

        # Aggiungi il caso clinico alle entità
        entities.append({
            "text": case_name,
            "label": "caso clinico",
            "start": 0,
            "end": 0
        })

        # Estrarre le relazioni dalle entità
```

```
relations = extract_relations(entities, case_name)

# Scrivi le relazioni estratte nel file di output

with open(output_file_path, "w", encoding="utf-8") as output_file:

    output_file.write("Entità:\n")

    for entity in entities:

        output_file.write(f"{entity['text']} => {entity['label']}\n")

    output_file.write("\nRelazioni:\n")

    for relation in relations:

        output_file.write(f"{relation[0]} - {relation[1]} -\n{relation[2]}\n")

print("Processamento completato.")
```

Infine, i dati vengono scritti in file CSV separati per entità e relazioni, per anamnesi e diagnosi.

# 5. Knowledge Graph

## 5.1 Processo di costruzione

Per costruire il Knowledge Graph del progetto, è stato utilizzato **Neo4j**.

Il primo passo è stato quello di creare un nuovo database denominato **MedicalDB**. Successivamente, si è scelto di **filtrare** il dataset ottenuto nelle fasi precedenti del progetto poiché Neo4j non permetteva di lavorare con un dataset di dimensioni così elevate. Sono stati caricati i file CSV filtrati contenenti le entità e le relazioni di 50 casi clinici. Di seguito viene descritto il processo di costruzione del grafo.

Per caricare le entità relative all'anamnesi e alla diagnosi, sono state utilizzate due query distinte. Queste creano nodi nel database, assegnando etichette specifiche in base al tipo di entità (ad esempio, sintomi, malattie, farmaci) utilizzando i dati dei file CSV. Di seguito la query realizzata per le entità di anamnesi (per la diagnosi si presenta in maniera analoga):

```
LOAD CSV WITH HEADERS FROM 'file:///entita_anamnesi.csv' AS row
MERGE (e {case_id: row.case_id, id: row.id, label: row.label})
SET e:`${row.label}`;
```

Per stabilire le relazioni tra le entità sono state utilizzate altre query. Queste caricano le relazioni specifiche tra le entità, stabilendo connessioni basate sulle relazioni definite nei file CSV come `PRESENTA`, `ANAMNESI`, `RICHIEDE`, `TRATTATA\_CON`, `CONFERMATA\_DA` e `CAUSA`. Di seguito la query realizzata per le relazioni di anamnesi (per la diagnosi si presenta in maniera analoga):

```
LOAD CSV WITH HEADERS FROM 'file:///relazioni_anamnesi.csv' AS row
MATCH (start {id: row.start})
MATCH (end {id: row.end})
WITH start, end, row.relationship AS relType
FOREACH (_ IN CASE WHEN relType = 'PRESENTA' THEN [1] ELSE [] END |
    MERGE (start)-[:PRESENTA]->(end))
FOREACH (_ IN CASE WHEN relType = 'ANAMNESI' THEN [1] ELSE [] END |
    MERGE (start)-[:ANAMNESI]->(end))
FOREACH (_ IN CASE WHEN relType = 'RICHIEDE' THEN [1] ELSE [] END |
    MERGE (start)-[:RICHIEDE]->(end))
FOREACH (_ IN CASE WHEN relType = 'TRATTATA_CON' THEN [1] ELSE [] END |
    MERGE (start)-[:TRATTATA_CON]->(end))
FOREACH (_ IN CASE WHEN relType = 'CONFERMATA_DA' THEN [1] ELSE [] END |
    MERGE (start)-[:CONFERMATA_DA]->(end))
FOREACH (_ IN CASE WHEN relType = 'CAUSA' THEN [1] ELSE [] END |
    MERGE (start)-[:CAUSA]->(end));
```

## 5.2 Esempio di grafi creati

Per la realizzazione del grafo del caso clinico n°41 è stata eseguita la seguente query:

```
MATCH (e {case_id: 'casos_clinicos_cardiologia41'})  
WHERE e.label IN ['paziente', 'malattia', 'procedura', 'sintomo', 'esame complementare', 'farmaco', 'caso clinico']  
  
OPTIONAL MATCH (e)-[r]->(related)  
  
WHERE related.case_id = 'casos_clinicos_cardiologia41'  
  
RETURN e, r, related;
```

In Figura 4 è possibile osservare come si presenta il Knowledge Graph:

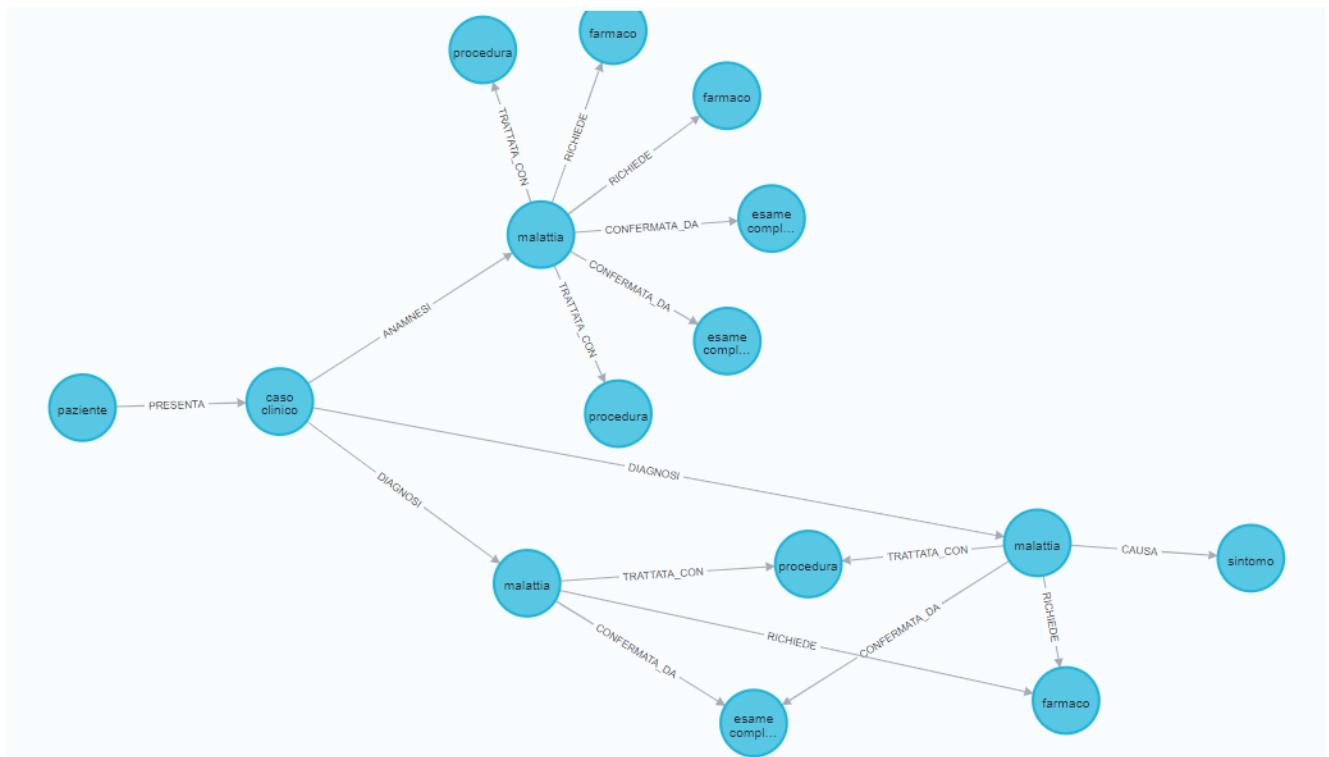


Figura 4: Knowledge Graph caso clinico n°41

# 6. Dashboard

## 6.1 Progettazione dell'interfaccia utente

Per il nostro progetto, abbiamo deciso di adottare un approccio user-centric nella progettazione dell'interfaccia utente, tenendo conto delle esigenze e delle aspettative degli utenti finali. Abbiamo realizzato un prototipo che illustra chiaramente le funzionalità principali e il flusso di lavoro previsto.



Figura 5: Prototipo dashboard home

## PAGINA CASO CLINICO SPECIFICO

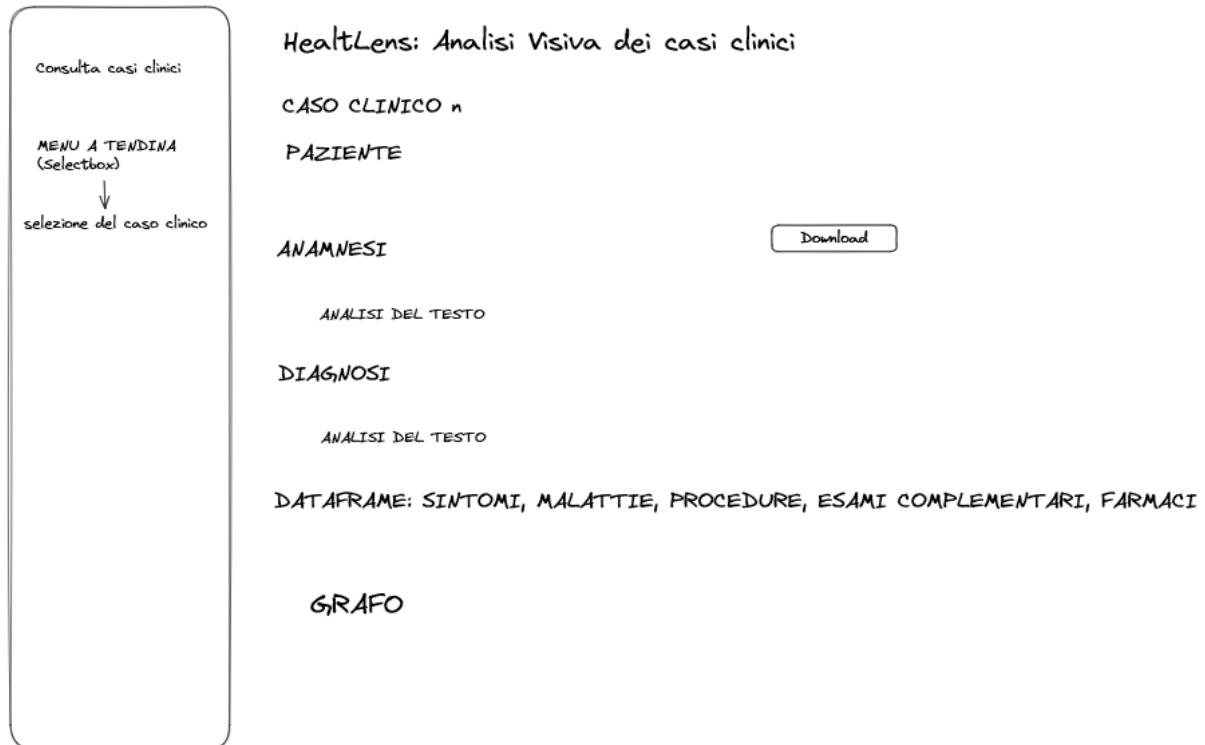


Figura 6: Prototipo dashboard caso clinico

## 6.2 Implementazione delle funzionalità

Il codice fornito implementa una dashboard interattiva con **Streamlit** per l'analisi e la visualizzazione di dati clinici. La struttura del progetto è suddivisa in diversi file per facilitare la gestione del codice e la sua manutenibilità. Le scelte di progetto sono state fatte per garantire una chiara separazione delle responsabilità e per rendere il codice modulare e riutilizzabile. Di seguito le informazioni per ogni file: medical.py, utils.py, requirements.txt, styles.css .

### MEDICAL.PY

Il file *medical.py* contiene la logica principale dell'applicazione Streamlit. È responsabile dell'interfaccia utente, della visualizzazione dei dati e dell'interazione con l'utente. Le funzionalità principali includono:

- Configurazione della pagina e applicazione di stili CSS personalizzati.

```
import streamlit as st
st.set_page_config(
    layout="wide", # Imposta il layout su "wide" per riempire lo spazio
    initial_sidebar_state="expanded" # Opzionale: espandi la barra laterale)
# Leggi il contenuto del file CSS
with open("styles.css", "r") as css_file:
```

```

css = css_file.read()

st.markdown(f"<style>{css}</style>", unsafe_allow_html=True)

• Connessione al database Neo4j per recuperare e visualizzare informazioni sui casi clinici.

# Inizializzazione del driver Neo4j

driver = get_driver("bolt://localhost:7687", "neo4j", "MedicalDB")

• Visualizzazione di metriche e grafici interattivi che mostrano la distribuzione dei casi clinici per malattia e farmaci utilizzati (per semplicità di seguito il codice inerente alle malattie).

with driver.session() as session:

    numero_di_casi_clinici_diabete =
    session.read_transaction(get_numero_di_casi_clinici_diabete)

with driver.session() as session:

    numero_di_casi_clinici_insuff =
    session.read_transaction(get_numero_di_casi_clinici_insuff)

with driver.session() as session:

    numero_di_casi_clinici_cardiomio =
    session.read_transaction(get_numero_di_casi_clinici_cardiomio)

with driver.session() as session:

    numero_di_casi_clinici_fibr =
    session.read_transaction(get_numero_di_casi_clinici_fibr)

with st.expander("Statistiche generiche", expanded=True):

    st.title("Casi Clinici e Malattie più frequenti")

    col1, col2, col3, col4 = st.columns(4)

    col1.metric("Diabete", f"{numero_di_casi_clinici_diabete}")

    col2.metric("Insufficienza Cardiaca", f"
{numero_di_casi_clinici_insuff}")

    col3.metric("Cardiomiopatia", f" {numero_di_casi_clinici_cardiomio}")

    col4.metric("Fibrillazione", f" {numero_di_casi_clinici_fibr}")

def get_case_data_summary_malattie():

    with driver.session() as session:

        numero_di_casi_clinici_diabete =
        session.read_transaction(get_numero_di_casi_clinici_diabete)

        numero_di_casi_clinici_insuff =
        session.read_transaction(get_numero_di_casi_clinici_insuff)

```

```

        numero_di_casi_clinici_cardiomio =
session.read_transaction(get_numero_di_casi_clinici_cardiomio)

        numero_di_casi_clinici_fibr =
session.read_transaction(get_numero_di_casi_clinici_fibr)

        return {
            "Diabete": numero_di_casi_clinici_diabete,
            "Insufficienza Cardiaca": numero_di_casi_clinici_insuff,
            "Cardiomiopatia": numero_di_casi_clinici_cardiomio,
            "Fibrillazione": numero_di_casi_clinici_fibr
        }

case_data = get_case_data_summary_malattie()

# Crea il grafico a torta
fig_malattia = px.pie(
    names=list(case_data.keys()),
    values=list(case_data.values()),
    title="Distribuzione dei Casi Clinici per Malattia"
)
fig_malattia.update_layout(title_font_color="#0000ff")

st.plotly_chart(fig_malattia)

```

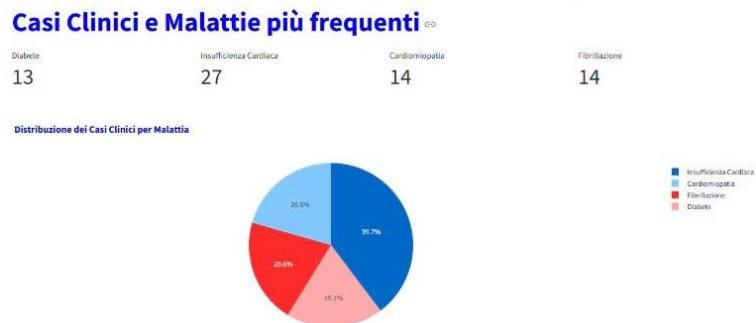


Figura 7: Grafico a torta "Malattie più frequenti"



Figura 8: Grafico a torta "Farmaci più frequenti"

- Creazione di una sidebar per la selezione dei casi clinici e la visualizzazione dettagliata delle informazioni correlate.

```
st.sidebar.title("Consulta casi clinici")

st.sidebar.caption("Esplora i casi clinici disponibili per visualizzare i dati relativi ai pazienti, alle loro condizioni di salute, alle malattie riscontrate, i farmaci prescritti, le procedure effettuate e gli esami complementari eseguiti.")

st.sidebar.divider()

# Percorso base per i file di testo

base_path = "./data"

# Ottenere i nomi dei casi clinici dalla cartella

name_path = "./data/anamnesi"

case_ids = [filename.replace(".txt", "") for filename in
os.listdir(name_path)]
```

# Creare il selectbox nella sidebar con i nomi dei casi clinici

```
case_id = st.sidebar.selectbox(
    "Caso clinico n°:",
    case_ids,
    index=None,
    placeholder="caso clinico")
```

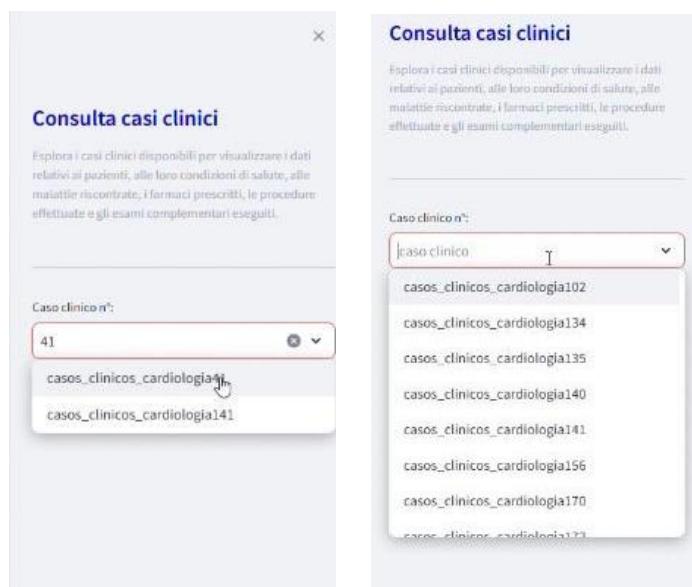


Figura 9: Sidebar

- Il codice verifica se il parametro `case_id` è presente. Se è presente, viene recuperato una serie di informazioni sul paziente, come farmaci, malattie, sintomi, procedure ed esami complementari.

```
if case_id:

    patient_info = get_patient_info(driver, case_id)

    medications = get_medications(driver, case_id)

    disease = get_disease(driver, case_id)

    symptoms = get_symptoms(driver, case_id)

    procedures = get_procedures(driver, case_id)

    exams = get_exams(driver, case_id)
```

- Viene visualizzato il titolo del caso clinico e le informazioni del paziente e consente il download dell'anamnesi e della diagnosi in un unico file.

```
# Mostra titolo e informazioni sul paziente

numero = re.search(r'\d+', case_id).group()

st.title(f"Caso clinico n°{numero}")

st.subheader("Paziente ⚡")

if patient_info:

    st.text(f"{patient_info}")

# Download caso clinico

anamnesi_content = load_text_file(base_path, case_id, "anamnesi")

diagnosi_content = load_text_file(base_path, case_id, "diagnosi")

st.download_button("Download", anamnesi_content + diagnosi_content)
```

## Caso clinico n°559

Paziente ⚡

uomo 63 anni

Download

Figura 10: Caso clinico, Paziente e button Download

- Questa sezione carica evidenzia e visualizza il testo dell'anamnesi, utilizzando annotazioni da un file CSV (analogo per diagnosi, non riportato per semplicità).

```
# Sezione Anamnesi

st.subheader("Anamnesi")

with st.expander("Dettagli"):

    def highlight_text(text, annotations):

        for word, label in annotations.items():

            css_class = label.replace(" ", "_").lower()

            text = re.sub(f"(?i)({{word}})", f'<span
class="{css_class}">\\1</span>', text)

        return text

entita_anamnesi_path = 'entita_anamnesi.csv'

entita_anamnesi_df = pd.read_csv(entita_anamnesi_path)

st.markdown(""""
<style>

.malattia { background-color: yellow; }

.procedura { background-color: lightpink; }

.esame_complementare { background-color: lightcoral; }

.farmaco { background-color: lightgreen; }

.sintomo { background-color: orange; }

.paziente { background-color: lightblue; }

</style>

""", unsafe_allow_html=True)

base_path = './data/anamnesi/'

case_ids = entita_anamnesi_df['case_id'].unique()
```

```

annotations_df = entita_anamnesi_df[entita_anamnesi_df['case_id'] == case_id]

text_file_path = os.path.join(base_path, f'{case_id}.txt')

if os.path.exists(text_file_path):

    with open(text_file_path, 'r', encoding='utf-8') as file:

        text = file.read()

        annotations = dict(zip(annotations_df['id'],
                               annotations_df['label']))

        highlighted_text = highlight_text(text, annotations)

        st.markdown(highlighted_text, unsafe_allow_html=True)

else:

    st.error(f'File {case_id}.txt non trovato in {base_path}')

```

### Anamnesi

**Dettagli**

**EVOLUZIONE** Data la presenza di un'infezione non controllata (**accesso aortico**), il **paziente** è stato sottoposto a un intervento di sostituzione valvolare. Prima dell'intervento, è stata eseguita una **PET/CT** per valutare il coinvolgimento del generatore di pacemaker e degli elettrocatereteri e per decidere se sostituire o meno il sistema di stimolazione. Visti i risultati dell'esame, si è deciso di non rimuovere il pacemaker. L'evoluzione postoperatoria è stata soddisfacente, il trattamento **antibiotico** con **ceftriaxone** è stato continuato per altri 20 giorni e gli **ecocardiogrammi** di controllo non hanno mostrato complicazioni. Durante le **visite cliniche successive**, il **paziente** non ha riferito febbre o dispnea.

### Diagnosi

**Dettagli**

**Esame fisico:** Pressione 145/62 mmHg; FC 92 bpm; SatO<sub>2</sub> 98% con aria ambiente, Ta 38,2°C. Cosciente e orientato. **Inurgito pluigolare**. Auscultazione cardiaca: ritmica senza soffi. Auscultazione polmonare: **crepitii bilabiali**. Addome: morbido, depressibile, non dolente alla palpazione, senza visceromegalia. Arti inferiori: **edema con fovea pretibiale ++**.

**ESAMI COMPLEMENTARI** **Emocolture:** 4 campioni positivi per **Streptococcus anginosus**. Ecodiagramma transtoracico e transesofageo: vegetazioni multiple a livello aortico (la più grande misura 10 x 8 mm) e un **spessimento disomogeneo**, perivalvolare e di aspetto ecodenso, corrispondente a un **accesso dell'anulus aortico** di 3,6 x 2,3 cm. **Tc cardiaca**: oltre a quanto sopra, sono stati osservati due piccoli pseudoaneurismi a livello subvalvolare. **18F-FDG PET/CT**: deposito patologico di attività intorno alla protesi aortica con maggiore intensità nella regione posteriore e assenza di captazione nel sistema di stimolazione cardiaca.

**DIAGNOSI** Endocardite infettiva tardiva su protesi aortica biologica, dovuta a **Streptococcus anginosus**, complicata da accesso perivalvolare aortico. Intervento di sostituzione valvolare. Portatore di **pacemaker bicompartimentale** a causa di un blocco atrioventricolare avanzato, senza evidenza di infezione del dispositivo.

Figura 11: Annotazioni Anamnesi e Diagnosi

- Visualizza un riepilogo combinato delle varie informazioni cliniche in una tabella.

```
st.subheader("Sintomi, Malattie, Procedure, Esami complementari e Farmaci")
```

```
symptoms_data = pd.DataFrame(symptoms)
```

```
disease_data = pd.DataFrame(disease)

procedures_data = pd.DataFrame(procedures)

exams_data = pd.DataFrame(exams)

medications_data = pd.DataFrame(medications)

# Inizializza colonne vuote se i DataFrame originali sono vuoti

if symptoms_data.empty:

    symptoms_data = pd.DataFrame({"id": [None]})

symptoms_data_id = symptoms_data[["id"]]

symptoms_data_id.columns = ["Sintomi 😷"]

if disease_data.empty:

    disease_data = pd.DataFrame({"id": [None]})

disease_data_id = disease_data[["id"]]

disease_data_id.columns = ["Malattie 💊"]

if procedures_data.empty:

    procedures_data = pd.DataFrame({"id": [None]})

procedures_data_id = procedures_data[["id"]]

procedures_data_id.columns = ["Procedure 🧟"]

if exams_data.empty:

    exams_data = pd.DataFrame({"id": [None]})

exams_data_id = exams_data[["id"]]

exams_data_id.columns = ["Esami complementari 🖌️"]

if medications_data.empty:

    medications_data = pd.DataFrame({"id": [None]})
```

```

medications_data_id = medications_data[["id"]]

medications_data_id.columns = ["Farmaci 🍯"]

combined_data = pd.DataFrame(index=symptoms_data_id.index)

combined_data = pd.merge(combined_data, symptoms_data_id, how='outer',
left_index=True, right_index=True)

combined_data = pd.merge(combined_data, disease_data_id, how='outer',
left_index=True, right_index=True)

combined_data = pd.merge(combined_data, procedures_data_id, how='outer',
left_index=True, right_index=True)

combined_data = pd.merge(combined_data, exams_data_id, how='outer',
left_index=True, right_index=True)

combined_data = pd.merge(combined_data, medications_data_id, how='outer',
left_index=True, right_index=True)

```

AgGrid(combined\_data, height=200)

### Sintomi, Malattie, Procedure, Esami complementari e Farmaci

Sintomi 🌱	Malattie 🌱	Procedure 🚑	Esami complementari 🧪	Farmaci 🍯
ingorgito glugolare	ascesso aortico	intervento sostituzione valvolare	pet/ct	antibiotice
crepitii bibasali	streptococcus anginosus	visite cliniche successive	ecocardiogrammi	ceftriaxone
edema	ascesso dell'anulus aortico	esame fisico	emocolture	pacemaker bicamerale
fovea pretibiale		auscultazione polmonare	tc cardiaca	
ispessimento		intervento sost		

Figura 12: Dataframe informazioni generali sul caso clinico

- Il codice finale costruisce e visualizza un grafo delle relazioni cliniche, utilizzando la libreria *neo4j*. La query viene eseguita per ottenere le relazioni tra vari nodi nel grafo, come pazienti, malattie, farmaci, sintomi, esami complementari e procedure. Viene chiamata la funzione *visualize\_graph* che mostra il grafo interattivo nel frontend Streamlit.

```

query = """
    /// Trova le relazioni di tipo ANAMNESI
    MATCH (p {label: 'paziente', case_id: $case_id})-[:PRESENTA]->(c {label: 'caso clinico', case_id: $case_id})-[:ANAMNESI]->(m {label: 'malattia', case_id: $case_id})

    WITH p, c, m

    OPTIONAL MATCH (m)-[rel:RICHIEDE | TRATTATA_CON | CONFERMATA_DA | CAUSA]->(n {case_id: $case_id})

    RETURN p, c, m, n,

```

```

'ANAMNESI' AS tipo_relazione,
CASE
    WHEN n.label = 'farmaco' THEN 'farmaco'
    WHEN n.label = 'sintomo' THEN 'sintomo'
    WHEN n.label = 'esame complementare' THEN 'esame complementare'
    WHEN n.label = 'procedura' THEN 'procedura'
END AS tipo_nodo,
TYPE(rel) AS tipo_relazione_specifica
UNION
/// Trova le relazioni di tipo DIAGNOSI
MATCH (p {label: 'paziente', case_id: $case_id})-[:PRESENTA]->(c {label: 'caso clinico', case_id: $case_id})-[:DIAGNOSI]->(m {label: 'malattia', case_id: $case_id})
WITH p, c, m
OPTIONAL MATCH (m)-[rel:RICHIEDE | TRATTATA_CON | CONFIRMATA_DA | CAUSA]->(n {case_id: $case_id})
RETURN p, c, m, n,
'DIAGNOSI' AS tipo_relazione,
CASE
    WHEN n.label = 'farmaco' THEN 'farmaco'
    WHEN n.label = 'sintomo' THEN 'sintomo'
    WHEN n.label = 'esame complementare' THEN 'esame complementare'
    WHEN n.label = 'procedura' THEN 'procedura'
END AS tipo_nodo,
TYPE(rel) AS tipo_relazione_specifica
"""
def visualize_graph(driver, query, case_id):
    with driver.session() as session:
        result = session.run(query, case_id=case_id)

        # Initialize dictionaries to store nodes and edges
        node_dict = {}
        edges = []

        # Define a color map for node types

```

```

color_map = {
    'farmaco': 'lightgreen',
    'sintomo': 'orange',
    'esame complementare': 'lightcoral',
    'procedura': 'lightpink'
}

for record in result:
    paziente = record["p"]
    caso_clinico = record["c"]
    malattia = record["m"]
    n = record["n"]
    tipo_nodo = record["tipo_nodo"]
    tipo_relazione_specifica = record["tipo_relazione_specifica"]

    # Controllo se il nodo n è None
    if n is None:
        continue

    n_color = color_map.get(tipo_nodo, 'gray')

    # Add nodes if they are not already added
    if paziente.id not in node_dict:
        node_dict[paziente.id] = (paziente["label"], "lightblue")

    if caso_clinico.id not in node_dict:
        node_dict[caso_clinico.id] = (caso_clinico["label"], "lightblue")

    if malattia.id not in node_dict:
        node_dict[malattia.id] = (malattia["label"], "yellow")

    if n and n.id not in node_dict:
        node_dict[n.id] = (tipo_nodo, n_color)

    # Add edges with relationship names
    edges.append((paziente.id, caso_clinico.id, "PRESENTA", "black"))

    edges.append((caso_clinico.id, malattia.id,
    record["tipo_relazione"], "black"))

```

```

if n:

    edges.append((malattia.id,    n.id,    tipo_relazione_specifica,
"black"))

# Create nodes and edges for agraph

agraph_nodes      =      [Node(id=node_id,          label=node_data[0],
color=node_data[1]) for node_id, node_data in node_dict.items()]

agraph_edges     = [Edge(source=edge[0],   target=edge[1],   label=edge[2],
color=edge[3]) for edge in edges]

config = Config(width=750, height=750, directed=True, physics=True)

return agraph(nodes=agraph_nodes, edges=agraph_edges, config=config)

```

Grafo

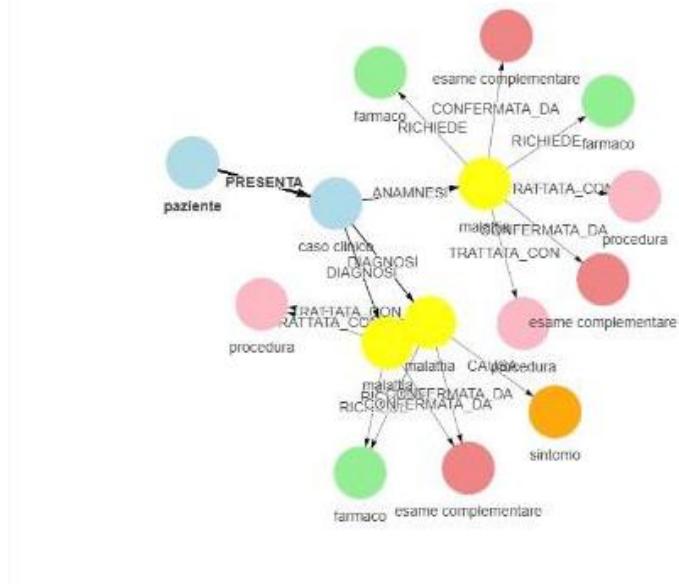


Figura 13: Grafo

## UTILS.PY

Il file *utils.py* raccoglie funzioni di utilità che supportano le operazioni del file principale. Queste funzioni includono:

- Connessione e gestione del driver Neo4j.

```

# Inizializzazione del driver Neo4j

def get_driver(uri, user, password):

    driver = GraphDatabase.driver(uri, auth=(user, password))

    return driver

```

- Recupero di informazioni sui pazienti, farmaci, malattie, sintomi, procedure ed esami complementari dai nodi del database Neo4j.

```

def get_patient_info(driver, case_id):
    query = """
        MATCH (p {case_id: $case_id})
        WHERE p.label IN ['paziente']
        RETURN p
    """

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        patient_info = result.single()

        return patient_info["p"].get("id", "Informazioni sul paziente non trovate.") if patient_info else "Informazioni sul paziente non trovate."


def get_medications(driver, case_id):
    query = """
        MATCH (m {case_id: $case_id})
        WHERE m.label IN ['farmaco']
        RETURN m
    """

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        medications = [dict(record["m"]) for record in result]
        return medications


def get_disease(driver, case_id):
    query = """
        MATCH (d {case_id: $case_id})
        WHERE d.label IN ['malattia']
        RETURN d
    """

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        disease = [dict(record["d"]) for record in result]
        return disease

```

```

def get_symptoms(driver, case_id):
    query = """
MATCH (s {case_id: $case_id})
WHERE s.label IN ['sintomo']
RETURN s
"""

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        symptoms = [dict(record["s"]) for record in result]
        return symptoms

def get_procedures(driver, case_id):
    query = """
MATCH (proc {case_id: $case_id})
WHERE proc.label IN ['procedura']
RETURN proc
"""

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        procedures = [dict(record["proc"]) for record in result]
        return procedures

def get_exams(driver, case_id):
    query = """
MATCH (e {case_id: $case_id})
WHERE e.label IN ['esame complementare']
RETURN e
"""

    with driver.session() as session:
        result = session.run(query, case_id=case_id)
        exams = [dict(record["e"]) for record in result]
        return exams

```

- Caricamento di contenuti da file di testo per visualizzare dettagli anamnestici e diagnostici.

```

def load_text_file(base_path, case_id, category):
    file_path = os.path.join(base_path, category, f"{case_id}.txt")
    try:
        with open(file_path, "r", encoding="utf-8") as file:
            content = file.read()
    return content
except FileNotFoundError:
    return "File non trovato."

```

- Funzioni per calcolare il numero di casi clinici associati a specifiche malattie o farmaci (per semplicità verranno mostrati un solo caso associato alle malattie, diabete, e un solo caso associato ai farmaci, cardioaspirina).

```

def get_numero_di_casi_clinici_diabete(tx):
    query = """
    MATCH (c {label: 'caso clinico'}) -[:ANAMNESI|DIAGNOSI]-> (m {label: 'malattia'})
    WHERE m.id STARTS WITH 'diabete'
    RETURN count(DISTINCT c.case_id) AS numero_di_casi_clinici
    """

    result = tx.run(query)
    return result.single()["numero_di_casi_clinici"]

def get_numero_di_casi_clinici_cardioaspirina(tx):
    query = """
    MATCH (c {label: 'caso clinico'}) -[:ANAMNESI|DIAGNOSI]-> (m {label: 'malattia'}) -[:RICHIEDE]-> (f {label: 'farmaco', id: 'cardioaspirina'})
    RETURN count(DISTINCT c.case_id) AS numero_di_casi_clinici
    """

    result = tx.run(query)
    return result.single()["numero_di_casi_clinici"]

```

## REQUIREMENTS.TXT

Il file *requirements.txt* specifica le dipendenze del progetto. Questo file è cruciale per garantire che tutte le librerie necessarie siano installate correttamente, consentendo una facile replicazione dell'ambiente di sviluppo.

```

streamlit==1.11.1
pandas==1.4.2
neo4j==4.4.6
py2neo==2021.2.3

```

```

pyvis==0.1.9
st-aggrid==0.3.3
matplotlib==3.5.2
seaborn==0.11.2

```

Infine, è presente il file *styles.css* che presenta le scelte di front-end effettuate per una visualizzazione più uniforme e pulita.

## 6.3 Visualizzazione dei dati

Per avviare il progetto in ambiente Streamlit sono stati eseguiti i seguenti passaggi:

- Accesso alla directory del progetto
- ```
cd C:\Users\traet\Desktop\ProgettoBD
```
- Attivazione dell'ambiente virtuale
- ```
.\venv\Scripts\Activate.ps1
```
- Esecuzione dell'applicazione Streamlit

```
streamlit run medical.py
```

Di seguito è possibile osservare come l'applicazione Streamlit si mostra agli utenti nella pagina di Home e in due casi clinici specifici.

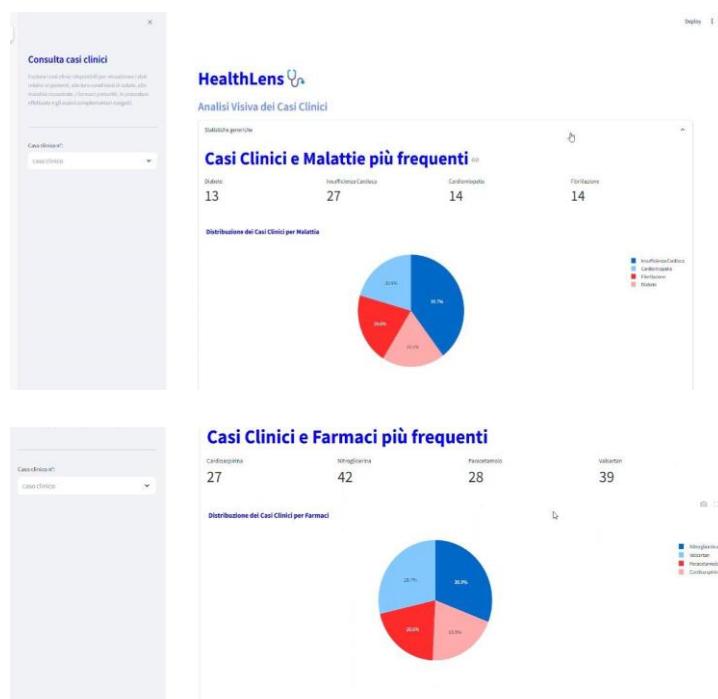


Figura 14: Home Streamlit

**Caso clinico n°41**

**Anamnesi**

**Diagnosi**

**Sintomi, Malattie, Procedure, Esami complementari e Farmaci**

**Grafo**

Figura 15: Caso clinico n°41 Streamlit

**Caso clinico n°559**

**Paziente** ♂

uomo - 63 anni

**Anamnesi**

**Diagnosi**

**Sintomi, Malattie, Procedure, Esami complementari e Farmaci**

Sintomi	Malattie	Procedure	Esami complementari	Farmaci
ipocinesia generalizzata	diabète mellito	follow-up		fer carbossimaltosio
	dislipidemia	test analitici		ferinject
	linfoma diffuso grandi cellule b	valutazione strutturale		ivtivi
	insufficienza cardiaca cronica	mappatura		ivtdvd
	cardiomiopatia dilatativa	trattamento con carbossimaltosio di ferro		ivtsid
	cardiomiopatia dilatata			ivel

<p><b>Consulta casi clinici</b></p> <p>Esplora i casi clinici disponibili per visualizzare i dati relativi ai pazienti, alle loro condizioni di salute, alle malattie riscontrate, i farmaci prescritti, le procedure effettuate e gli esami complementari eseguiti.</p> <hr/> <p>Caso clinico n°:</p> <p>casos_clinicos_cardiologia559</p>	<h2>Anamnesi</h2> <p>Dettagli</p> <p><b>ANAMNESI, MALATTIA ATTUALE ED ESAME FISICO</b></p> <p>Anamnesi: Fattori di rischio cardiovascolare (CVRF): diabete mellito, ex fumatore, dislipidemia. Linfoma diffuso a grandi cellule B, trattato con chemioterapia 10 anni fa, in remissione completa. Insufficienza cardiaca cronica: cardiomiopatia dilatativa non ischemica (DCM). LVEF 33%, ipocompenza generalizzata.</p> <p>Malattia attuale: Uomo di 63 anni in follow-up presso l'unità di insufficienza cardiaca del nostro ospedale per cardiomiopatia dilatativa di etiologia non ischemica. Il paziente è in NYHA II, senza segni di congestione periferica.</p> <p>EVOLUZIONE CLINICA: Secondo il protocollo, sono stati somministrati 1.000 mg di ferro carbossimato per via endovenosa (Ferinject®) ogni settimana per 2 settimane e, dopo aver confermato la normalizzazione del profilo del ferro con test analitici, è stata eseguita una nuova valutazione strutturale. CMR dopo 4 settimane di trattamento con ferro: ventricolo sinistro con spessore della parete e volume diastolico conservati (IVTDW 84 ml/m<sup>2</sup>; IVTSW 48 ml/m<sup>2</sup>). Funzione sistolica globale moderatamente depressa (LVEF 43%), secondaria a ipocompenza generalizzata. Ventricolo destro di morfologia e volumi conservati (IVTDW 65 ml/m<sup>2</sup>; IVTSW 30 ml/m<sup>2</sup>) e funzione sistolica borderline (LVEF 54%). Atrio sinistro di dimensioni normali: 34 mm, 23 cm<sup>2</sup>, 13 cm<sup>2</sup>/m<sup>2</sup>. Aorta e grandi vasi di diametro e morfologia normali. Miocardio T1: 1037 ms Miocardio T2*: 38 ms. C'è stato un miglioramento significativo della frazione di eiezione LV stimata dalla CMR (LVEF 43% a 30 giorni), con una diminuzione della mappatura T1 a 1037 ms, indicando la deplezione del ferro miocardico.</p>
---	--

Diagnosi	
Dettagli	
Sintomi, Malattie, Procedure, Esami complementari e Farmaci	
Sintomi	Malattie
dispiacente	test analitici
infusione di fibre grandi cellule b	valutazione strutturale
insufficienza cardiaca cronica	mapping
cardiomiopatia dilatativa	trattamento con carbossimaltosio di ferro
cardiomiopatia dilatata	ivef

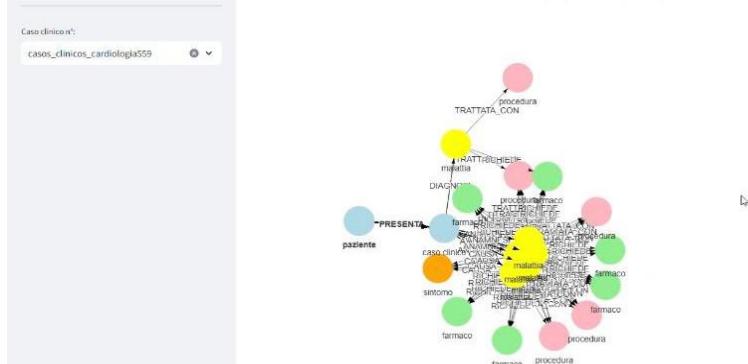


Figura 16: Caso clinico n°559 Streamlit

# 7. Requisito opzionale

## 7.1 Scelte progettuali

La traccia richiedeva un’ulteriore requisito opzionale da poter implementare, quello di trovare **“Pazienti simili”**. La realizzazione del requisito mira a fornire agli utenti la possibilità di individuare pazienti simili a un paziente selezionato, basandosi sia sulle malattie diagnostiche che sulle procedure diagnostiche presenti nel database.

Abbiamo scelto di effettuare i calcoli di similarità solo tra le **malattie** e le **procedure diagnostiche**, poiché sono state considerate come le entità di maggior rilevanza per eventuali confronti tra casi clinici.

Utilizzando **Streamlit** come interfaccia utente, il modulo consente agli utenti di selezionare un caso clinico di interesse e visualizzare una lista di altri casi clinici, ordinati in base alla loro similarità con il caso selezionato.

Per calcolare la similarità tra i casi clinici, il sistema utilizza un grafo incrementale costruito utilizzando **NetworkX** e le informazioni estratte dal database. NetworkX è una libreria Python che viene utilizzata per la creazione, manipolazione e studio della struttura, dinamica e funzioni delle reti complesse. Oltre alla funzionalità principale di individuazione dei pazienti simili, il modulo fornisce feedback all’utente nel caso in cui il caso selezionato non sia presente nel database.

## 7.2 Implementazione delle funzionalità

Il codice è stato interamente realizzato nel file *medical.py* poiché implementato successivamente al funzionamento dell’applicazione **Streamlit** nella sua versione “base”. Questo codice mostra un approccio per individuare casi clinici simili utilizzando due criteri: malattie diagnostiche e procedure diagnostiche.

- Importazione della libreria NetworkX

```
import networkx as nx
```

- Estrazione delle informazioni: viene eseguita una query per ottenere gli ID delle procedure associate a un dato caso clinico, viene calcolata la similarità tra le procedure di due casi clinici, vengono estratti tutti i case ID presenti nel database.

```
# Funzione per ottenere gli ID delle procedure

def get_procedure_info(driver, case_id):
    query = """
        MATCH (procedure {case_id: $case_id})
        WHERE procedure.label IN ['procedura']
        RETURN procedure.id AS id
    """

    result = driver.run_query(query, {"case_id": case_id})
    return [row["id"] for row in result]
```

```

"""
with driver.session() as session:
    result = session.run(query, case_id=case_id)
    procedure_ids = [record["id"] for record in result]
    return procedure_ids if procedure_ids else "Informazioni sulle
procedure non trovate."

# Funzione per calcolare la similarità basata sugli ID delle procedure
def calculate_procedure_similarity(selected_procedures,
other_procedures):
    similar_procedures = [proc for proc in selected_procedures if proc in
other_procedures]
    return len(similar_procedures), similar_procedures

# Funzione per ottenere tutti i case_id presenti nel database
def get_all_case_ids(driver):
    query = """
    MATCH (c {label: 'caso clinico'})
    RETURN DISTINCT c.case_id AS case_id
    """
    with driver.session() as session:
        result = session.run(query)
        case_ids = [record["case_id"] for record in result]
        return case_ids

```

- Caricamento del Grafo Incrementale: esegue una query per ottenere i pazienti, le malattie e le procedure diagnostiche dal database e costruisce un grafo NetworkX. Inoltre, il case ID selezionato viene assegnato alla variabile.

```

# Carica il grafo incrementale
def load_graph_incremental(driver):
    query = """
    MATCH (p {label: 'paziente'})-[:PRESENTA]->(c {label: 'caso
clinico'})-[:DIAGNOSI]->(m {label: 'malattia'})
    OPTIONAL MATCH (m)-[TRATTATA_CON]->(n {label: 'procedura'})
    RETURN p.case_id AS patient_id, n
    """
    with driver.session() as session:

```

```

        result = session.run(query)

        graph = nx.Graph()
        all_procedures = []
        for record in result:

            patient_id = record["patient_id"]
            n = record["n"]

            # Aggiungi nodo paziente
            graph.add_node(patient_id, label='paziente')

            # Aggiungi nodo procedura se esiste
            if n:
                n_id = n.id
                graph.add_node(n_id, label='procedura', id=n.id)
                graph.add_edge(patient_id, n_id, type="TRATTATA_CON")
                all_procedures.append(n_id)

        return graph, all_procedures

    # Carica il grafo incrementale
    graph, all_procedures = load_graph_incremental(driver)

```

- Vengono ottenute le malattie di un caso clinico, viene caricato il grafo. Se il case ID è presente, vengono estratte le malattie del caso clinico selezionato e calcolata la similarità con gli altri casi clinici basata sulle malattie. I risultati vengono ordinati in ordine decrescente di similarità e visualizzati in un DataFrame dove verranno visualizzate tutte le malattie generali degli altri casi.

```

    # Funzione per ottenere le malattie di un caso clinico
    def get_disease_info(driver, case_id):

        query = """
        MATCH (malattia {case_id: $case_id})
        WHERE malattia.label IN ['malattia']
        RETURN malattia.id AS id
        """

        with driver.session() as session:

            result = session.run(query, case_id=case_id)
            disease_ids = [record["id"] for record in result]

```

```

        return disease_ids if disease_ids else "Informazioni sulle
malattie non trovate."

# Carica il grafo incrementale
graph, all_diseases = load_graph_incremental(driver)

# Verifica che il case_id selezionato esista nel grafo
if selected_case_id not in graph:
    st.write(f"Il case_id {selected_case_id} non è presente nel grafo.")
else:
    st.subheader("Malattie simili al caso clinico in esame")
    st.write(f"Di seguito verranno riportati i casi clinici simili a
quello selezionato, quindi i pazienti simili, basandosi sulle malattie
diagnostiche.")

# Malattie del case_id specifico
selected_diseases = get_disease_info(driver, selected_case_id)

# Ottieni tutti i case_id presenti nel database
all_case_ids = get_all_case_ids(driver)

# Rimuovi il case_id selezionato dalla lista di tutti i case_id
all_case_ids.remove(selected_case_id)

# Calcola la similarità per ogni case_id
similarity_scores = []
for case_id in all_case_ids:
    other_diseases = get_disease_info(driver, case_id)
    similarity_score = len(set(selected_diseases) &
set(other_diseases))
    similarity_scores.append((case_id, similarity_score))

# Ordina i case_id in base al punteggio di similarità (in ordine
decrescente)
similarity_scores.sort(key=lambda x: x[1], reverse=True)

# Visualizza i casi clinici simili in un DataFrame

```

```

        df = pd.DataFrame(similarity_scores, columns=['Case ID', 'Numero di
Malattie Simili'])

        st.write("Casi clinici c simili (in ordine decrescente):")

        # Aggiungiamo le malattie generali dei casi clinici al DataFrame
        similar_diseases = [case_id for case_id, score in similarity_scores]

        df['Malattie generali del caso clinico'] = df['Case ID'].apply(lambda
x: ', '.join(get_disease_info(driver, x)) if x in similar_diseases else None)

        st.write(df)

```

- La logica è simile a quella delle malattie simili, ma stavolta si calcola la similarità basandosi sulle procedure diagnostiche. Vengono estratte le procedure del caso clinico selezionato e calcolata la similarità con gli altri casi clinici basata sulle procedure. I risultati vengono ordinati e visualizzati in un DataFrame, dove verranno visualizzate, questa volta, solo le procedure simili degli altri casi.

```

if selected_case_id not in graph:

    st.write(f"Il case_id {selected_case_id} non è presente nel grafo.")

else:

    st.subheader("Procedure simili al caso clinico in esame")

    st.write(f"Di seguito verranno riportati i casi clinici simili a
quello selezionato, basandosi sulle procedure diagnostiche.")

    # Procedure del case_id specifico
    selected_procedures = get_procedure_info(driver, selected_case_id)

    # Ottieni tutti i case_id presenti nel database
    all_case_ids = get_all_case_ids(driver)

    # Rimuovi il case_id selezionato dalla lista di tutti i case_id
    all_case_ids.remove(selected_case_id)

    # Calcola la similarità per ogni case_id
    similarity_scores = []
    for case_id in all_case_ids:

        other_procedures = get_procedure_info(driver, case_id)
        similarity_score, similar_procedures =
calculate_procedure_similarity(selected_procedures, other_procedures)

        similarity_scores.append((case_id, similarity_score,
similar_procedures))

```

```

# Ordina i case_id in base al punteggio di similarità (in ordine
decrescente)

similarity_scores.sort(key=lambda x: x[1], reverse=True)

st.write("Casi clinici simili (in ordine decrescente):")

df = pd.DataFrame(similarity_scores, columns=['Case ID', 'Numero di
Procedure Simili', 'Procedure Simili'])

st.write(df)

```

## 7.3 Visualizzazione dei dati

Di seguito è possibile osservare come l'applicazione Streamlit mostra agli utenti il modulo “Pazienti simili” per un caso clinico di interesse (caso n°596).

### Pazienti simili

#### Malattie simili al caso clinico in esame

Di seguito verranno riportati i casi clinici simili a quello selezionato, quindi i pazienti simili, basandosi sulle malattie diagnostiche.

Casi clinici simili (in ordine decrescente):

	Case ID	Numero di Malattie Simili	Malattie generali del caso clinico
0	casos_clinicos_cardiologia272	4	ipertensione arteriosa, ipercolesterolemia, diabete mellito tipo 2, sindrome apnea-ip
1	casos_clinicos_cardiologia280	4	ipertensione, dislipidemia, diabete mellito tipo ii, retinopatia diabetica, obesità patolog
2	casos_clinicos_cardiologia359	4	aneurisma dell'aod, patologia, disfunzione sistolica, dissezione aortica tipo b, patolog
3	casos_clinicos_cardiologia357	4	ipertensione arteriosa, dislipidemia, iperuricemia, insufficienza reumatica aortica, er
4	casos_clinicos_cardiologia170	4	dislipidemia, diabete mellito, ipertensione arteriosa, stenosi aortica, asma, trombosi
5	casos_clinicos_cardiologia135	3	cardiopatia ischemica, infarto silenzioso, infarto miocardico, artensione arteriosa, dis
6	casos_clinicos_cardiologia140	3	diabetico tipo 2, insufficienza aortica, mediastinite, protesi aortica, insufficienza mitr
7	casos_clinicos_cardiologia191	3	cardiomiopatia dilatativa, dislipidemia, disfunzione sисто, lica biventricolare ischemic
8	casos_clinicos_cardiologia3	3	tempesta aritmica, ipertensione arteriosa, dislipidemia, diabete mellito tipo 2, malattia
9	casos_clinicos_cardiologia98	2	diabete mellito secondario, pancreatite cronica, malattia castleman, varici esofagee,

#### Procedure simili al caso clinico in esame

Di seguito verranno riportati i casi clinici simili a quello selezionato, basandosi sulle procedure diagnostiche.

Casi clinici simili (in ordine decrescente):

	Case ID	Numero di Procedure Simili	Procedure Simili
0	casos_clinicos_cardiologia135	4	interventi chirurgici appendicectomia tonsillectomia ecocardiogramma
1	casos_clinicos_cardiologia280	4	interventi chirurgici ecocardiogramma auscultazione cardiaca auscultazione polmonare follow-up
2	casos_clinicos_cardiologia309	4	ecocardiogramma auscultazione cardiaca auscultazione polmonare follow-up
3	casos_clinicos_cardiologia92	3	interventi chirurgici auscultazione cardiaca auscultazione polmonare
4	casos_clinicos_cardiologia282	3	auscultazione cardiaca auscultazione polmonare esami laboratorio
5	casos_clinicos_cardiologia59	3	ecocardiogramma auscultazione cardiaca auscultazione polmonare
6	casos_clinicos_cardiologia526	3	ecocardiogramma auscultazione cardiaca auscultazione polmonare
7	casos_clinicos_cardiologia357	3	interventi chirurgici auscultazione cardiaca auscultazione polmonare
8	casos_clinicos_cardiologia170	3	interventi chirurgici auscultazione cardiaca auscultazione polmonare
9	casos_clinicos_cardiologia81	2	ecocardiogramma follow-up

Figura 17: Caso clinico n°596 "Pazienti simili"

# 8. Risultati

## 8.1 Benefici ottenuti

L'implementazione del sistema descritto potrebbe portare a benefici significativi nel contesto dei big data nel settore sanitario. L'estrazione automatizzata delle informazioni cliniche dalle note mediche non strutturate consente una riduzione del tempo necessario per la raccolta e l'analisi dei dati, migliorando l'efficienza operativa del personale sanitario e permettendo loro di concentrarsi maggiormente sull'assistenza diretta ai pazienti.

Il Knowledge Graph costruito con Neo4j ha fornito una base per una rappresentazione strutturata e facilmente interrogabile delle relazioni complesse tra i vari concetti medici. Questo potrebbe portare ad un miglioramento della precisione diagnostica e della comprensione delle condizioni dei pazienti, contribuendo a una migliore pianificazione dei trattamenti. La dashboard interattiva sviluppata con Streamlit ha offerto un'interfaccia intuitiva per visualizzare e analizzare i dati clinici, che potrebbe portare ad un miglioramento della comunicazione tra i professionisti della salute, supportando decisioni cliniche più informate.

L'utilizzo di NetworkX per individuare pazienti simili ha rappresentato una base di un ulteriore valore aggiunto, permettendo di identificare pattern e casi clinici comparabili, il che può portare a trattamenti personalizzati e a una migliore gestione dei pazienti con condizioni simili.

## 8.2 Limiti riscontrati

Nonostante i numerosi benefici, il progetto ha incontrato alcuni limiti significativi. Primo fra tutti, la **scalabilità** del sistema. È stato necessario filtrare il dataset di partenza da 250 casi clinici a 50, poiché Neo4j non riusciva a gestire dimensioni così elevate. Questo ha comportato una limitazione nel trarre conclusioni globali su dataset più ampi.

La libreria NetworkX, pur essendo potente, ha mostrato limiti di performance quando applicata a dataset molto grandi; tuttavia, non abbiamo filtrato ulteriormente il dataset ma abbiamo costruito grafi incrementali.

Un altro limite riscontrato riguarda la qualità e la completezza dei dati di input. La presenza di dati mancanti o di annotazioni errate nei documenti clinici ha influenzato la precisione dell'estrazione delle entità e delle relazioni, richiedendo interventi manuali per la correzione e la validazione dei risultati.

Infine, l'adozione di modelli di linguaggio pre-addestrati ha presentato sfide in termini di adattamento al linguaggio specifico e alle terminologie utilizzate nelle note cliniche italiane, evidenziando la necessità di ulteriori personalizzazioni e miglioramenti dei modelli NLP utilizzati.

## **8.3 Futuri sviluppi e ricerche**

Per superare i limiti riscontrati e migliorare ulteriormente il sistema, sono previste diverse direzioni di sviluppo futuro e di ricerca. Un obiettivo chiave sarà l'integrazione di tecnologie di big data più avanzate per gestire dataset di dimensioni ancora maggiori e migliorare la scalabilità e le performance del sistema di analisi dei grafi.

Un'altra area di sviluppo sarà il miglioramento dei modelli di linguaggio utilizzati, attraverso l'addestramento su dataset specifici del dominio medico italiano e l'integrazione di tecniche di machine learning avanzate per migliorare la precisione del Named Entity Recognition e della Relation Extraction.

Inoltre, si prevede di integrare il dataset DrugTEMIST per arricchire il corpus con informazioni sui farmaci, ampliando così le capacità di estrazione e analisi delle informazioni cliniche. DrugTEMIST, combinato con DisTEMIST, fornirà una visione più completa e dettagliata delle condizioni mediche e dei trattamenti, migliorando ulteriormente la qualità del Knowledge Graph.

Infine, sarà importante condurre studi longitudinali e ricerche cliniche per valutare l'impatto del sistema sulle pratiche mediche e sulla qualità della cura dei pazienti, al fine di ottimizzare ulteriormente gli strumenti sviluppati e garantire il massimo beneficio per i professionisti della salute e i pazienti stessi.