

**Corso di Laurea Magistrale
in
Ingegneria Informatica**

**Big Data Analytics
and Business Intelligence
Elaborato d'esame**

prof. Giancarlo Sperlì

Cardiological Examinations Graph

M63000989 Fabio d'Andrea
M63000986 Guido Di Chiara
M63001040 Antimo Iannucci



Università degli Studi di Napoli Federico II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
Anno Accademico 2020/2021
Secondo Semestre

Indice

1	Introduzione	1
1.1	Business Understanding	1
1.2	Data Understanding	3
2	Preprocessing	4
2.1	Data Cleaning	4
2.1.1	VISITA_CONTROLLO_ECG	4
2.1.2	VISITA_CARDIOLOGICA	8
2.2	Merging	9
2.2.1	Correzione colonna ESAMINATORE	10
2.3	Data Preparation	11
2.3.1	Aggiunta ID_VISITA	11
2.3.2	Preparazione dati Visita	11
2.3.3	Preparazione Anamnesi e Diagnosi	12
2.3.4	Preparazione Farmaci	12
2.3.5	Correzione Farmaci misspelled	13
3	Named Entity Recognition (NER)	17
3.1	Introduzione al problema	17
3.1.1	Modello utilizzato	17
3.2	Training	19
3.2.1	Preprocessing	19
3.2.2	Data Analysis	21
3.2.3	Data Preparation	21

3.2.4	Training	22
3.2.5	Evaluation	24
3.3	Inferenza	25
3.3.1	Anamnesi	26
3.3.2	Diagnosi	27
4	Graph-based Database	28
4.1	Modello dei dati	28
4.2	Implementazione	29
4.2.1	Postprocessing delle entità	29
4.2.2	Nodi	30
4.2.3	Relazioni	32
5	Data Visualization	36
5.1	PowerBI	36
5.1.1	Caricamento dei dati	36
5.1.2	Modello dei dati	38
5.1.3	Dashboard Medici	40
5.1.4	Dashboard Pazienti	42
5.1.5	Dashboard Sintomi, Malattie e Farmaci	44
5.2	Neo4j Bloom	46
5.2.1	Query Pazienti	46
5.2.2	Query Medici	55
5.2.3	Query su informazioni generali	58

Capitolo 1

Introduzione

La seguente trattazione illustra il lavoro svolto per l'elaborato finale del corso di *Big Data Analytics and Business Intelligence* presso l'*Università degli Studi di Napoli Federico II*. Il task dell'elaborato consiste nell'applicazione delle conoscenze acquisite durante il corso, al fine di estrapolare alcune informazioni cliniche a partire da dati semi strutturati riguardanti visite cardiologiche e rappresentare le entità estratte in un graph-based database.

1.1 Business Understanding

Le fasi di manipolazione, analisi, memorizzazione e visualizzazione dei dati a disposizione possono essere organizzate in una pipeline composta da quattro stadi riportata in Figura 1.1.

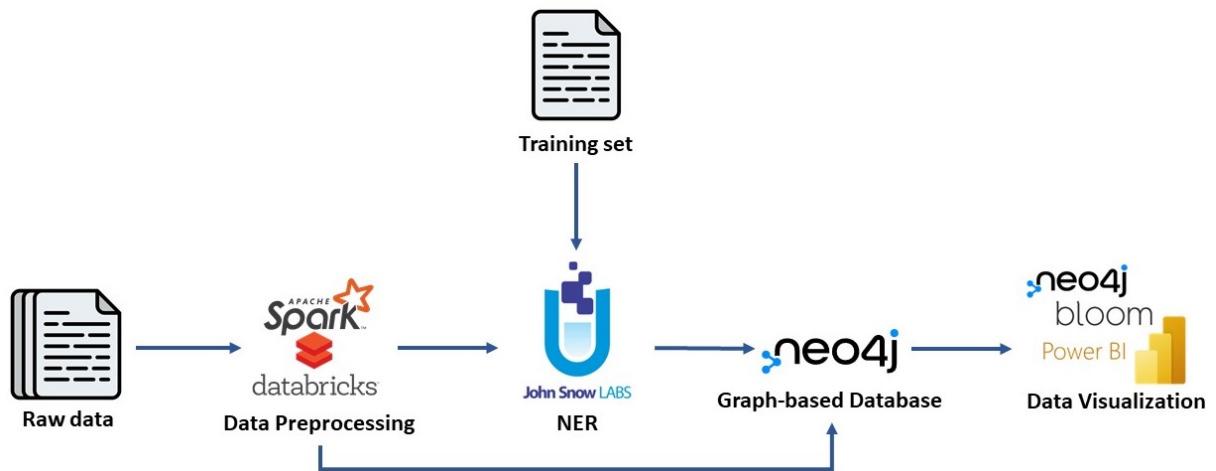


Figura 1.1: Pipeline di gestione dei dati

Di seguito si riportano le operazioni realizzate ad ogni stadio:

- **Data preprocessing:** tale stadio prende in ingresso i dati grezzi relativi agli esami cardiologici effettuati dai pazienti e si occupa di trasformarli opportunamente

per facilitarne l'elaborazione negli stadi successivi. In particolare, si occupa di organizzare i dati non strutturati e strutturati in file differenti.

- **Named Entity Recognition (NER)**: tale stadio prende in ingresso dei dati di training, necessari per addestrare un modello basato su *Deep Learning (DL)* che sia in grado di riconoscere sintomi e malattie all'interno di testi scritti in linguaggio naturale. Il modello ottenuto viene poi applicato alle anamnesi e diagnosi provenienti dallo stadio precedente della pipe.
- **Graph-based database**: tale stadio prende in ingresso i dati strutturati provenienti dalla fase di preprocessing e le informazioni in uscita dal modello NER per organizzarli in un modello dei dati coerente ed efficiente. Il grafo costruito interconnette opportunamente pazienti, medici, anamnesi e diagnosi con i relativi farmaci assunti, malattie e sintomi.
- **Data visualization**: in tale stadio sono utilizzati strumenti di *Business Intelligence (BI)* per la visualizzazione dei dati risultanti dalle elaborazioni effettuate, con l'obiettivo di mostrare la conoscenza estratta dai dati di partenza.

Per quanto concerne gli strumenti impiegati nella gestione del ciclo di vita dei dati, lo step di data processing è stata condotto mediante **Apache Spark**, noto framework impiegato per il calcolo distribuito, ampiamente utilizzato in applicazioni di Big Data. Nello specifico, è stato utilizzato **PySpark**, APIs Python che permette di scrivere applicazioni Spark. Il modulo principalmente utilizzato è **Spark SQL**, il quale permette di processare dati strutturati in modo distribuito, fornendo un'astrazione di alto livello chiamata *DataFrame*.

Per fare uso delle tecniche di *Natural Language Processing (NLP)* sui dati non strutturati si è fatto ricorso alla libreria open-source SparkNLP, realizzata da **John Snow Labs**. Essa rappresenta lo stato dell'arte per applicazioni di NLP in ambienti distribuiti e offre tutta una serie di modelli di intelligenza artificiale e pipeline di NLP pre-addestrati, oltre alla possibilità di addestrare ulteriormente i modelli sui propri dati.

L'addestramento e l'inferenza del modello di NER è stato eseguito su **Google Colab**, piattaforma cloud che offre gratuitamente la possibilità di utilizzare una GPU dotata di 12GB di RAM. Ciò ha permesso di velocizzare notevolmente i tempi di esecuzione, rispetto alla scelta di impiegare semplici CPU. L'elaborazione dei dati è stata invece effettuata principalmente sulla piattaforma cloud **Databricks**, in versione community. Essa, fondata dagli stessi creatori di Apache Spark, rappresenta un ambiente web-based che fornisce gestione automatica di cluster per l'esecuzione di calcolo distribuito ed un ambiente integrato di sviluppo di notebook Python.

Per la memorizzazione dei dati si è scelto di utilizzare **Neo4j**, un database No-SQL open-source di tipo grafo, in grado di modellare i dati come insiemi di nodi e archi. La scelta di un database graph-based permette di rappresentare al meglio le relazioni esistenti tra le diverse entità del dominio studiato (pazienti, malattie, sintomi, etc.). Nello specifico, è stata utilizzata una versione locale di Neo4j, e i dati elaborati su Databricks sono stati direttamente salvati sul database, attraverso un opportuno connettore.

Il passo finale di data visualization è stato invece realizzato in **PowerBI**, il quale fornisce funzionalità di BI con un'interfaccia semplice ed intuitiva, permettendo di creare

dashboard e report. Tale step è stato arricchito dall'utilizzo di **Neo4j Bloom**, un'applicazione di esplorazione dei grafi facile da usare che permette di interagire visualmente con i grafi in Neo4j.

Il codice relativo alle elaborazioni effettuate è contenuto in diversi file `.ipynb`, reperibili sul repository GitHub [1].

1.2 Data Understanding

Nella sezione corrente si vogliono descrivere con maggiore precisione i dati elaborati, i quali possono essere divisi in due categorie:

- **Dati visite:** informazioni relative agli esami cardiologici svolti da diversi pazienti, organizzati in due file, denominati rispettivamente `VISITA_CONTROLLO_ECG.csv` e `VISITACARDIOLOGICA.csv`, i quali contengono sia informazioni strutturate che non strutturate. Il primo file, composto da 2125 entry e 66 colonne, risulta essere pulito e ben organizzato, mentre il secondo, composto da 151267 entry e 227 colonne, ha richiesto una fase di preprocessing più accentuata, in quanto più confusionario e disorganizzato.
- **Dati training NER:** dati contenuti nel file `training_set.txt`, necessari per l'addestramento del modello di NER. I dati sono rappresentati nel **formato BIO (Begin-Inside-Outside)**; ogni entry include quindi una parola, o meglio un *token*, e la relativa etichetta, la quale può assumere i seguenti valori:
 - *B-entity*: se il token coincide con l'inizio di un'entità;
 - *I-entity*: se il token si trova all'interno di un'entità;
 - *O*: se il token non è un'entità.

Le specifiche entità da riconoscere sono *Disease* (malattia) e *Symptom* (sintomo).

Capitolo 2

Preprocessing

Nel capitolo corrente sono riportate tutte le operazioni di preprocessing effettuate in **PySpark** sui dati, al fine di facilitarne l'elaborazione negli stadi successivi.

2.1 Data Cleaning

In questa sezione sono riportate le prime operazioni effettuate sui dati grezzi presenti nei file `VISITA_CONTROLLO_ECG.csv` e `VISITACARDIOLOGICA.csv`. Tutte le operazioni sono contenute nel notebook `data_cleaning_e_merging.ipynb` e coincidono con le seguenti fasi del **processo di big data**:

- *Acquisizione*: si acquisiscono i dati dai file grezzi
- *Estrazione*: si applicano una serie di operazioni di trasformazione e standardizzazione dei dati a disposizione
- *Integrazione*: si integrano i dati trasformati che a questo punto possiedono uno schema ben definito

2.1.1 VISITA_CONTROLLO_ECG

Si procede con l'illustrazione degli step di preprocessing che hanno caratterizzato il file `VISITA_CONTROLLO_ECG.csv`.

2.1.1.1 Selezione colonne di interesse

Il file in esame è stato dapprima caricato in un DataFrame denominato `df_ecg`, sul quale si è proceduto a considerare soltanto le seguenti colonne di interesse:

- CODPAZ: identificativo numerico univoco, relativo al paziente sottoposto alla visita.
- ETA: età del paziente.
- PESO: peso del paziente.

- ALTEZZA: altezza del paziente.
- DATA_EVENTO: data in cui si è svolta la visita.
- COMMENTO: campo testuale contenente la diagnosi effettuata dal medico al paziente in seguito alla visita.
- COD_MEDICO_FIRMANTE: identificativo univoco relativo al medico che esegue la visita.
- FARMACO*i*_FINALE: 10 campi testuali contenenti i farmaci prescritti dal medico al paziente in seguito alla visita.
- DOSE_FARMACO*i*_FINALE: 10 campi testuali contenenti i dosaggi dei rispettivi farmaci prescritti dal medico al paziente in seguito alla visita.

A causa della scarsa qualità dei dati contenuti nel file in esame, accade che vi siano alcune inversioni nei nomi delle colonne di interesse. Contestualmente alla selezione delle colonne, si è proceduto a risolvere tale problematica.

Selezione colonne di interesse

```
columns = ["CODPAZ", "ETA", "FUMOSTRAT1", "PESO", "ALTEZZA", "DATA_EVENTO", "COMMENTO",
           "CHECK_FI", "FARMACO1_FINAL", "FARMACO2_FINAL", "FARMACO3_FINAL", "FARMACO4_FINAL",
           "FARMACO5_FINAL", "FARMACO6_FINAL", "FARMACO7_FINAL", "FARMACO8_FINAL",
           "FARMACO9_FINAL", "FARMACO10_FINAL", "DOSE_FARMACO1_FINAL", "DOSE_FARMACO2_FINAL",
           "DOSE_FARMACO3_FINAL", "DOSE_FARMACO4_FINAL", "DOSE_FARMACO5_FINAL",
           "DOSE_FARMACO6_FINAL", "DOSE_FARMACO7_FINAL", "DOSE_FARMACO8_FINAL",
           "DOSE_FARMACO9_FINAL", "DOSE_FARMACO10_FINAL"]

df_ecg = df_ecg.select(columns) \
    .withColumnRenamed("ETA", "SESSO") \
    .withColumnRenamed("FUMOSTRAT1", "ETA") \
    .withColumnRenamed("CHECK_FI", "COD_MEDICO_FIRMANTE")
display(df_ecg)
```

2.1.1.2 Sostituzione elementi vuoti con *null*

Dall'analisi dei dati all'interno del DataFrame è emersa la presenza di elementi vuoti (caratterizzati da uno spazio), i quali sono stati opportunamente rimpiazzati con il valore *null*.

Sostituzione elementi vuoti con *null*

```
for column in df_ecg.columns:
    df_ecg = df_ecg.withColumn(column, when(df_ecg[column] == " ", None) \
        .otherwise(df_ecg[column]))
```

2.1.1.3 Rimozione entry non significative

Sono state rimosse tutte le entry del DataFrame che possiedono valori nulli nei campi di maggiore interesse, quali COMMENTO, fondamentale per la successiva fase di NLP, ed il campo FARMACO1, in quanto la mancanza di esso implica la mancanza di tutti i restanti farmaci.

Rimozione entry non significative

```
df_ecg = df_ecg.where(~((col("COMMENTO").isNull() & (col("FARMACO1_FINE") .isNull()))))
```

2.1.1.4 Correzione formati colonne

Per favorire le fasi successive, si è reso necessario correggere i formati di alcune colonne, tra cui CODPAZ e ETA, convertiti a *Integer*, e DATA_EVENTO, trasformato da stringa al formato *date*.

Correzione formati colonne

```
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")
df_ecg = df_ecg.withColumn("CODPAZ", df_ecg["CODPAZ"].cast(IntegerType()) ) \
    .withColumn("ETA", df_ecg["ETA"].cast(IntegerType()) ) \
    .withColumn("DATA_EVENTO", to_date(col("DATA_EVENTO"), "dd/MM/yyyy HH:mm:ss"))
df_ecg = df_ecg.dropna(subset=[ "CODPAZ", "DATA_EVENTO"])
```

Contestualmente, a valle della conversione di formato, si rimuovono alcune entry che non presentano valori per le colonne CODPAZ e DATA_EVENTO, necessarie invece durante la fase di memorizzazione dei dati nel database.

2.1.1.5 Correzione colonna SESSO

Si è proceduto a ripulire i valori contenuti nella colonna SESSO, in quanto tale colonna è caratterizzata da rumorosità, come la presenza di alcuni valori numerici o misspelling. Per risolvere tale problematica, si è fatto in modo da ridurre i valori ammissibili di tale colonna a *M*, *F* e *null*.

Correzione colonna SESSO

```
df_ecg = df_ecg.withColumn('SESSO', when((df_ecg.SESSO == "Maschio"), "M") \
    .when((df_ecg.SESSO == "maschio"), "M") \
    .when((df_ecg.SESSO == "Femmina"), "F") \
    .when((df_ecg.SESSO == "M") | (df_ecg.SESSO == "F"), df_ecg.SESSO) \
    .otherwise(None))
```

Il formato della colonna originaria ed risultato finale con le suddette correzioni è riportato in Figura 2.1:

SESSO	count
1 F	13130
2 null	111298
3 M	20066
4 55	1
5 Maschio	712
6 Femmina	345
7 maschio	1

(a) Colonna SESSO prima
del processing

SESSO	count
1 F	13475
2 null	111299
3 M	20779

(b) Colonna SESSO dopo il
processing

Figura 2.1: Colonna SESSO

2.1.1.6 Correzione colonna ETA

La colonna ETA presenta alcune incongruenze, probabilmente legate a errori di digitazione da parte del medico. Si è quindi scelto di limitare il range di valori ammissibili tra 0 e 100, impostando a *null* tutti i restanti valori.

Correzione colonna ETA

```
df_ecg = df_ecg.withColumn("ETA", \
    when((df_ecg.ETA >= 0) & (df_ecg.ETA <=100), df_ecg.ETA) \
    .otherwise(None))
```

2.1.1.7 Correzione colonna PESO

La colonna PESO presenta alcune incongruenze, probabilmente legate a errori di digitazione. In questo caso si è scelto di limitare il range di valori ammissibili di peso tra 25 e 250, impostando a *null* tutti i restanti valori.

Correzione colonna PESO

```
df_ecg = df_ecg.withColumn("PESO", \
    when((df_ecg.PESO < 25) | (df_ecg.PESO >250), None) \
    .otherwise(df_ecg.PESO))
```

2.1.1.8 Correzione colonna ALTEZZA

La colonna ALTEZZA presenta delle incongruenze legate a possibili errori di digitazione o a differenti formati di scrittura dell'altezza. Si è proceduto a moltiplicare per 100 tutti i valori di altezza minori di 2, in quanto probabilmente essi sono espressi in metri, mentre sono stati impostati a *null* tutti i restanti valori non compresi nell'intervallo 20-250.

Correzione colonna ALTEZZA

```
df_ecg = df_ecg.withColumn("ALTEZZA", \
    when((df_ecg.ALTEZZA < 2), df_ecg.ALTEZZA*100) \
    .when((df_ecg.ALTEZZA >= 2) & (df_ecg.ALTEZZA < 20), None) \
    .when((df_ecg.ALTEZZA > 250), None) \
```

```
.otherwise(df_ecg.ALTEZZA))
```

2.1.1.9 Correzione colonna COD_MEDICO_FIRMANTE

La colonna COD_MEDICO_FIRMANTE presenta delle incongruenze legate a possibili errori di digitazione o al concatenamento di valori di colonne adiacenti. Si è proceduto quindi a considerare solo i codici con lunghezza compresa tra 4 e 6 caratteri, mentre sono stati impostati a *null* tutti i codici che non soddisfano tale condizione.

Correzione colonna COD_MEDICO_FIRMANTE

```
df_ecg = df_ecg.withColumn("COD_MEDICO_FIRMANTE", \  
    when((length(df_ecg.COD_MEDICO_FIRMANTE) < 4) | \  
        (length(df_ecg.COD_MEDICO_FIRMANTE) > 6), None) \  
    .otherwise(df_ecg.COD_MEDICO_FIRMANTE))
```

2.1.2 VISITA_CARDIOLOGICA

Si procede con la descrizione degli step di preprocessing che hanno caratterizzato il file VISITA_CARDIOLOGICA.csv.

2.1.2.1 Selezione colonne di interesse

Il file in esame è stato dapprima caricato in un DataFrame denominato df_card, sul quale si è proceduto a considerare soltanto le seguenti colonne di interesse:

- CODPAZ: identificativo numerico univoco, relativo al paziente sottoposto alla visita.
- DATA_EVENTO: data in cui si è svolta la visita.
- ANAMNESI: campo testuale contenente l'anamnesi del paziente nel contesto della visita.
- DIAGNOSI: campo testuale contenente la diagnosi effettuata dal medico al paziente in seguito alla visita.
- ESAMINATORE: stringa indicante il nome del medico che esegue la visita.
- COD_MEDICO_FIRMANTE: identificativo univoco relativo al medico che esegue la visita.
- FARMACO*i*: 10 campi testuali contenenti i farmaci prescritti dal medico al paziente in seguito alla visita.
- DOSE_FARMACO*i*: 10 campi testuali contenenti i dosaggi dei rispettivi farmaci prescritti dal medico al paziente in seguito alla visita.

Selezione colonne di interesse

```
columns = ["CODPAZ", "DATA_EVENTO", "ANAMNESI", "DIAGNOSI", "ESAMINATORE",
"COD_MEDICO_FIRMANTE", "FARMACO1", "FARMACO2", "FARMACO3", "FARMACO4",
"FARMACO5", "FARMACO6", "FARMACO7", "FARMACO8", "FARMACO9", "FARMACO10",
"DOSE_FARMACO1", "DOSE_FARMACO2", "DOSE_FARMACO3", "DOSE_FARMACO4",
"DOSE_FARMACO5", "DOSE_FARMACO6", "DOSE_FARMACO7", "DOSE_FARMACO8",
"DOSE_FARMACO9", "DOSE_FARMACO10"]

df_card = df_card.select(columns)

display(df_card)
```

2.1.2.2 Rimozione entry non significative

Sono state rimosse tutte le entry del DataFrame che possiedono valori *null* nei campi di maggiore interesse, quali ANAMNESI e DIAGNOSI, fondamentali per la successiva fase di NLP, ed il campo FARMACO1, in quanto la mancanza di esso implica la mancanza di tutti i restanti farmaci.

Rimozione entry non significative

```
df_card = df_card.where(~((col("ANAMNESI").isNull() & (col("DIAGNOSI").isNull() & (col("FARMACO1").isNull()))))
```

2.1.2.3 Correzione formato colonna DATA _ EVENTO

Per favorire le fasi successive, si è reso necessario correggere il formato della colonna DATA _ EVENTO da stringa al formato *date*.

Correzione formato DATA _ EVENTO

```
df_card = df_card.withColumn("DATA_EVENTO", to_date(col("DATA_EVENTO"), "dd/MM/yyyy"))

conteggio = df_card.select([count(when(col(c).isNull(), c)).alias(c) for c in df_card.columns])
```

Contestualmente, a valle della conversione di formato, si verifica che non vi siano valori nulli per le colonne CODPAZ e DATA _ EVENTO, necessarie invece durante la fase di memorizzazione dei dati nel database.

2.2 Merging

Successivamente alla fase di cleaning, è stata realizzata la fase di Merging dei due DataFrame in un unico DataFrame, standardizzando le loro colonne nel rispetto del contenuto semantico. Tali operazioni sono contenute nel notebook *data_cleaning_e_merging.ipynb*.

Merging

```
ecg = ['CODPAZ', 'SESSO', 'ETA', 'PESO', 'ALTEZZA', 'DATA_EVENTO', 'COD_MEDICO_FIRMANTE',
'COMMENTO', 'FARMACO1_FINE', 'FARMACO2_FINE', 'FARMACO3_FINE', 'FARMACO4_FINE',
'FARMACO5_FINE', 'FARMACO6_FINE', 'FARMACO7_FINE', 'FARMACO8_FINE',
'FARMACO9_FINE', 'FARMACO10_FINE', 'DOSE_FARMACO1_FINE', 'DOSE_FARMACO2_FINE',
```

```
'DOSE_FARMACO3_FINELE', 'DOSE_FARMACO4_FINELE', 'DOSE_FARMACO5_FINELE',
'DOSE_FARMACO6_FINELE', 'DOSE_FARMACO7_FINELE', 'DOSE_FARMACO8_FINELE',
'DOSE_FARMACO9_FINELE', 'DOSE_FARMACO10_FINELE']

card = ["CODPAZ", "DATA_EVENTO", "ANAMNESI", "DIAGNOSI", "ESAMINATORE",
"COD_MEDICO_FIRMANTE", "FARMACO1", "FARMACO2", "FARMACO3", "FARMACO4", "FARMACO5",
"FARMACO6", "FARMACO7", "FARMACO8", "FARMACO9", "FARMACO10", "DOSE_FARMACO1",
"DOSE_FARMACO2", "DOSE_FARMACO3", "DOSE_FARMACO4", "DOSE_FARMACO5", "DOSE_FARMACO6",
"DOSE_FARMACO7", "DOSE_FARMACO8", "DOSE_FARMACO9", "DOSE_FARMACO10"]

df_ecg = df_ecg.select(ecg).withColumnRenamed("COMMENTO", "DIAGNOSI") \
.withColumn("ANAMNESI", lit(None)).withColumn("ESAMINATORE", lit(None))

df_card = df_card.select(card) \
.withColumn("SESSO", lit(None)) \
.withColumn("ETA", lit(None)) \
.withColumn("PESO", lit(None)) \
.withColumn("ALTEZZA", lit(None))

for i in range(1,11):
    col_name = 'FARMACO'+ str(i) + '_FINELE'
    df_ecg = df_ecg.withColumnRenamed(col_name, "FARMACO" + str(i))
    col_name = 'DOSE_FARMACO'+ str(i) + '_FINELE'
    df_ecg = df_ecg.withColumnRenamed(col_name, "DOSE_FARMACO" + str(i))

df_card = df_card.select(df_ecg.columns)
df = df_ecg.union(df_card)
```

2.2.1 Correzione colonna ESAMINATORE

A valle dell'operazione di unione dei due dataset di partenza, si osserva che la colonna ESAMINATORE, riportante il nome del medico che ha eseguito la visita, presenta alcune informazioni superflue, come la presenza del titolo del medico stesso o di codici alfanumerici. Contestualmente, si è proceduto a correggere la presenza di esaminatori senza il nome solo in specifiche entry del dataset, a partire dalle entry in cui tale informazione è invece presente; in aggiunta, viene corretta la presenza di codici differenti per lo stesso medico esaminatore.

Correzione colonna ESAMINATORE

```
split_col = split(df.ESAMINATORE, ' ', limit=2)
split_col = split(split_col.getItem(1), ' [A-Z][A-Z][.]| [A-Z][A-Z]\d', limit=2)
df = df.withColumn('ESAMINATORE', split_col.getItem(0))

df_med = df.select("ESAMINATORE", "COD_MEDICO_FIRMANTE") \
.where(~df.ESAMINATORE.isNull() & ~df.COD_MEDICO_FIRMANTE.isNull()) \
.withColumnRenamed("ESAMINATORE", "ESAMINATORE_NEW") \
.distinct() \
.dropDuplicates(subset=['ESAMINATORE_NEW'])

df_join = df.join(df_med, on="COD_MEDICO_FIRMANTE", how="left_outer")
df = df_join.drop("ESAMINATORE").withColumnRenamed("ESAMINATORE_NEW", "ESAMINATORE")
```

Il formato della colonna originaria ed il risultato finale con le suddette correzioni è riportato in Figura 2.2:

ESAMINATORE	
1	Dott.ssa Cinzia Perrino
2	Dr. Aniello Viggiano CE6447
3	Dr. Annibale Pepe
4	Dr. Carmine Morisco
5	Dr. Eugenio Stabile
6	Dr. Eugenio Stabile OO. MM SA008442
7	Dr. Francesco Borgia OO. MM. n°. NA030456
8	Dr. Gerardo Carpinella
9	Dr. Giacomo Mattiello
10	Dr. Gianni Luigi Iovino OO. MM. n° NA024621

ESAMINATORE	
1	Aniello Viggiano
2	Annibale Pepe
3	Antonio Rapacciulo
4	Carmine Morisco
5	Cinzia Perrino
6	Emanuele Barbato
7	Eugenio Stabile
8	Francesco Borgia
9	Gerardo Carpinella
10	Giacomo Mattiello

(a) Colonna ESAMINATORE
prima del processing

(b) Colonna ESAMINATORE dopo il processing

Figura 2.2: Colonna ESAMINATORE

2.3 Data Preparation

In questa sezione si procede alla preparazione dei dati contenuti nel DataFrame completo appena generato durante la fase di Merging, al fine di ottimizzare la successiva fase di NER e di creazione del database. Tali operazioni sono contenute nel notebook `data_preparation.ipynb`.

2.3.1 Aggiunta ID_VISITA

Il DataFrame completo `df_completo` precedentemente generato viene arricchito di un identificativo intero univoco `ID_VISITA`, relativo alla specifica visita medica, portando alla creazione del DataFrame `df`. Tale operazione si è resa necessaria per favorire la successiva fase di creazione del database a grafo.

Aggiunta ID_VISITA

```
df = df_completo.select(df_completo.columns)
df = df.withColumn("ID_VISITA", monotonically_increasing_id())
```

2.3.2 Preparazione dati Visita

A partire dal DataFrame `df`, si procede ad estrarre le sole informazioni strutturate relative alla visita, ad esclusione dei farmaci per cui si è reso necessario un ulteriore step di processing. Tale operazione porta alla creazione del DataFrame `df_visita`, il quale viene salvato nel file `DF_VISITA_COMPLETO.csv`.

Preparazione dati Visita

```
df_visita = df.select('ID_VISITA', 'CODPAZ', 'SESSO', 'ETA', 'PESO',
                      'ALTEZZA', 'DATA_EVENTO', 'ESAMINATORE', 'COD_MEDICO_FIRMANTE')
display(df_visita)
```

2.3.3 Preparazione Anamnesi e Diagnosi

Analogamente, si procede alla creazione del DataFrame `df_anam_dia`, estrapolando le sole informazioni non strutturate relative alla visita medica, oltre all'identificativo della visita stessa. Tale DataFrame viene quindi salvato nel file `DF_ANAM_DIA_COMPLETO.csv`, su cui sarà poi effettuata la NER.

Preparazione Anamnesi e Diagnosi

```
df_anam_dia = df.select("ID_VISITA", "ANAMNESI", "DIAGNOSI")
display(df_anam_dia)
```

2.3.4 Preparazione Farmaci

La preparazione dei farmaci ha richiesto una fase più spinta di processing, in quanto tali campi testuali risultano essere fortemente rumorosi a causa di misspelling, dosaggi e informazioni superflue specificanti la tipologia del farmaco (compresse, gocce, spray, gel, cerotto, ecc.). Per risolvere tale problematica, si è innanzitutto proceduto a convertire tutte le diciture in upper case, per poi eliminare tutte le informazioni superflue appena descritte. In questo modo si ripulisce la stragrande maggioranza delle diciture, cosicché esse contengano esclusivamente il nome del farmaco prescritto. In questo modo sono state generate delle nuove colonne `FARMACO`, seguite dalla dicitura '`CLEAN`'.

Cleaning Farmaci

```
for i in range(1,11):
    colonna = 'FARMACO' + str(i)
    colonna_new = 'FARMACO' + str(i) + '_CLEAN'
    df = df.withColumn(colonna_new, upper(col(columna))) \
    .withColumn(colonna_new, regexp_replace(colonna_new, ' MG|\\d+MG| MCG|\\d+MCG| MGR| \\\d+MGR|\\d+GR| GR| CPR| COMPRESSE| CEROTTO| CEROTTI| GOCCE| GEL| GTT| BUST| FL| UI| SPRAY|\\d+CPR| CP|\\d+CP|[^a-zA-Z0-9_ ]|\\d+', ''))
    df = df.withColumn(colonna_new, regexp_replace(colonna_new, '^[\t]+|[ \t]+\$', ''))
    df = df.withColumn(colonna_new, '[\t]+', ''))
```

Un esempio del formato delle colonne originarie ed risultato finale con le suddette correzioni è riportato in Figura 2.3:

FARMACO1		FARMACO1_CLEAN
1	ADALAT 60	ADALAT
2	ENAPREN 20 mg	BIVIS
3	RAMIPRIL 5	COUMADIN
4	Norvasc 10 mg	ENAPREN
5	Lamictal 50 mg cp	INSULINA HUMULIN
6	OLPREZIDE40/12.5	LAMICTAL
7	Insulina Humulin 30/70	NORVASC
8	NORVASC 5	NORVASC
9	COUMADIN 5 mg	OLPREZIDE
10	BIVIS 40/10	RAMIPRIL

(a) Colonna FARMACO
prima del processing

(b) Colonna FARMACO
dopo il processing

Figura 2.3: Colonna FARMACO

2.3.5 Correzione Farmaci misspelled

La correzione dei misspelling è stata eseguita a partire dalle colonne dei farmaci appena ripulite. In primo luogo, si è proceduto ad unire tutte le colonne dei farmaci in un unico DataFrame union, successivamente ripulito dei valori *null* e dei duplicati, in modo da visualizzare il numero totale di farmaci distinti contenuti nel dataset complessivo. Tale verifica ha portato al conteggio di 4888 farmaci differenti.

Preparazione farmaci distinti (`union`)

```
union = df.withColumn("farmaci", array('FARMACO1_CLEAN', 'FARMACO2_CLEAN',
'FARMACO3_CLEAN', 'FARMACO4_CLEAN', 'FARMACO5_CLEAN', 'FARMACO6_CLEAN',
'FARMACO7_CLEAN', 'FARMACO8_CLEAN', 'FARMACO9_CLEAN', 'FARMACO10_CLEAN')) \
.select("farmaci").withColumn("farmaci", explode("farmaci"))

union = union.where(~union.farmaci.isNull()).where(union.farmaci != "").distinct()
```

A questo punto si è proceduto ad utilizzare una *ground truth* contenente le nomenclature corrette dei farmaci, reperita dal sito web dell'Agenzia Italiana del Farmaco [2]. Nello specifico, si è fatto uso delle liste relative ai farmaci di Classe A e di Classe H. A partire da tali liste, sono stati estrapolati esclusivamente i nomi corretti dei farmaci, per poi unire il tutto in un unico DataFrame denominato `farmaci_gt`, il quale conta 3210 farmaci distinti.

Preparazione ground truth

```
split_col = split(farmaci_gtA['Denominazione e Confezione'], '\n')
farmaci_gtA = farmaci_gtA.withColumn('Farmaco', split_col.getItem(0)) \
.select('Farmaco').distinct()
split_col = split(farmaci_gtH['Denominazione e Confezione'], '\n')
farmaci_gtH = farmaci_gtH.withColumn('Farmaco', split_col.getItem(0)) \
.select('Farmaco').distinct()
farmaci_gt = farmaci_gtA.select('Farmaco').union(farmaci_gtH.select('Farmaco')) \
.distinct().withColumnRenamed('Farmaco', 'farmaci_corretti')
```

Lo scopo dell'utilizzo di tale ground truth risiede nell'intento di voler sostituire i farmaci misspelled contenuti nel dataset con i nomi corretti dei farmaci contenuti nella ground truth, in modo da selezionare per ogni farmaco misspelled il rispettivo farmaco corretto "più simile". La metrica di similarità utilizzata per stimare le differenze tra i nomi dei farmaci è la **distanza di Levenshtein**, una misura in grado di determinare quanto due stringhe siano simili tra loro. In realtà, si è fatto uso del Levenshtein Distance Ratio, il quale presenta valori compresi in $[0, 1]$ e cresce all'aumentare della similarità tra le stringhe s_1 e s_2 .

$$LDR(s_1, s_2) = \frac{\text{len}(s_1) \cdot \text{len}(s_2) - D_{\text{Levenshtein}}}{\text{len}(s_1) \cdot \text{len}(s_2)} \in [0, 1]$$

Nel dettaglio, calcolando il matching in termini di distanza di Levenshtein tra tutti i farmaci del dataset completo (`union`) e tutti i farmaci corretti della ground truth (`farmaci_gt`), si otterrebbe un numero di combinazioni, tramite un'operazione di *cross join*, pari a $4888 \times 3210 = 15'690'480$. Tuttavia, osservando che la correzione dei farmaci deve avvenire esclusivamente per gli effettivi misspelling e non per tutti i farmaci, si è dapprima calcolato l'*inner join* tra `union` e `farmaci_gt` in modo da ottenere il DataFrame `gt`, con i farmaci effettivamente corretti del dataset, per poi sottrarre tale insieme di farmaci ad `union` tramite un'operazione di *anti join*: in tal modo si è ottenuto il DataFrame `anti` di 3877 entry, rappresentanti i misspelling distinti contenuti nel dataset completo. A questo punto, il cross join tra i misspelling di `anti` e la ground truth ridotta `gt` presenta $3877 \times 1011 = 3'919'647$ combinazioni, scalando la complessità computazionale circa di un fattore 5.

Preparazione groud truth ridotta e cross join

```
gt = union.join(farmaci_gt.withColumnRenamed('farmaci_corretti', 'farmaci'),
                 on='farmaci', how='inner')
anti = union.join(gt, on="farmaci", how='anti')
cross = anti.select("farmaci").withColumnRenamed('farmaci', 'farmaci_misspelled') \
    .crossJoin(gt.withColumnRenamed("farmaci", "farmaci_corretti")) \
    .select('farmaci_corretti')
```

In realtà, un limite di tale approccio è rappresentato dalla situazione in cui uno specifico farmaco è presente nel dataset completo solo con dei misspelling: con il procedimento messo in atto non siamo in grado di correggere tale nome, in quanto la riduzione della ground truth tramite l'*inner join* con `union` comporta la presenza in `gt` dei soli farmaci scritti correttamente almeno una volta nel nostro dataset. Ciononostante, essendo il vantaggio computazionale particolarmente significativo, si è scelto comunque di seguire questo approccio.

La correzione dei misspelling continua con il calcolo della distanza di Levenshtein per ogni combinazione scaturita dal cross join, per poi considerare come possibili correzioni solo i match con un Levenshtein Distance Ratio maggiore di 0.9. Tali match sono stati memorizzati nel DataFrame `similarity_09`.

Calcolo Levenshtein Distance Ratio

```
similarity = cross.withColumn("result",
                               levenshtein(cross.farmaci_misspelled, cross.farmaci_corretti))
similarity = similarity.withColumn("result", similarity["result"].cast("float"))
similarity_09 = similarity.filter(similarity.result >= 0.9)
```

A questo punto potrebbe accadere che un farmaco abbia più corrispondenze con punteggio di similarità maggiore di 0.9, pertanto è necessario considerare solo le corrispondenze col valore massimo di punteggio. La considerazione delle sole entry col massimo punteggio porta ad ottenere la tabella delle corrispondenze che è rappresentata dal DataFrame mapping, riportato in Figura 2.4.

Generazione tabella delle corrispondenze

```
temp = similarity_09.groupBy("farmaci_misspelled").max("result")
mapping = temp.join(similarity_09.withColumnRenamed("result", "max(result)"),
on=["max(result)", "farmaci_misspelled"]).orderBy("farmaci_misspelled")
```

max(result)	farmaci_misspelled	farmaci_corretti
1	AATORVASTATINA	ATORVASTATINA
2	ABSORCORL	ABSORCOL
3	ACCURETC	ACCURETIC
4	ACEQUID	ACEQUIDE
5	ACESITEM	ACESISTEM
6	ACIFDO FOLICO	ACIDO FOLICO
7	ACRIPTIN	ASCRIPPTIN
8	ACUPRIN	ACCUPRIN
9	ACURETC	ACCURETIC
10	ADALA CRON	ADALAT CRONO

Figura 2.4: Tabella delle corrispondenze

A questo punto non resta che sostituire i nomi dei farmaci nel dataset per i quali si è individuato un misspelling. Per prima cosa, sono state selezionate dal DataFrame df solo le colonne riguardanti i farmaci, i dosaggi e l'ID della visita, memorizzando il risultato in df_farmaci. Tale DataFrame è stato successivamente manipolato in modo tale da ottenere uno schema del tipo ID_VISITA - FARMACO - CONFEZIONE - DOSE, ottenendo di fatto una entry per ogni farmaco prescritto di una specifica visita, come mostrato in Figura 2.5.

Dopodiché si effettua un *left outer join* tra df_farmaci e mapping in modo da poter facilmente sostituire alla colonna FARMACO di df_farmaci il nome corretto laddove si individui una corrispondenza. A valle di questa operazione si ottiene il DataFrame in Figura 2.6, il quale viene salvato nel file DF_FARMACI_COMPLETO.csv.

Correzione dei misspelling

```
df_farmaci = df.select(df.columns)
df_farmaci = df_farmaci.drop("ANAMNESI", "DIAGNOSI", "CODPAZ",
                             "COD_MEDICO_FIRMANTE", "DATA_EVENTO")

df_zip = df_farmaci.withColumn("FARMACO", array('FARMACO1_CLEAN',
                                                 'FARMACO2_CLEAN', 'FARMACO3_CLEAN', 'FARMACO4_CLEAN',
                                                 'FARMACO5_CLEAN', 'FARMACO6_CLEAN', 'FARMACO7_CLEAN',
                                                 'FARMACO8_CLEAN', 'FARMACO9_CLEAN', 'FARMACO10_CLEAN')) \
    .withColumn("CONFEZIONE", array('FARMACO1', 'FARMACO2',
                                    'FARMACO3', 'FARMACO4', 'FARMACO5', 'FARMACO6', 'FARMACO7',
                                    'FARMACO8', 'FARMACO9', 'FARMACO10')) \
```

```

.withColumn("DOSE", array('DOSE_FARMACO1', 'DOSE_FARMACO2',
    'DOSE_FARMACO3', 'DOSE_FARMACO4', 'DOSE_FARMACO5',
    'DOSE_FARMACO6', 'DOSE_FARMACO7', 'DOSE_FARMACO8',
    'DOSE_FARMACO9', 'DOSE_FARMACO10')) \
.select("ID_VISITA", "FARMACO", "CONFEZIONE", "DOSE")

df_farmaci = df_zip.withColumn("tmp", arrays_zip("FARMACO", "CONFEZIONE", "DOSE")) \
.withColumn("tmp", explode("tmp")) \
.select("ID_VISITA", "tmp.FARMACO", "tmp.CONFEZIONE", "tmp.DOSE") \
.dropna(subset="FARMACO")

temp = df_farmaci.join(mapping.withColumnRenamed("farmaci_misspelled", "FARMACO"),
    on="FARMACO", how="left_outer")

finale = temp.withColumn("FARMACO", when(temp.farmaci_corretti.isNull(),
    temp["FARMACO"]) \
    .otherwise(temp["farmaci_corretti"])) \
    .withColumnRenamed("FARMACO", "FARMACO_FINELE")

df_farmaci = finale.drop("max(result)", "farmaci_corretti").drop_duplicates()

```

ID_VISITA	FARMACO	CONFEZIONE	DOSE
1 0	RATACAND	RATACAND 16 mg	1/2 al mattino
2 0	LOPRESOR	LOPRESOR	1/2 al mattino e alla sera
3 0	ARMOLIPID	ARMOLIPID	un/una alla sera
4 0	MAALOX	MAALOX 400mg	ogni 8 ore
5 1	RATACAND	RATACAND 16 mg	1/2 al mattino
6 1	LOPRESOR	LOPRESOR	1/2 al mattino e alla sera
7 1	LIMPIDEX	LIMPIDEX 15 mg	ore 07.00 ore 22.00
8 1	MAALOX PLUS	MAALOX PLUS	ore 10.00 e ore 17.00
9 7	LOPRESOR	lopresor	un/una al mattino
10 8	LOPRESOR	LOPRESOR	null

Figura 2.5: DataFrame df_farmaci senza correzioni

FARMACO_FINELE	ID_VISITA	CONFEZIONE	DOSE
1 RATACAND	0	RATACAND 16 mg	1/2 al mattino
2 LOPRESOR	0	LOPRESOR	1/2 al mattino e alla sera
3 MAALOX	0	MAALOX 400mg	ogni 8 ore
4 ARMOLIPID	0	ARMOLIPID	un/una alla sera
5 LOPRESOR	1	LOPRESOR	1/2 al mattino e alla sera
6 RATACAND	1	RATACAND 16 mg	1/2 al mattino
7 LIMPIDEX	1	LIMPIDEX 15 mg	ore 07.00 ore 22.00
8 MAALOX PLUS	1	MAALOX PLUS	ore 10.00 e ore 17.00
9 LOPRESOR	7	lopresor	un/una al mattino
10 SIVASTIN	8	SIVASTIN	null

Figura 2.6: DataFrame df_farmaci con correzioni

Capitolo 3

Named Entity Recognition (NER)

In questo capitolo si riportano addestramento ed inferenza delle pipeline di NLP implementate tramite **John Snow Labs** al fine di estrarre informazioni relative a sintomi e malattie dei pazienti.

3.1 Introduzione al problema

Named Entity Recognition (NER) è un task di **Information Extraction (IE)**, il cui scopo è quello di identificare entità di interesse, dette *named entity*, all'interno di documenti testuali non strutturati. Le entità di interesse sono definite a priori e dipendono fortemente dal dominio del problema. Nel problema trattato, le entità considerate sono sintomi e malattie dei pazienti, da ricercare all'interno delle anamnesi e delle diagnosi scritte dai medici durante ogni visita.

Come riportato in [3], è possibile distinguere le tecniche di NER in quattro categorie:

- **Rule-based**: approcci tradizionali basati su regole sintattiche e semantiche definite a priori. Se da un lato non sono richiesti dati etichettati, dall'altro è necessario che un esperto di dominio definisca le regole ad-hoc.
- **Unsupervised learning**: approcci basati su algoritmi di *clustering*. L'idea di base è quella di estrarre le entità dai cluster determinati, sulla base di informazioni relative al loro contesto.
- **Feature-based supervised learning**: approcci basati su algoritmi di *Machine Learning (ML)* tradizionali, applicati a valle di una critica fase di *feature engineering*, il cui obiettivo è quello di determinare una rappresentazione vettoriale delle parole.
- **Deep Learning (DL)**: approcci basati su reti neurali, in grado di estrarre automaticamente le feature necessarie per identificare le entità di interesse.

3.1.1 Modello utilizzato

Per risolvere il problema di NER si è scelto di utilizzare uno degli approcci basati su DL, in quanto rappresentano l'attuale stato dell'arte. Nello specifico, è stata utilizza-

ta l'architettura tipo **Char CNN - Bi-LSTM - CRF** [4], implementata dalla classe `NerDLApproach` della libreria *John Snow Labs*. Come riportato in Figura 3.1, l'architettura è composta da 3 stadi principali:

- **Char CNN:** *Convolutional Neural Network (CNN)* utilizzata per estrarre una rappresentazione a livello di carattere di ogni parola. La rete convoluzionale riceve in ingresso i *char embeddings*, ovvero dei vettori associati ai caratteri della parola, e, dopo alcune convoluzioni monodimensionali effettuate con diversi filtri e dopo un'operazione di max-pooling, restituisce in uscita la rappresentazione a livello di carattere della parola. I char embeddings sono appresi durante il processo di addestramento della rete.
- **Bi-directional Long Short-Term Memory (Bi-LSTM):** particolare *Recurrent Neural Network (RNN)* in grado di catturare le dinamiche all'interno di sequenze di dati grazie alla presenza di cicli nell'architettura. A differenza delle RNNs tradizionali, le LSTMs non soffrono dei problemi di *gradient vanishing/exploding*. Difatti, una LSTM è composta da un'insieme di unità elementari dotate di porte logiche che controllano la porzione di informazione da dimenticare e da preservare per il prossimo step temporale. Mentre nelle LSTMs tradizionali le informazioni provengono solo dai campioni passati, nelle Bi-LSTM si hanno due percorsi distinti, uno di forward e uno di backward, che ricevono rispettivamente campioni passati e futuri. Nel contesto dell'architettura del modello di NER, lo stadio Bi-LSTM riceve in ingresso dei vettori ottenuti concatenando la rappresentazione a livello di carattere di una parola con il relativo *word embeddings*. A differenza dei char embeddings, i word embedding sono forniti in ingresso alla rete, per cui devono essere determinati in una fase preliminare.
- **Conditional Random Field (CRF):** modello probabilistico usato per assegnare una label ad ogni parola di una frase. Difatti, CRF può essere visto come una generalizzazione di un *Hidden Markov Model (HMM)* in cui le probabilità di transizione non sono costanti ma arbitrarie. Ricevuti in ingresso le uscite della Bi-LSTM, CRF determina in maniera congiunta la catena di label più probabili per una determinata frase, sfruttando la correlazione esistente tra parole vicine.

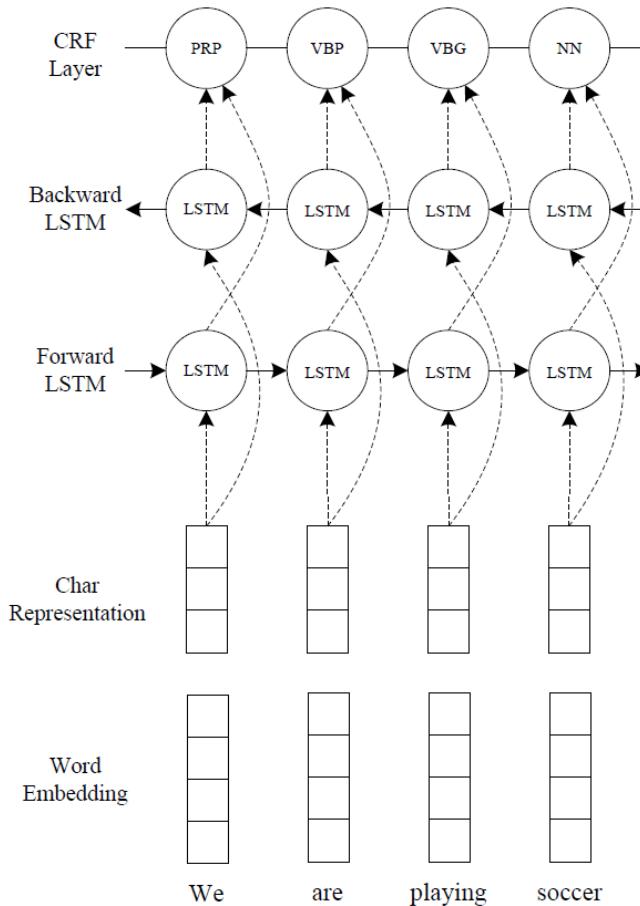


Figura 3.1: Modello Char CNN - Bi-LSTM - CRF

3.2 Training

In questa sezione sono riportate le operazioni effettuate per addestrare il modello appena presentato, utilizzando i dati contenuti nel file `symptoms_diseases.txt`. Tutte le operazioni relative all'addestramento sono contenute nel notebook `NER_training_colab.ipynb`.

3.2.1 Preprocessing

Per poter addestrare il modello è necessario convertire il dataset di training in un formato compatibile con l'API offerta da spark-nlp.

Conversione del dataset originale

```
df_sym = df_sym.withColumn("value", regexp_replace("value", '\t', ' '))
columns = ['value']
vals = [['DOCSTART-X-X-O'], ['']]
intestazione = spark.createDataFrame(vals, columns)
df_sym = intestazione.union(df_sym)
```

A questo punto il dataset ottenuto a valle della conversione, `training_set.txt`, può essere letto utilizzando la classe `CoNLL` di `spark-nlp`. Tale operazione restituisce un DataFrame costituito dalle seguenti colonne:

- *text*: contiene il testo di un'istanza di training (e.g. una singola anamnesi o diagnosi).
- *document*: contiene la lista dei documenti presenti nelle istanze di training; in questo contesto ogni istanza è vista come un singolo documento.
- *sentence*: contiene la lista delle frasi presenti in ogni documento.
- *token*: contiene la lista dei token presenti in ogni frase.
- *pos*: contiene la lista di label associate ai diversi token, ognuna delle quali indica il ruolo grammaticale del relativo token; tale label fa riferimento ad un problema noto come **Part Of Speech (POS) Tagging**.
- *label*: contiene la lista di label associate ai diversi token, ognuna delle quali indica l'entità del relativo token, con riferimento al problema di NER.

È ben osservare che nel dataset iniziale non si hanno a disposizione informazioni relative a *sentence* e *pos*, di conseguenza tali colonne durante la lettura non vengono popolate correttamente. Per estrarre le informazioni corrette a partire dal testo si utilizza una pipeline composta da 3 stadi:

- `SentenceDetector`: classe in grado di rilevare le frasi all'interno di un documento, applicando tecniche di NLP tradizionali.
- `PerceptronModel`: modello di ML tradizionale, detto *Multi-Layer Perceptron (MLP)*, preaddestrato su un dataset in lingua italiana, in grado di assegnare ad ogni token il relativo POS tag.
- `NerConverter`: classe utilizzata per convertire le entità estratte dal NER in un formato più user-friendly. Ad esempio, le parole "Ex" e "fumatore", a cui sono associate rispettivamente le label *B-Symptom* ed *I-Symptom*, vengono accorpate in un unico *chunk* "Ex fumatore" a cui viene associata la label *Symptom*.

Pipeline di preprocessing

```

sentence = SentenceDetector() \
.setInputCols(["document"]) \
.setOutputCol("sentence")

pos = PerceptronModel.pretrained("pos_ud_isdt", "it") \
.setInputCols(["document", "token"]) \
.setOutputCol("pos")

converter = NerConverter() \
.setInputCols(["document", "token", "label"]) \
.setOutputCol("chunk")

preproc_pipeline = Pipeline(
    stages = [
        sentence,
        pos,
        converter
    ]
)

```

```
])
df = preproc_pipeline.fit(df).transform(df)
```

Quindi, la fase di preprocessing si conclude eliminando le entry duplicate dal DataFrame, in modo da ottenere un numero di entry distinte pari a 5410.

3.2.2 Data Analysis

A questo punto è stata effettuata un'analisi esplorativa dei dati a disposizione. Tra le altre cose, sono state calcolate il numero di entità distinte e non, distinguendo anche tra malattie e sintomi come riportato in Figura 3.2.

Entità nei dati di training

```
entities = df.select("chunk.result", "chunk.metadata") \
    .withColumn("tmp", arrays_zip("result", "metadata")) \
    .withColumn("tmp", explode("tmp")) \
    .select("tmp.result", "tmp.metadata.entity") \
    .withColumnRenamed("result", "chunk")

print("Numero di coppie (chunk, entity):", entities.count())
print("Numero di coppie (chunk, entity) distinte:", entities.distinct().count())

display(entities.groupBy("entity").count())
display(entities.distinct().groupBy("entity").count())
```

	entity	count
1	Disease	6167
2	Symptom	4942

(a) Numero di entità non distinte

	entity	count
1	Disease	1736
2	Symptom	1788

(b) Numero di entità distinte

Figura 3.2: Conteggio entità

3.2.3 Data Preparation

Prima di procedere con l'addestramento del modello, si è scelto di dividere i dati a disposizione in due insiemi disgiunti:

- *training set*: composto dal 90% del dataset originale, viene utilizzato per addestrare il modello.
- *test set*: composto dal restante 10% del dataset originale, viene utilizzato per valutare le prestazioni del modello. Questo permette di avere una stima realistica delle performance, in quanto il modello viene testato su istanze che non ha mai processato in fase di training.

In particolare, le istanze di test sono ottenute mediante un campionamento casuale senza ripetizione, mentre quelle di training attraverso un'operazione di *anti join*.

Divisione in training e test set

```
temp = df.withColumn("id", monotonically_increasing_id())
df_test = temp.sample(False, 0.1, seed=0)
df_train = temp.join(df_test, on="id", how="left_anti")
```

Infine, si verifica che i due dataset ottenuti siano bilanciati in termini di entità (distinte e non), come riportato in Figura 3.3 e Figura 3.4.

Verifica bilanciamento dei dataset

```
train_entities = df_train.select("chunk.result", "chunk.metadata") \
    .withColumn("tmp", arrays_zip("result", "metadata")) \
    .withColumn("tmp", explode("tmp")) \
    .select("tmp.result", "tmp.metadata.entity") \
    .withColumnRenamed("result", "chunk")

display(train_entities.groupBy("entity").count())
display(train_entities.distinct().groupBy("entity").count())

test_entities = df_test.select("chunk.result", "chunk.metadata") \
    .withColumn("tmp", arrays_zip("result", "metadata")) \
    .withColumn("tmp", explode("tmp")) \
    .select("tmp.result", "tmp.metadata.entity") \
    .withColumnRenamed("result", "chunk")

display(test_entities.groupBy("entity").count())
display(test_entities.distinct().groupBy("entity").count())
```

entity	count
1 Disease	5638
2 Symptom	4474

(a) Numero di entità non distinte

entity	count
1 Disease	1618
2 Symptom	1650

(b) Numero di entità distinte

Figura 3.3: Conteggio entità training set

entity	count
1 Disease	529
2 Symptom	468

(a) Numero di entità non distinte

entity	count
1 Disease	315
2 Symptom	320

(b) Numero di entità distinte

Figura 3.4: Conteggio entità test set

3.2.4 Training

Come illustrato in Figura 3.1, il modello NER utilizzato richiede in ingresso gli embeddings delle parole che compongono una frase. Un **word embedding** non è altro che una rappresentazione alternativa di una parola, un vettore di numeri reali che codifica il

significato della parola stessa, in modo che parole semanticamente simili siano più vicine nello spazio vettoriale. Per ottenere gli embeddings si è scelto di utilizzare la classe `BertEmbeddings` di `spark-nlp`, la quale fornisce un modello **Bidirectional Encoder Representations from Transformers (BERT)** preaddestrato su un dataset in lingua italiana. Ricevuti in ingresso una sentence e i relativi token, BERT restituisce gli embeddings delle parole.

Per addestrare il modello NER è necessario impostare alcuni iperparametri, tra cui *learning rate*, *batch size* e *max epochs*. Ai diversi iperparametri sono assegnati i valori di default, ad eccezione del numero massimo di epoche che è stato impostato a 10 per motivi di costo computazionale. Non è stato possibile addestrare il modello per un numero maggiore di epoche a causa delle risorse computazionali limitate.

Pipeline di training

```
bert = BertEmbeddings.pretrained("bert_base_italian_uncased", lang="it") \
.setInputCols("sentence", "token") \
.setOutputCol("bert") \
.setCaseSensitive(False)

DL_ner = NerDLApproach() \
.setInputCols(["sentence", "token", "bert"]) \
.setLabelColumn("label") \
.setOutputCol("ner") \
.setMaxEpochs(10) \
.setLr(1e-3) \
.setPo(0.005) \
.setBatchSize(8) \
.setRandomSeed(0) \
.setVerbose(2) \
.setEnabledOutputLogs(True)

DL_pipeline = Pipeline(
    stages = [
        bert,
        DL_ner
    ]
)
```

Definita la pipeline di training concatenando BERT e NER è possibile procedere con l'addestramento. Terminato l'addestramento, il quale ha impiegato più di 2 ore, si salvano i pesi del NER, corrispondente al secondo stadio della pipeline, in modo da poterlo utilizzare in futuro.

Fit della pipeline e salvataggio del modello

```
DL_model = DL_pipeline.fit(df_train)

model_path = '/content/drive/MyDrive/final-project-BDABI/models/NER_DL_' + \
    datetime.datetime.today().strftime("%y-%m-%d-%H-%M")
DL_model.stages[1].write().save(model_path)
```

Infine, in Figura 3.5 si riporta l'andamento della funzione di loss calcolata al variare delle epoche. I valori di loss vengono scritti automaticamente in un file di log, opportunamente abilitato.

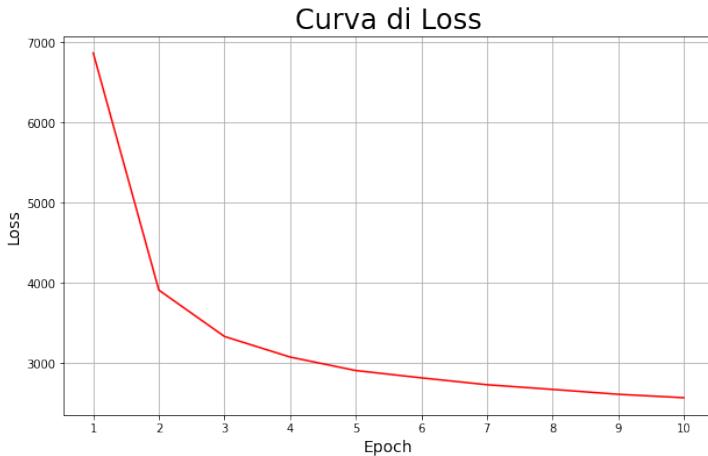


Figura 3.5: Curva di Loss

3.2.5 Evaluation

Una volta completato l'addestramento è possibile caricare il modello appena salvato e valutarne le performance sui dati di test. È bene osservare che anche in fase di test è necessario estrarre gli embeddings con BERT, prima di porre i dati in ingresso al NER.

Caricamento del modello e inferenza sui dati di test

```
DL_ner_loaded = NerDLModel.load(model_path) \
    .setInputCols(["sentence", "token", "bert"]) \
    .setOutputCol("ner")

DL_preds = bert.transform(df_test)
DL_preds = DL_ner_loaded.transform(DL_preds)
```

A questo punto, dopo alcune elaborazioni, è possibile costruire un DataFrame in cui sono riportati i token, le label effettive a loro associate e le label predette dal modello.

Elaborazione dei risultati

```
preds = DL_preds.select(col("token.result").alias("token"), \
    col("label.result").alias("label"), \
    col("ner.result").alias("ner")) \
    .withColumn("tmp", arrays_zip("token", "label", "ner")) \
    .withColumn("tmp", explode("tmp")) \
    .select("tmp.token", "tmp.label", "tmp.ner")
```

A partire dal DataFrame ottenuto, dopo averlo convertito in un DataFrame della libreria *pandas*, si possono calcolare le performance del modello utilizzando la funzione `classification_report` della libreria *scikit-learn*. In particolare, la funzione valuta le prestazioni calcolando, tra le altre cose, le seguenti metriche, ognuna delle quali viene calcolata per ogni label:

- **precision**: rapporto tra il numero di istanze classificate correttamente appartenenti alla classe C e il numero di istanze a cui viene assegnata l'etichetta C.

$$prec = \frac{TP}{TP + FP}$$

- **recall:** rapporto tra il numero di istanze classificate correttamente appartenenti alla classe C e il numero di istanze totali appartenenti alla classe C.

$$rec = \frac{TP}{TP + FN}$$

- **F1:** due volte il prodotto di precision e recall diviso la somma di precision e recall.

$$F1 = 2 \cdot \frac{prec \cdot rec}{prec + rec}$$

Metriche di performance

```
preds_df = preds.toPandas()
report = classification_report(preds_df["label"], preds_df["ner"])
```

In Tabella 3.1 e 3.2 si riportano le performance del modello addestrato per 5 e 10 epoche. Evidentemente si hanno prestazioni migliori nella rilevazione di malattie rispetto a quella di sintomi.

	Precision	Recall	F1-score
<i>B-Disease</i>	0.74	0.82	0.77
<i>B-Symptom</i>	0.68	0.46	0.55
<i>I-Disease</i>	0.66	0.77	0.71
<i>I-Symptom</i>	0.58	0.41	0.48
<i>O</i>	0.94	0.96	0.95

Tabella 3.1: Performance NER 5 epoche

	Precision	Recall	F1-score
<i>B-Disease</i>	0.70	0.80	0.75
<i>B-Symptom</i>	0.59	0.55	0.57
<i>I-Disease</i>	0.66	0.62	0.64
<i>I-Symptom</i>	0.60	0.56	0.58
<i>O</i>	0.94	0.94	0.94

Tabella 3.2: Performance NER 10 epoche

3.3 Inferenza

Dopo aver addestrato un modello di NER in grado di raggiungere delle buone performance, si è passati alla fase di inferenza. In questa fase vengono estratte le entità di interesse, ovvero sintomi e malattie dei pazienti, a partire dalle anamnesi e diagnosi contenute nel file DF_ANAM_DIA_COMPLETO.csv. Tutte le operazioni descritte nella sezione corrente sono contenute nel notebook NER_inference_colab.ipynb.

Dopo aver letto il file proveniente dalla fase di preprocessing, si definisce la pipeline di inferenza, la cui architettura è riportata in Figura 3.6.

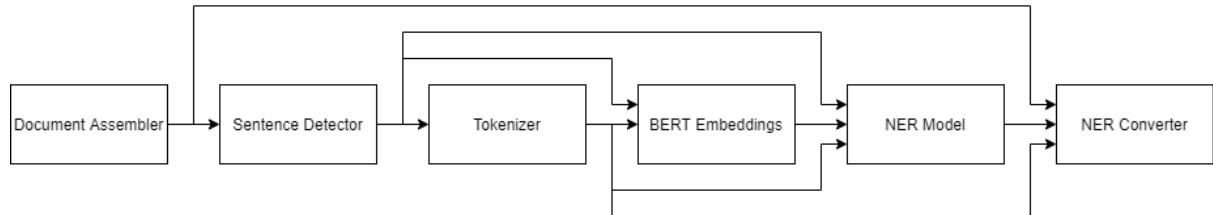


Figura 3.6: Architettura pipeline di inferenza

Pipeline di inferenza

```

document = DocumentAssembler() \
    .setInputCol("text") \
    .setOutputCol("document")

sentence = SentenceDetector() \
    .setInputCols(["document"]) \
    .setOutputCol("sentence")

tokenizer = Tokenizer() \
    .setInputCols(["sentence"]) \
    .setOutputCol("token") \
    .setSplitChars(["."])

bert = BertEmbeddings.pretrained("bert_base_italian_uncased", lang="it") \
    .setInputCols("sentence", "token") \
    .setOutputCol("bert") \
    .setCaseSensitive(False)

model = "NER_DL_21-07-02-13-02"    # 10 epochs

ner = NerDLModel.load("/content/drive/MyDrive/final-project-BDABI/models/" + model) \
    .setInputCols(["sentence", "token", "bert"]) \
    .setOutputCol("ner")

converter = NerConverter() \
    .setInputCols(["document", "token", "ner"]) \
    .setOutputCol("chunk")

pipeline = Pipeline(
    stages = [
        document,
        sentence,
        tokenizer,
        bert,
        ner,
        converter
    ]
)
  
```

Tutti gli stadi della pipe sono stati descritti nelle sezioni precedenti, ad eccezione del Tokenizer, classe in grado di estrarre i token all'interno di una frase, applicando tecniche di NLP tradizionali.

3.3.1 Anamnesi

Dopo aver selezionato le visite in cui è presente effettivamente l'anamnesi, 2012 su un totale di 147599, vengono forniti i dati in ingresso alla pipeline.

Inferenza sulle anamnesi

```
anamnesi = df.dropna(subset=[ "ANAMNESI" ]) \
    .select("ID_VISITA", "ANAMNESI") \
    .withColumnRenamed("ANAMNESI", "text")

res_anamnesi = pipeline.fit(anamnesi).transform(anamnesi)
```

Terminata l'inferenza, operazione che ha richiesto circa 5 minuti, viene salvato il DataFrame risultante, contenente il testo di ogni anamnesi e le entità in esso identificate.

In Figura 3.7 è riportato il risultato della fase di inferenza.

ID_VISITA	chunk	text	
1	145553	^ [{"annotatorType": "chunk", "begin": 13, "embeddings": [], "end": 20, "metadata": {"chunk": "0", "entity": "Disease", "sentence": "0"}, "result": "iperteso"}, {"annotatorType": "chunk", "begin": 23, "embeddings": [], "end": 35, "metadata": {"chunk": "1", "entity": "Disease", "sentence": "0"}, "result": "dislipidemico"}, {"annotatorType": "chunk", "begin": 57, "embeddings": [], "end": 59, "metadata": {"chunk": "2", "entity": "Disease", "sentence": "0"}, "result": "IMA"}, {"annotatorType": "chunk", "begin": 137, "embeddings": [], "end": 166, "metadata": {"chunk": "3", "entity": "Symptom", "sentence": "0"}, "result": "dyspnea da sforzi non abituali"}]	Ex fumatore, iperteso, dislipidemico, nega familiari. IMA nel 2006. PCI primaria su IVA e successiva PCI su Cx. Nega angor, riferisce dispnea da sforzi non abituali.
2	145554	^ [{"annotatorType": "chunk", "begin": 0, "embeddings": [], "end": 7, "metadata": {"chunk": "0", "entity": "Disease", "sentence": "0"}, "result": "iperteso"}, {"annotatorType": "chunk", "begin": 104, "embeddings": [], "end": 115, "metadata": {"chunk": "1", "entity": "Symptom", "sentence": "0"}, "result": "palpitazioni"}, {"annotatorType": "chunk", "begin": 221, "embeddings": [], "end": 249, "metadata": {"chunk": "2", "entity": "Disease", "sentence": "0"}, "result": "tachicardia da rientro nodale"}, {"annotatorType": "chunk", "begin": 441, "embeddings": [], "end": 443, "metadata": {"chunk": "3", "entity": "Disease", "sentence": "0"}, "result": "BBS"}]	Iperteso in terapia con vasoretti. Ex fumatore, nega altri fattori di rischio cardiovascolare. Riferisce palpitazioni da circa 5 anni. Ha eseguito nel maggio 2015 studio elettrofisiologico transesofageo che ha evidenziato tachicardia da rientro nodale. Ha praticato terapia con flecainide e verapamil attualmente assume cronicamente e che non ha ridotto la frequenza degli episodi che rimangono all'incirca bimestrali. Ha sviluppato per&BBS. Ultimamente ha sospeso terapia con vasoretti autonomam...
3	145555	^ [{"annotatorType": "chunk", "begin": 78, "embeddings": [], "end": 89, "metadata": {"chunk": "0", "entity": "Symptom", "sentence": "0"}, "result": "palpitazioni"}, {"annotatorType": "chunk", "begin": 159, "embeddings": [], "end": 187, "metadata": {"chunk": "1", "entity": "Disease", "sentence": "0"}, "result": "tachicardia da rientro nodale"}, {"annotatorType": "chunk", "begin": 321, "embeddings": [], "end": 323, "metadata": {"chunk": "2", "entity": "Disease", "sentence": "0"}, "result": "BBS"}]	Iperteso. Ex fumatore, nega altri fattori di rischio cardiovascolare. Riferisce palpitazioni da circa 5 anni. Ha eseguito nel maggio 2015 SEE che ha evidenziato tachicardia da rientro nodale. Ha praticato terapia con flecainide e verapamil cronicamente che non hanno ridotto la frequenza degli episodi. Ha sviluppato per&BBS. A Giugno un ablazione con RF di AVNRT con approccio alla via lenta. Non ricevile documentabili nA sintomi di rilievo.
4	145556	^ [{"annotatorType": "chunk", "begin": 78, "embeddings": [], "end": 89, "metadata": {"chunk": "0", "entity": "Symptom", "sentence": "0"}, "result": "palpitazioni"}, {"annotatorType": "chunk", "begin": 159, "embeddings": [], "end": 187, "metadata": {"chunk": "1", "entity": "Disease", "sentence": "0"}, "result": "tachicardia da rientro nodale"}, {"annotatorType": "chunk", "begin": 321, "embeddings": [], "end": 323, "metadata": {"chunk": "2", "entity": "Disease", "sentence": "0"}, "result": "BBS"}]	Iperteso. Ex fumatore, nega altri fattori di rischio cardiovascolare. Riferisce palpitazioni da circa 5 anni. Ha eseguito nel maggio 2015 SEE che ha evidenziato tachicardia da rientro nodale. Ha praticato terapia con flecainide e verapamil cronicamente che non hanno ridotto la frequenza degli episodi. Ha sviluppato per&BBS. A Giugno un ablazione con RF di AVNRT con approccio alla via lenta. Non ricevile documentabili nA sintomi di rilievo.

Figura 3.7: Risultato dell'inferenza

3.3.2 Diagnosi

Analogamente, dopo aver selezionato le visite in cui è presente effettivamente la diagnosi, 134094 su un totale di 147599, vengono forniti i dati in ingresso alla pipeline.

Inferenza sulle diagnosi

```
diagnosi = df.dropna(subset=[ "DIAGNOSI" ]) \
    .select("ID_VISITA", "DIAGNOSI") \
    .withColumnRenamed("DIAGNOSI", "text")

res_diagnosi = pipeline.fit(diagnosi).transform(diagnosi)
```

Terminata l'inferenza, operazione che ha richiesto circa 2 ore, viene salvato il DataFrame risultante, contenente il testo di ogni diagnosi e le entità in esso identificate.

Capitolo 4

Graph-based Database

Nel capitolo corrente viene descritta la modellazione dei dati e la successiva implementazione del database a grafo in **Neo4j**.

4.1 Modello dei dati

Nel modello dei dati sono incluse tutte le informazioni ottenute a valle delle fasi di preprocessing e NER, tra cui quelle relative a:

- *visite*: dati contenuti nel file `DF_VISITA_COMPLETO.csv`; includono la data della visita, informazioni sul paziente e il medico coinvolti.
- *farmaci*: dati contenuti nel file `DF_FARMACI_COMPLETO.csv`; includono nome, confezione e dosaggio di ogni farmaco prescritto ad ogni visita.
- *malattie e sintomi*: dati contenuti nei file memorizzati dopo aver applicato la pipeline di inferenza; includono tutte le malattie e i sintomi rilevati nelle anamnesi e nelle diagnosi scritte ad ogni visita.

Si è scelto quindi di modellare le informazioni appena descritte in un grafo coerente, la cui struttura è riportata in Figura 4.1.

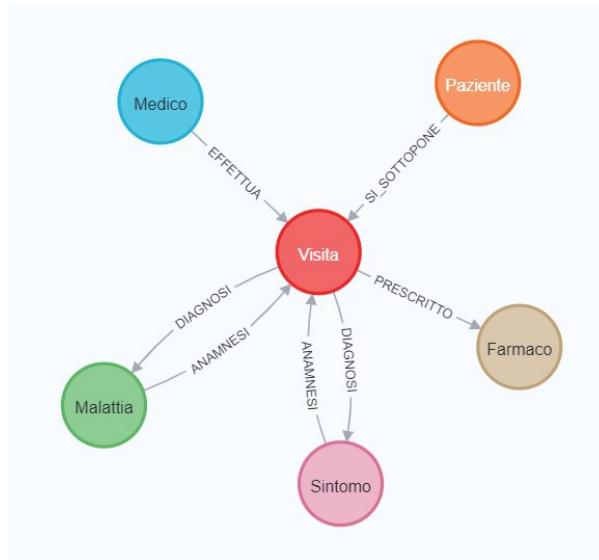


Figura 4.1: Schema del database

Lo schema del database prevede un nodo *Visita*, opportunamente collegato al *Paziente* e al *Medico* esaminatore. Contestualmente, tale nodo è stato collegato ai vari *Farmaci* prescritti dal medico, nonché ai vari *Sintomi* e *Malattie* estratti dalle informazioni non strutturate di anamnesi e diagnosi. Si noti inoltre come, per distinguere le entità rilevate rispettivamente in *Anamnesi* e *Diagnosi*, siano state definite due relazioni diverse che legano sintomi e malattie alla visita.

4.2 Implementazione

In questa sezione si riportano tutte le operazioni effettuate per memorizzare i dati elaborati all'interno di Neo4j.

4.2.1 Postprocessing delle entità

Dopo aver letto i file provenienti dalle fasi precedenti è necessario convertire i dati relativi alle entità (malattie e sintomi) in un formato più adatto alla memorizzazione sul database.

Postprocessing entity anamnesi

```

entities_anamnesi = res_anamnesi.select("ID_VISITA", "chunk.result", "chunk.metadata") \
    .withColumn("tmp", arrays_zip("result", "metadata")) \
    .withColumn("tmp", explode("tmp")) \
    .select("ID_VISITA", "tmp.result", "tmp.metadata.entity") \
    .withColumnRenamed("result", "chunk") \
    .withColumn("chunk", upper(col("chunk")))
entities_anamnesi = entities_anamnesi.drop_duplicates()
  
```

In Figura 4.2 è riportato il risultato del postprocessing.

	ID_VISITA	chunk	entity
1	145553	IPERTESO	Disease
2	145553	DISLIPIDEMICO	Disease
3	145553	IMA	Disease
4	145553	DISPNEA DA SFORZI NON ABITUALI	Symptom
5	145554	IPERTESO	Disease
6	145554	PALPITAZIONI	Symptom
7	145554	TACHICARDIA DA RIENTRO NODALE	Disease
8	145554	BBS	Disease
9	145555	PALPITAZIONI	Symptom
10	145555	TACHICARDIA DA RIENTRO NODALE	Disease

Figura 4.2: Risultato del postprocessing

Postprocessing entity diagnosis

```
entities_diagnosi = res_diagnosi.select("ID_VISITA", "chunk.result", "chunk.metadata") \
    .withColumn("tmp", arrays_zip("result", "metadata")) \
    .withColumn("tmp", explode("tmp")) \
    .select("ID_VISITA", "tmp.result", "tmp.metadata.entity") \
    .withColumnRenamed("result", "chunk") \
    .withColumn("chunk", upper(col("chunk")))

entities_diagnosi = entities_diagnosi.drop_duplicates()
```

4.2.2 Nodi

A questo punto è possibile procedere con la memorizzazione dei nodi del grafo.

4.2.2.1 Pazienti

Si caricano nel database i nodi *Paziente*, identificati dal campo *codice*.

Caricamento pazienti

```
patients = exams.select("CODPAZ") \
    .distinct() \
    .withColumnRenamed("CODPAZ", "codice") \
    .orderBy("codice")

patients.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Paziente") \
    .option("node.keys", "codice") \
    .mode("Overwrite") \
    .save()
```

4.2.2.2 Medici

Si caricano nel database i nodi *Medico*, caratterizzati dai campi *codice* e *nome*.

Caricamento medici

```
doctors = exams.select("COD_MEDICO_FIRMANTE", "ESAMINATORE") \
    .distinct() \
    .withColumnRenamed("COD_MEDICO_FIRMANTE", "codice") \
    .withColumnRenamed("ESAMINATORE", "nome") \
    .orderBy("codice") \
    .dropna(subset=["codice"])

doctors.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Medico") \
    .option("node.keys", "codice") \
    .option("node.properties", "nome") \
    .mode("Overwrite") \
    .save()
```

4.2.2.3 Visite

Si caricano nel database i nodi *Visita*, caratterizzati dai campi *id* e *data*.

Caricamento visite

```
exams_ = exams.select("ID_VISITA", "DATA_EVENTO") \
    .distinct() \
    .withColumnRenamed("ID_VISITA", "id") \
    .withColumnRenamed("DATA_EVENTO", "data") \
    .orderBy("id")

exams_.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Visita") \
    .option("node.keys", "id, data") \
    .mode("Overwrite") \
    .save()
```

4.2.2.4 Malattie

Si caricano nel database i nodi *Malattia*, identificati dal campo *nome*.

Caricamento malattie

```
disease_anamnesi = entities_anamnesi.filter("entity == 'Disease'") \
    .select("chunk") \
    .withColumnRenamed("chunk", "nome") \
    .distinct()

disease_diagnosi = entities_diagnosi.filter("entity == 'Disease'") \
    .select("chunk") \
    .withColumnRenamed("chunk", "nome") \
    .distinct()

diseases = disease_anamnesi.union(disease_diagnosi).distinct()

diseases.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Malattia") \
    .option("node.keys", "nome") \
    .mode("Overwrite") \
    .save()
```

4.2.2.5 Sintomi

Si caricano nel database i nodi *Sintomo*, identificati dal campo *nome*.

Caricamento sintomi

```
symptom_anamnesi = entities_anamnesi.filter("entity == 'Symptom'") \
    .select("chunk") \
    .withColumnRenamed("chunk", "nome") \
    .distinct()

symptom_diagnosi = entities_diagnosi.filter("entity == 'Symptom'") \
    .select("chunk") \
    .withColumnRenamed("chunk", "nome") \
    .distinct()

symptoms = symptom_anamnesi.union(symptom_diagnosi).distinct()

symptoms.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Sintomo") \
    .option("node.keys", "nome") \
    .mode("Overwrite") \
    .save()
```

4.2.2.6 Farmaci

Si caricano nel database i nodi *Farmaco*, identificati dal campo *nome*.

Caricamento farmaci

```
drugs_ = drugs.select("FARMACO") \
    .distinct() \
    .withColumnRenamed("FARMACO", "nome") \
    .orderBy("nome")

drugs_.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("labels", ":Farmaco") \
    .option("node.keys", "nome") \
    .mode("Overwrite") \
    .save()
```

4.2.3 Relazioni

Per concludere, si definiscono le relazioni trai nodi, seguendo lo schema descritto in Figura 4.1.

4.2.3.1 Paziente - Visita

Si definisce la relazione tra i nodi *Paziente* e *Visita*, a cui è associata l'etichetta *SI_SOTTOPONE*.

È bene osservare che le informazioni relative a sesso, età, peso e altezza dei pazienti sono state associate alla relazione che lega il paziente alla visita, in quanto, in linea di principio, tali informazioni possono cambiare nel tempo.

Definizione relazione tra pazienti e visite

```
patient_exam = exams.select("ID_VISITA", "CODPAZ") \
    .distinct() \
    .withColumnRenamed("ID_VISITA", "id") \
    .withColumnRenamed("CODPAZ", "codice") \
    .orderBy("id")

patient_exam.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("relationship", "SI_SOTOPONE") \
    .option("relationship.save.strategy", "keys") \
    .option("relationship.source.labels", ":Paziente") \
    .option("relationship.source.save.mode", "overwrite") \
    .option("relationship.source.node.keys", "codice") \
    .option("relationship.target.labels", ":Visita") \
    .option("relationship.target.node.keys", "id") \
    .option("relationship.target.save.mode", "overwrite") \
    .option("relationship.properties", \
        "SESSO:sesso, ETA:eta, PESO:peso, ALTEZZA:altezza") \
    .mode("Overwrite") \
    .save()
```

4.2.3.2 Medico - Visita

Si definisce la relazione tra i nodi *Medico* e *Visita*, a cui è associata l'etichetta *EFFETTUUA*.

Definizione relazione tra medici e visite

```
doctor_exam = exams.select("ID_VISITA", "COD_MEDICO_FIRMANTE") \
    .distinct() \
    .withColumnRenamed("ID_VISITA", "id") \
    .withColumnRenamed("COD_MEDICO_FIRMANTE", "codice") \
    .orderBy("id") \
    .dropna()

doctor_exam.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("relationship", "EFFETTUUA") \
    .option("relationship.save.strategy", "keys") \
    .option("relationship.source.labels", ":Medico") \
    .option("relationship.source.save.mode", "overwrite") \
    .option("relationship.source.node.keys", "codice") \
    .option("relationship.target.labels", ":Visita") \
    .option("relationship.target.node.keys", "id") \
    .option("relationship.target.save.mode", "overwrite") \
    .mode("Overwrite") \
    .save()
```

4.2.3.3 Malattia - Visita

Nel definire la relazione tra i nodi *Malattia* e *Visita* è necessario distinguere le malattie che fanno riferimento ad anamnesi da quelle che fanno riferimento alla diagnosi

Si definisce la relazione *ANAMNESI*, orientata dalla malattia alla visita.

Definizione relazione tra malattie di anamnesi e visite

```
disease_anamnesi = entities_anamnesi.filter("entity == 'Disease'") \
    .select("ID_VISITA", "chunk") \
    .withColumnRenamed("chunk", "nome") \
    .withColumnRenamed("ID_VISITA", "id")
```

```
disease_anamnesi.write.format("org.neo4j.spark.DataSource")\
.option("url", url) \
.option("authentication.type", "basic") \
.option("authentication.basic.username", username) \
.option("authentication.basic.password", password) \
.option("relationship", "ANAMNESI") \
.option("relationship.save.strategy", "keys") \
.option("relationship.source.labels", ":Malattia") \
.option("relationship.source.save.mode", "overwrite") \
.option("relationship.source.node.keys", "nome") \
.option("relationship.target.labels", ":Visita") \
.option("relationship.target.node.keys", "id") \
.option("relationship.target.save.mode", "overwrite") \
.mode("Overwrite") \
.save()
```

Si definisce la relazione *DIAGNOSSI*, orientata dalla visita alla malattia.

Definizione relazione tra malattie di diagnosi e visite

```
disease_diagnosi = entities_diagnosi.filter("entity == 'Disease'") \
.select("ID_VISITA", "chunk") \
.withColumnRenamed("chunk", "nome") \
.withColumnRenamed("ID_VISITA", "id")

disease_diagnosi.write.format("org.neo4j.spark.DataSource")\
.option("url", url) \
.option("authentication.type", "basic") \
.option("authentication.basic.username", username) \
.option("authentication.basic.password", password) \
.option("relationship", "DIAGNOSSI") \
.option("relationship.save.strategy", "keys") \
.option("relationship.source.labels", ":Visita") \
.option("relationship.source.save.mode", "overwrite") \
.option("relationship.source.node.keys", "id") \
.option("relationship.target.labels", ":Malattia") \
.option("relationship.target.node.keys", "nome") \
.option("relationship.target.save.mode", "overwrite") \
.mode("Overwrite") \
.save()
```

4.2.3.4 Sintomo - Visita

Anche nel caso della relazione tra *Sintomo* e *Visita* è necessario distinguere i sintomi che fanno riferimento ad anamnesi da quelle che fanno riferimento alla diagnosi.

Si definisce la relazione *ANAMNESI*, orientata dal sintomo alla visita.

Definizione relazione tra sintomi di anamnesi e visite

```
symptom_anamnesi = entities_anamnesi.filter("entity == 'Symptom'") \
.select("ID_VISITA", "chunk") \
.withColumnRenamed("chunk", "nome") \
.withColumnRenamed("ID_VISITA", "id")

symptom_anamnesi.write.format("org.neo4j.spark.DataSource")\
.option("url", url) \
.option("authentication.type", "basic") \
.option("authentication.basic.username", username) \
.option("authentication.basic.password", password) \
.option("relationship", "ANAMNESI") \
.option("relationship.save.strategy", "keys") \
.option("relationship.source.labels", ":Sintomo") \
.option("relationship.source.save.mode", "overwrite") \
.option("relationship.source.node.keys", "nome") \
.option("relationship.target.labels", ":Visita") \
.option("relationship.target.node.keys", "id") \
.option("relationship.target.save.mode", "overwrite") \
.mode("Overwrite") \
.save()
```

Si definisce la relazione *DIAGNOSSI*, orientata dalla visita al sintomo.

Definizione relazione tra sintomi di diagnosi e visite

```
symptom_diagnosi = entities_diagnosi.filter("entity == 'Symptom'") \
    .select("ID_VISITA", "chunk") \
    .withColumnRenamed("chunk", "nome") \
    .withColumnRenamed("ID_VISITA", "id")

symptom_diagnosi.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("relationship", "DIAGNOSI") \
    .option("relationship.save.strategy", "keys") \
    .option("relationship.source.labels", ":Visita") \
    .option("relationship.source.save.mode", "overwrite") \
    .option("relationship.source.node.keys", "id") \
    .option("relationship.target.labels", ":Sintomo") \
    .option("relationship.target.node.keys", "nome") \
    .option("relationship.target.save.mode", "overwrite") \
    .mode("Overwrite") \
    .save()
```

4.2.3.5 Farmaco - Visita

Si definisce la relazione tra i nodi *Farmaco* e *Visita*, a cui è associata l'etichetta *PRESCRITTO*. A tale relazione sono assegnate due proprietà: la *confezione* del farmaco, coincidente con il nome originale, e la *dose* prescritta dal medico.

Definizione relazione tra farmaci e visite

```
drugs.write.format("org.neo4j.spark.DataSource") \
    .option("url", url) \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", username) \
    .option("authentication.basic.password", password) \
    .option("relationship", "PRESCRITTO") \
    .option("relationship.save.strategy", "keys") \
    .option("relationship.source.labels", ":Visita") \
    .option("relationship.source.save.mode", "overwrite") \
    .option("relationship.source.node.keys", "ID_VISITA:id") \
    .option("relationship.target.labels", ":Farmaco") \
    .option("relationship.target.node.keys", "FARMACO_FINAL:nome") \
    .option("relationship.target.save.mode", "overwrite") \
    .option("relationship.properties", "CONFEZIONE:confezione, DOSE:dose") \
    .mode("Append") \
    .save()
```

Capitolo 5

Data Visualization

Nel capitolo corrente si procede ad illustrare lo step conclusivo di visualizzazione dei dati, nel contesto del ciclo di sviluppo dell'elaborato. Tale fase è stata supportata da due strumenti di BI quali **PowerBI** e **Neo4j Bloom**.

5.1 PowerBI

Power BI è un noto servizio di Business Analytics prodotto da Microsoft, utilizzato ampiamente in ambito aziendale per applicazioni di Business Intelligence. L'obiettivo dell'analisi condotta con tale strumento è quello di mostrare le informazioni di maggiore interesse relative all'elaborato, attraverso un'interfaccia interattiva, semplice ed intuitiva.

Sono state progettate alcune **dashboard** che analizzano i dati da punti di vista diversi. In questo modo si è riusciti ad andare incontro alle varie esigenze che caratterizzano tipologie di utenti differenti.

5.1.1 Caricamento dei dati

Al fine di importare i dati presenti sul database Neo4j in PowerBI è stato utilizzato il connettore reperibile al repository GitHub [5]. Tale strumento offre la possibilità di effettuare query sul database in linguaggio **Cypher** e di importare i risultati delle interrogazioni in PowerBI.

Di seguito sono riportate le diverse sorgenti di dati e le relative query che ne permettono il caricamento.

5.1.1.1 Visite

Si caricano tutti i dati relativi alle visite, inclusi medico e pazienti coinvolti.

Query Cypher

```
match (p:Paziente)-[:SI_SOTTOPONE]-(v:Visita)
optional match (m:Medico)-[:EFFETTUÀ]-(v)
return p,v,m
```

5.1.1.2 Malattie

Si caricano tutti i dati relativi alle malattie, presenti sia in anamnesi che in diagnosi.

Query Cypher

```
match (m:Malattia) return m
```

5.1.1.3 Sintomi

Si caricano tutti i dati relativi ai sintomi, presenti sia in anamnesi che in diagnosi.

Query Cypher

```
match (s:Sintomo) return s
```

5.1.1.4 Farmaci

Si caricano tutti i dati relativi ai farmaci.

Query Cypher

```
MATCH (f:Farmaco) return f
```

5.1.1.5 Malattie Anamnesi

Si caricano tutte le malattie presenti nell'anamnesi di ogni visita.

Query Cypher

```
match (v:Visita)-[a:ANAMNESI]-(m:Malattia) return v.id,m
```

5.1.1.6 Malattie Diagnosi

Si caricano tutte le malattie presenti nella diagnosi di ogni visita.

Query Cypher

```
MATCH (v:Visita)-[d:DIAGNOSI]-(m:Malattia) RETURN v.id,m
```

5.1.1.7 Sintomi Anamnesi

Si caricano tutti i sintomi presenti nell'anamnesi di ogni visita.

Query Cypher

```
match (v:Visita)-[a:ANAMNESI]-(s:Sintomo) return v.id,s
```

5.1.1.8 Sintomi Diagnosi

Si caricano tutti i sintomi presenti nelle diagnosi di ogni visita.

Query Cypher

```
match (v:Visita)-[d:DIAGNOSI]-(s:Sintomo) return v.id,s
```

5.1.1.9 Farmaci e Visite

Si caricano tutti i farmaci prescritti in ogni visita.

Query Cypher

```
match (v:Visita)-[p:PRESCRIBE]-(f:Farmaco) return v.id,p,f
```

5.1.1.10 Malattie Diagnosi e Farmaci

Si caricano tutte le malattie diagnosticate insieme a tutti i farmaci prescritti in ogni visita, al fine di analizzare la correlazione tra le due entità.

Query Cypher

```
match r=(m:Malattia)-[:DIAGNOSI]-(v:Visita)-[:PRESCRIBE]-(f:Farmaco) return v.id,m,f
```

5.1.1.11 Malattie Diagnosi e Sintomi Anamnesi

Si caricano tutte le malattie diagnosticate insieme a tutti i sintomi accusati dal paziente in ogni visita, al fine di analizzare la correlazione tra le due entità.

Query Cypher

```
match (m:Malattia)-[:DIAGNOSI]-(v:Visita)-[:ANAMNESI]-(s:Sintomo) return v.id,m,s
```

5.1.2 Modello dei dati

Dopo aver caricato i dati in PowerBI sono stati effettuati solo alcuni piccoli accorgimenti, quali il riconoscimento del formato dei campi delle tabelle. A valle di tali operazioni si ottiene quindi un modello dei dati coerente e completo, riportato in Figura 5.1.

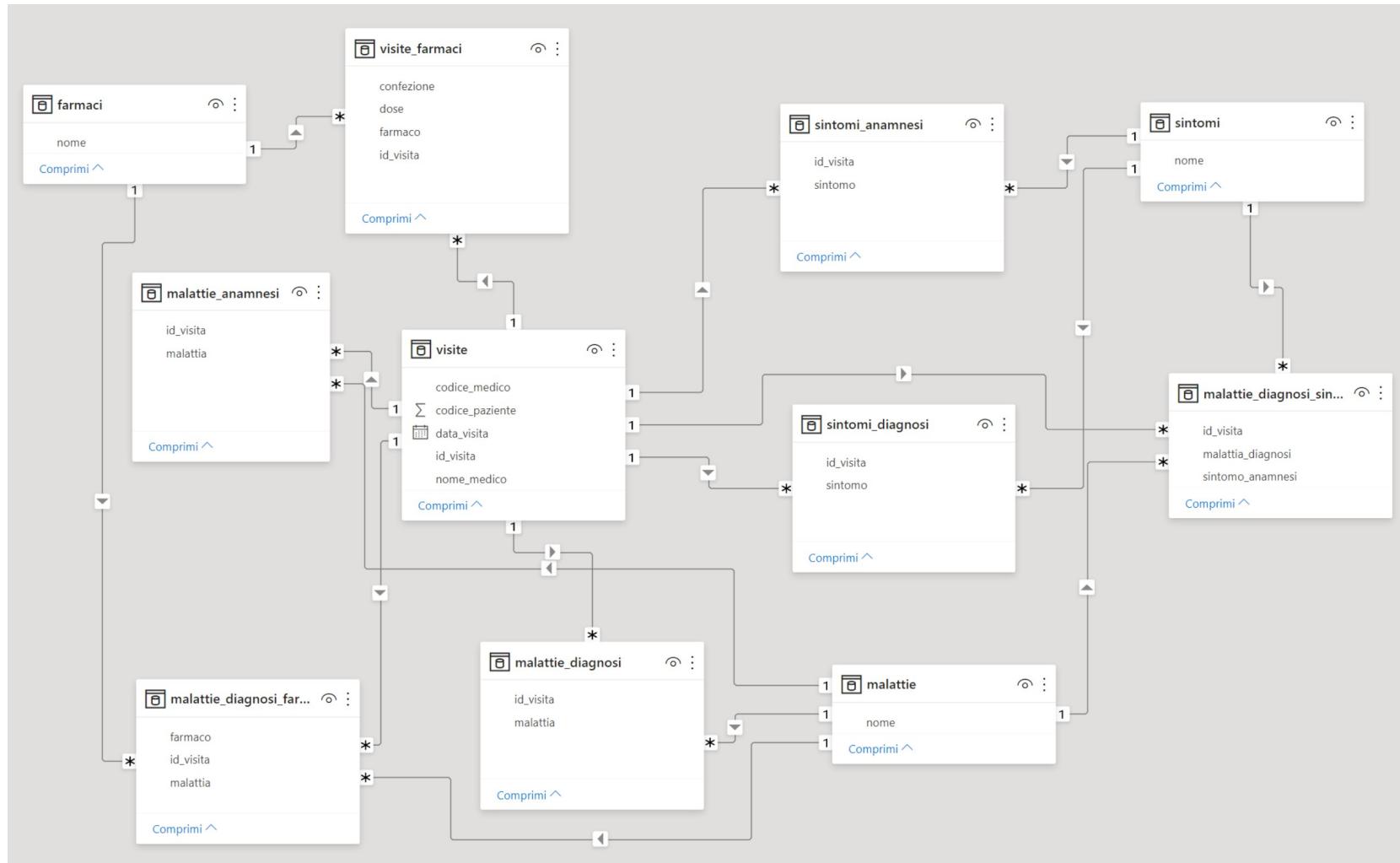


Figura 5.1: Modello dei dati

5.1.3 Dashboard Medici

La prima dashboard realizzata, riportata in Figura 5.2, riporta una panoramica relativa ai medici esaminatori che hanno condotto le visite cardiologiche. Le informazioni raccolte sono organizzate in vari grafici particolarmente intuitivi, in modo da rendere fruibile a un ipotetico utente (*e.g.* un amministratore ospedaliero) il risultato dell'analisi di BI condotta sui medici.

In particolare, all'interno della dashboard sono mostrati:

- L'*andamento mensile delle visite effettuate*, al fine di evidenziare quali siano i periodi con una densità di visite elevata o scarsa. In particolare, si osserva come i mesi estivi estivi di luglio e agosto sono i periodi in cui si registra il minor numero di visite.
- L'*istogramma del numero di visite effettuate da ogni medico*.
- L'*istogramma del numero di pazienti visitati da ogni medico*.
- L'*istogramma del numero di farmaci prescritti da ogni medico*.
- Il *diagramma a torta riportante i principali 10 farmaci prescritti*, in modo da visualizzare i 10 farmaci più prescritti in generale, ma anche per analizzare tale risultato rispetto a uno specifico medico.
- Una *scheda di riepilogo*, contenente informazioni complessive riguardanti il numero totale di visite, di pazienti e di farmaci coinvolti nei grafici.
- Una *tabella* riportante il codice e il nome dei medici visualizzati nei grafici, in modo da rendere facilmente reperibile il nome del medico a partire dal codice univoco che lo caratterizza.

Inoltre, è stata data all'utente la possibilità di interagire attivamente con i grafici realizzati, prevedendo all'interno della dashboard tre filtri:

- Un *filtro temporale*, che permette di specificare l'arco temporale su cui visualizzare le informazioni riportate nei grafici.
- Un *filtro sui farmaci*, che permette di selezionare uno specifico sottoinsieme di farmaci da considerare nell'analisi.
- Un *filtro sui medici*, che permette di selezionare uno specifico sottoinsieme di medici di cui visualizzarne le informazioni nei grafici.

Tali filtri sono applicati a tutti gli elementi visivi della dashboard, ad eccezione della tabella con codici e nomi dei medici.

Dashboard Medici

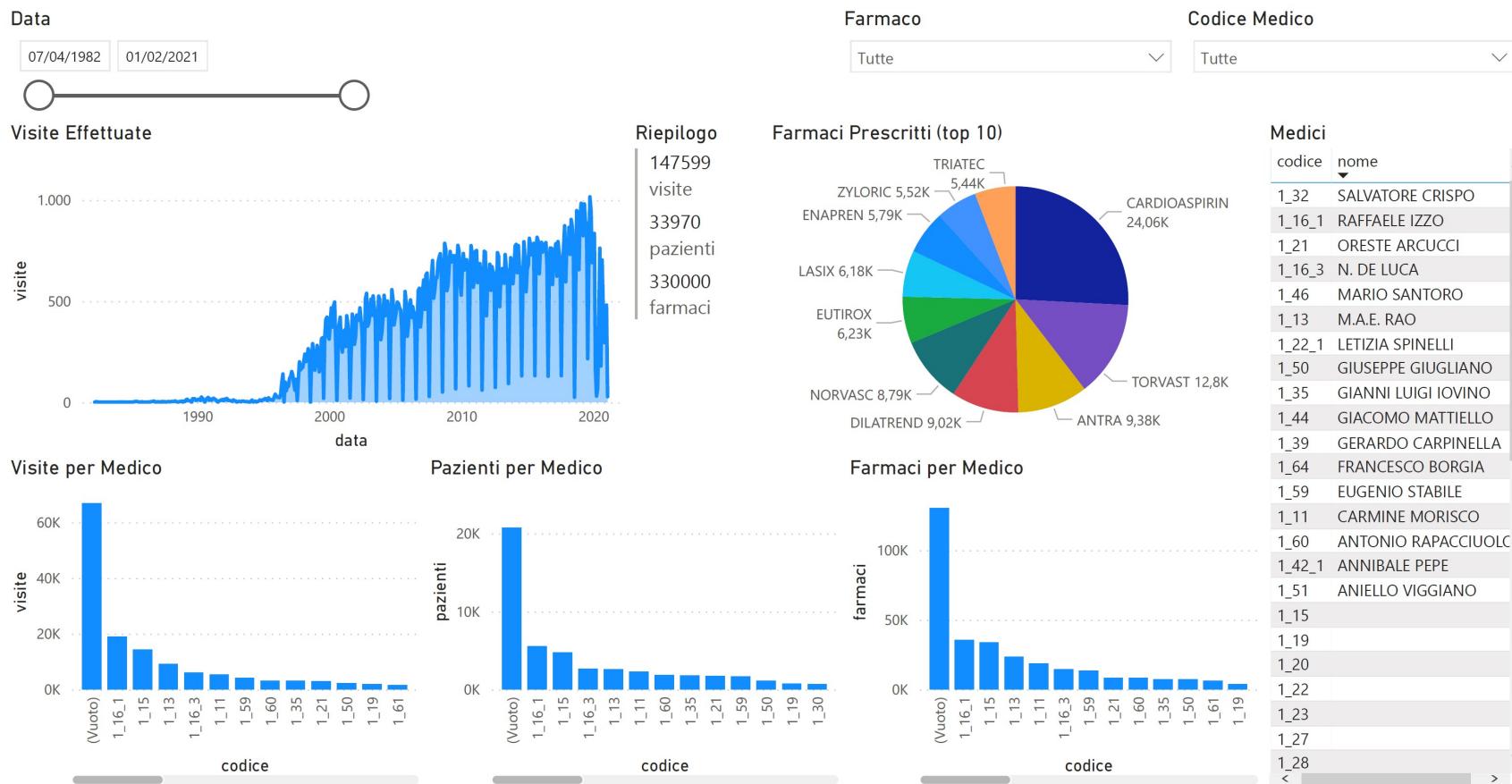


Figura 5.2: Dashboard Medici

5.1.4 Dashboard Pazienti

Tale dashboard, riportata in Figura 5.3, contiene un report riguardante i pazienti sottoposti alle visite cardiologiche. Le informazioni raccolte sono articolate in un insieme di grafici che permettono ad un ipotetico utente (*e.g.* un medico) di tenere sotto controllo le informazioni salienti che caratterizzano un paziente.

L'utente può interagire attivamente con i grafici, agendo tramite due filtri:

- Un *filtro sul paziente*, per selezionare lo specifico paziente di cui visualizzare le informazioni.
- Un *filtro temporale*, che permette di specificare l'arco temporale su cui visualizzare le informazioni raccolte nei grafici.

Tali filtri sono applicati a tutti gli elementi visivi della dashboard.

I grafici che compongono la dashboard sono invece i seguenti:

- Un *grafico a cascata riportante le visite differenti a cui si è sottoposto il paziente*, con relativa data, ed il numero di visite totali.
- Un *diagramma a torta contenente i vari medici che hanno visitato il paziente*, da cui è possibile reperire codice e nome.
- Un *diagramma a torta riportante i farmaci assunti dal paziente*, con il relativo numero di prescrizioni.
- Un *report di riepilogo*, in cui è possibile visualizzare tutte le informazioni relative al paziente selezionato, ponendo semplici domande in linguaggio naturale.
- 4 *mappe ad albero*, in cui sono riportati malattie e sintomi relative ad anamnesi e diagnosi del paziente. In particolare, in ogni diagramma è possibile distinguere le diverse visite in cui sono stati individuati malattie e sintomi.

Dashboard Pazienti



Figura 5.3: Dashboard Pazienti

5.1.5 Dashboard Sintomi, Malattie e Farmaci

L'ultima dashboard, riportata in Figura 5.4, comprende una panoramica generale riguardante i sintomi, le malattie e i farmaci contenuti nei dati analizzati. Tali informazioni, organizzate in diversi grafici, permettono a un ipotetico utente (e.g. un medico o un amministratore) di pervenire ad una visione sintetica dei risultati ottenuti tramite l'analisi di BI.

Nel dettaglio, i grafici che compongono la dashboard sono:

- Un *grafico a barre riportante il numero di occorrenze dei sintomi nelle anamnesi e nelle diagnosi*.
- Un *grafico a barre riportante il numero di occorrenze delle malattie nelle anamnesi e nelle diagnosi*.
- Un *grafico a barre riportante il numero di occorrenze dei farmaci prescritti*.
- Un *diagramma a torta riportante le 3 principali malattie diagnosticate rispetto a uno o più sintomi di anamnesi*, in modo da evidenziare quali malattie sono legate a determinati sintomi.
- Un *diagramma a torta riportante i 3 principali farmaci prescritti per una o più malattie di diagnosi*, al fine di rilevare quali farmaci sono prescritti in seguito ad una specifica diagnosi medica.
- Un *diagramma a torta riportante le 3 principali malattie diagnosticate rispetto a uno o più farmaci prescritti*, in modo da individuare le malattie che un farmaco è in grado di curare.

Inoltre, la dashboard presenta 3 filtri con cui l'utente è in grado di interagire:

- Un *filtro sul sintomo*, in modo da selezionare uno o più sintomi di anamnesi di cui si vogliono visualizzare le informazioni. Tale filtro è applicato solo ai grafici della colonna di sinistra, relativi appunto ai sintomi.
- Un *filtro sulla malattia*, per selezionare una o più malattie di diagnosi a cui si è interessati. Tale filtro è applicato solo ai grafici della colonna centrale, relativi appunto alle malattie.
- Un *filtro sul farmaco*, al fine di selezionare uno o più farmaci. Tale filtro è applicato solo ai grafici della colonna di destra, relativi appunto ai farmaci.

Dashboard Sintomi, Malattie e Farmaci

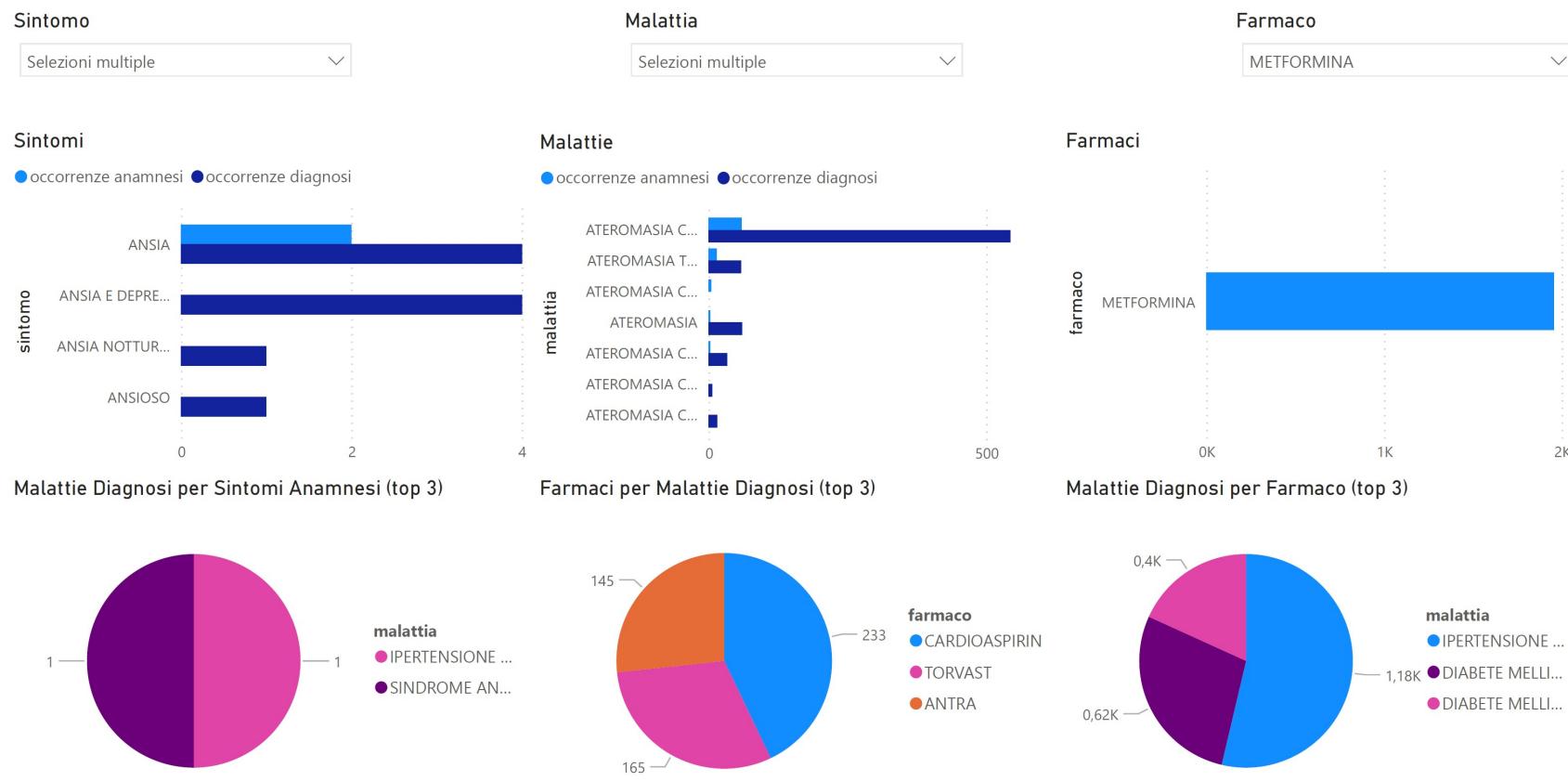


Figura 5.4: Dashboard Sintomi, Malattie e Farmaci

5.2 Neo4j Bloom

Neo4j Bloom è un'applicazione per l'esplorazione di grafi modellati in Neo4j. Essa offre la possibilità di investigare ed esplorare visualmente i grafi modellati anche a non esperti, tramite la sua interfaccia semplice ed intuitiva. Allo stesso tempo consente di definire opportune *frasi di ricerca* alle quali è possibile associare query in linguaggio **Cypher** per restituire particolari path di interesse nel grafo. Ogni frase di ricerca non è altro che una interrogazione del database in linguaggio naturale che consente all'utente di ottenere una particolare *prospettiva* dello stesso. Grazie alle query definite e tramite le frasi di ricerca gli utenti possono scrivere quindi semplici interrogazioni in linguaggio naturale senza necessità di conoscere il linguaggio Cypher.

L'obiettivo dell'utilizzo di tale strumento nel contesto dell'elaborato è quello di configurarlo opportunamente per l'utilizzo da parte di uno o più ipotetici utenti finali, come ad esempio un medico o un amministratore dell'ospedale.

5.2.1 Query Pazienti

In questa sezione si riportano le query realizzate in Neo4j Bloom per l'esplorazione e l'analisi di informazioni legate ai pazienti.

5.2.1.1 Cartella clinica di un paziente

La query consente di ottenere, dato il codice di un paziente, tutte le informazioni relative alla sua storia clinica. Le informazioni restituite includono: tutte le visite effettuate, tutte le malattie ed i sintomi rilevati per il paziente, distinti per fase di anamnesi e diagnosi, e tutti i farmaci che gli sono stati prescritti.

La frase di ricerca definita per tale query è `Cartella clinica del paziente $cod_paz`, dove il parametro `$cod_paz` è il codice identificativo del paziente.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[sott:SI_SOTTOPONE]->(v:Visita), (v)-[r]-(n)
where p.codice= $cod_paz
return *
```

Neo4j Bloom consente inoltre di specificare il formato dei parametri usati nelle query, di modo da fornire opportuni suggerimenti in fase di interrogazione del database e facilitando ulteriormente gli utenti che ne fanno uso. Ad esempio, in questo caso si è specificato come formato di `$cod_paz` il formato intero, e lo si è associato alla proprietà *codice* dei nodi *Paziente*. Tale operazione è stata ripetuta per tutti i parametri delle query implementate.

In Figura 5.5 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

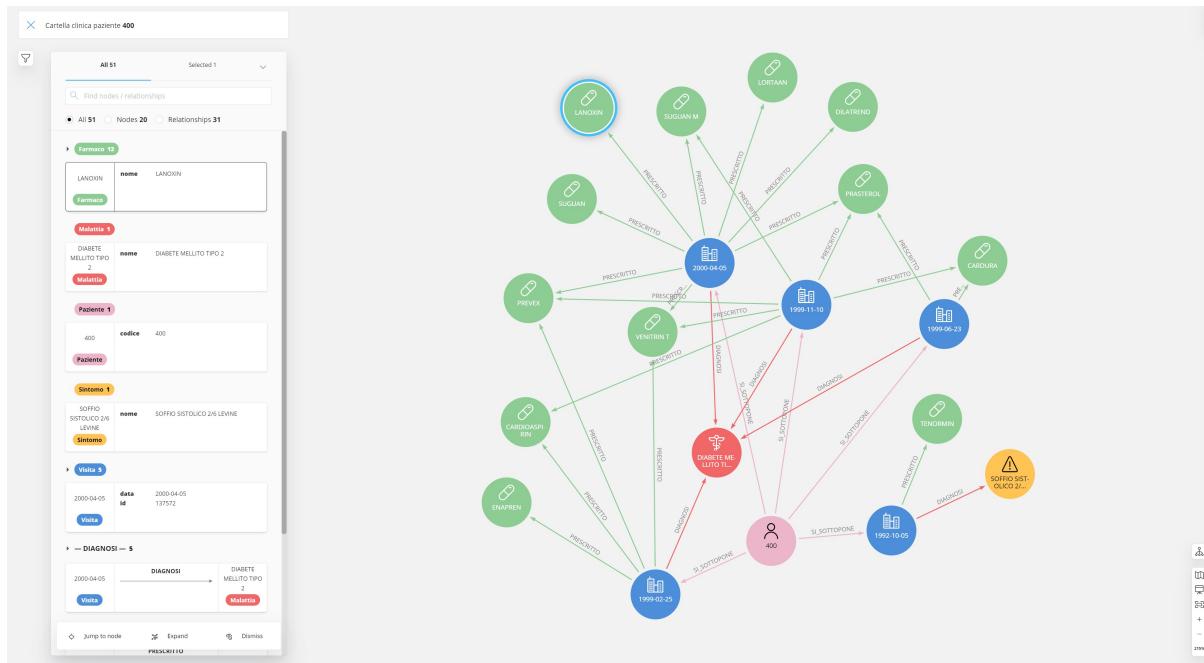


Figura 5.5: Cartella clinica del paziente 400

Come si nota dalla Figura 5.5 l'utente è in grado di esplorare con estrema semplicità il grafo ed i suoi componenti grazie alla ulteriore presenza del pannello a sinistra. Grazie ad esso infatti è possibile osservare tutte le proprietà di un nodo o di una relazione.

5.2.1.2 Visite effettuate da un certo paziente

La query consente di ottenere, dato il codice di un paziente, tutte visite a cui egli si è sottoposto. Le informazioni restituite includono: tutte le visite a cui si è sottoposto il paziente e, laddove possibile, il medico che ha eseguito ciascuna visita. Questo perché l'informazione relativa al medico che esegue una visita non è sempre disponibile.

La frase di ricerca definita per tale query è Visite del paziente \$cod_paz.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[s:SI_SOTTOPONE]-> (v:Visita)
where p.codice = $cod_paz
optional match (v)<- [e:EFFETTUA]-(m:Medico)
return *
```

In Figura 5.6 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

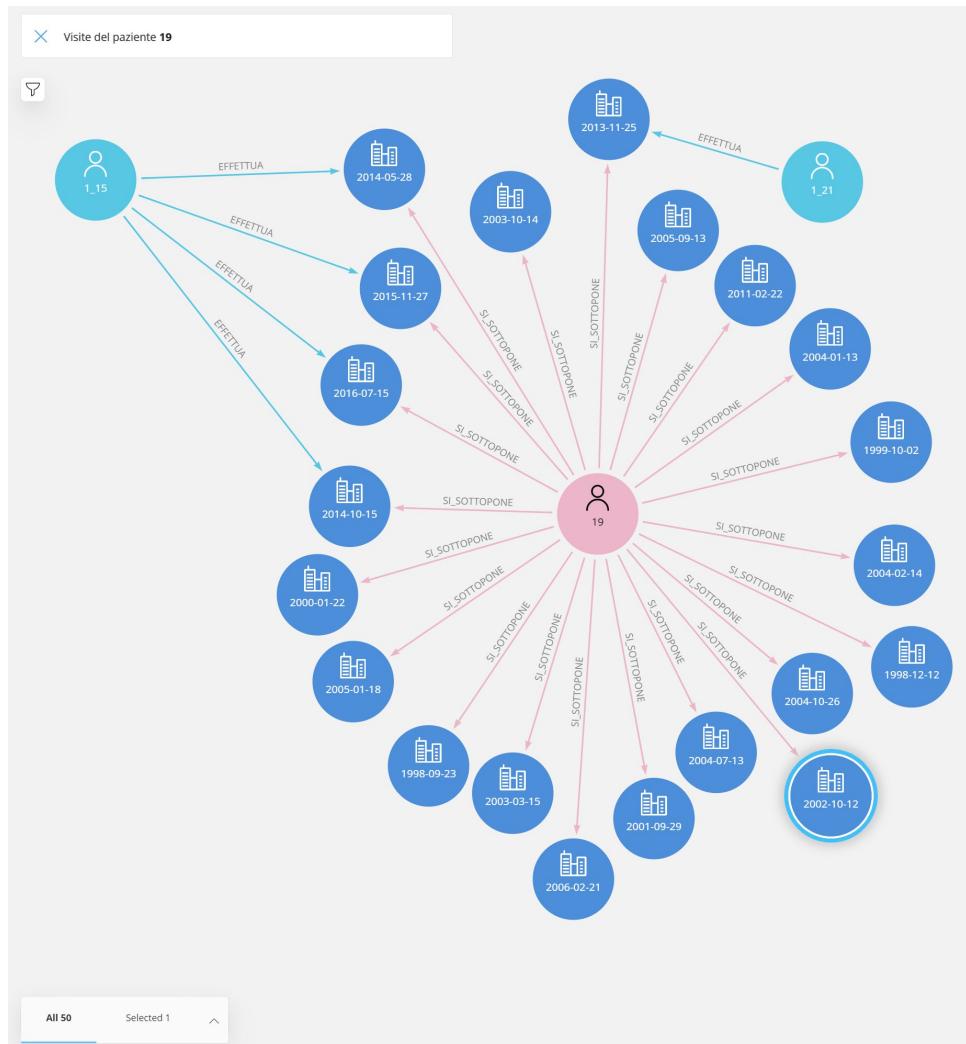


Figura 5.6: Visite effettuate dal paziente 19

5.2.1.3 Pazienti affetti da una certa malattia

La query consente di ottenere, dato il nome della malattia, tutti i pazienti a cui essa è stata diagnosticata. Si è scelto in questo caso di far restituire solo i nodi Paziente.

La frase di ricerca definita per tale query è `Pazienti con malattia $malattia`, dove il parametro `$malattia` è il suo nome.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[:SI_SOTTOPONE]->(:Visita)-[:DIAGNOSI]->(m:Malattia)
where m.nome = $malattia
return p
union
match (p:Paziente)-[:SI_SOTTOPONE]->(:Visita)<-[:ANAMNESI]-(m:Malattia)
where m.nome = $malattia
return p
```

Come si nota dalla query Cypher, si è scelto di considerare le malattie individuate sia in fase di anamnesi che in quella di diagnosi del paziente.

In Figura 5.7 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

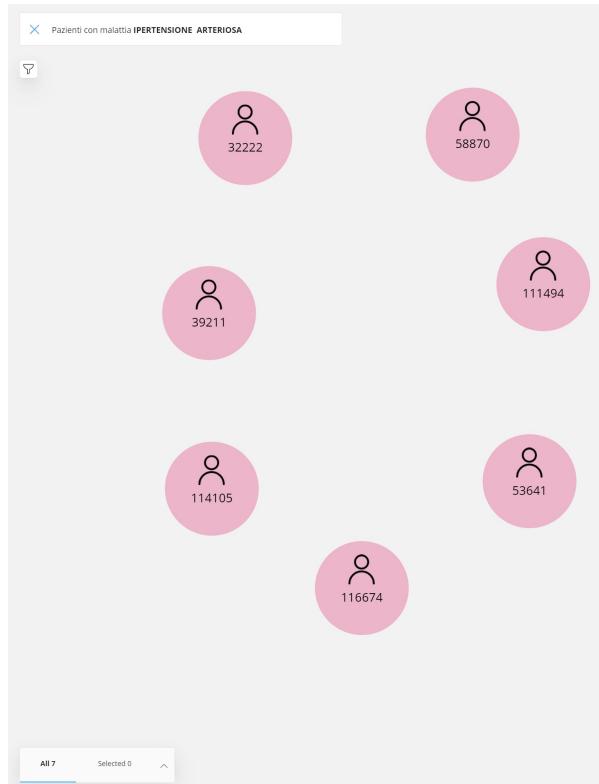


Figura 5.7: Pazienti con malattia Ipertensione Arteriosa

5.2.1.4 Malattie di un certo paziente

La query consente di ottenere, dato il codice del paziente, tutte le malattie che gli sono state diagnosticate. Le informazioni restituite includono: tutte le visite in cui sono state diagnosticate malattie al paziente e le relative malattie diagnosticate.

La frase di ricerca definita per tale query è `Malattie del paziente $cod_paz.`

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[s:SI_SOTTOPONE]->(v:Visita)-[r]-(m:Malattia)
where p.codice = $cod_paz
return *
```

Come si nota dalla query Cypher, si è scelto di considerare le malattie individuate sia in fase di anamnesi che in quella di diagnosi del paziente.

In Figura 5.8 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

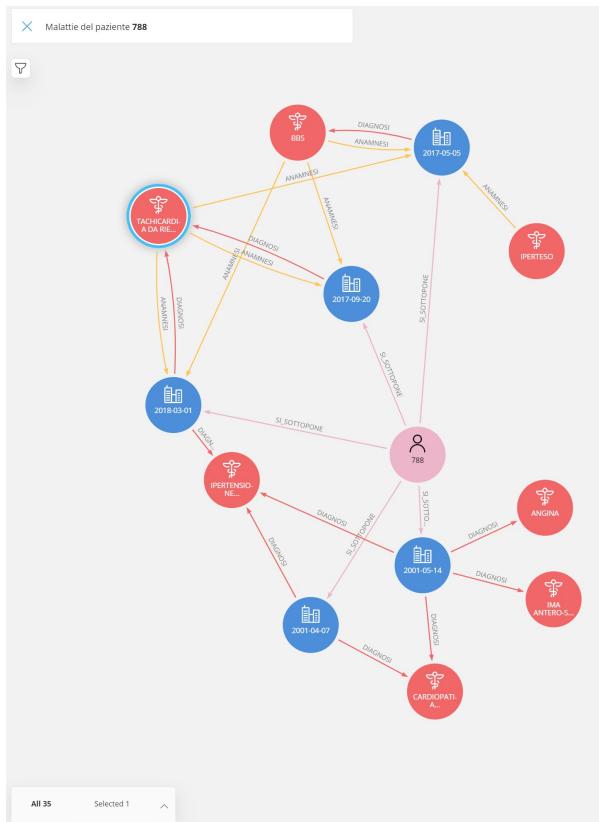


Figura 5.8: Malattie del paziente 788

5.2.1.5 Pazienti che mostrano un certo sintomo

La query consente di ottenere, dato il nome del sintomo, tutti i pazienti che lo mostrano. Si è scelto in questo caso di far restituire solo i nodi Paziente.

La frase di ricerca definita per tale query è `Pazienti con sintomo $sintomo`, dove il parametro `$sintomo` è il suo nome.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[sott:SI_SOTTOPONE]->(v:Visita)-[r]-(s:Sintomo)
where s.nome = $sintomo
return p
```

Come si nota dalla query Cypher, si è scelto di considerare i sintomi individuati sia in fase di anamnesi che in quella di diagnosi del paziente.

In Figura 5.9 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

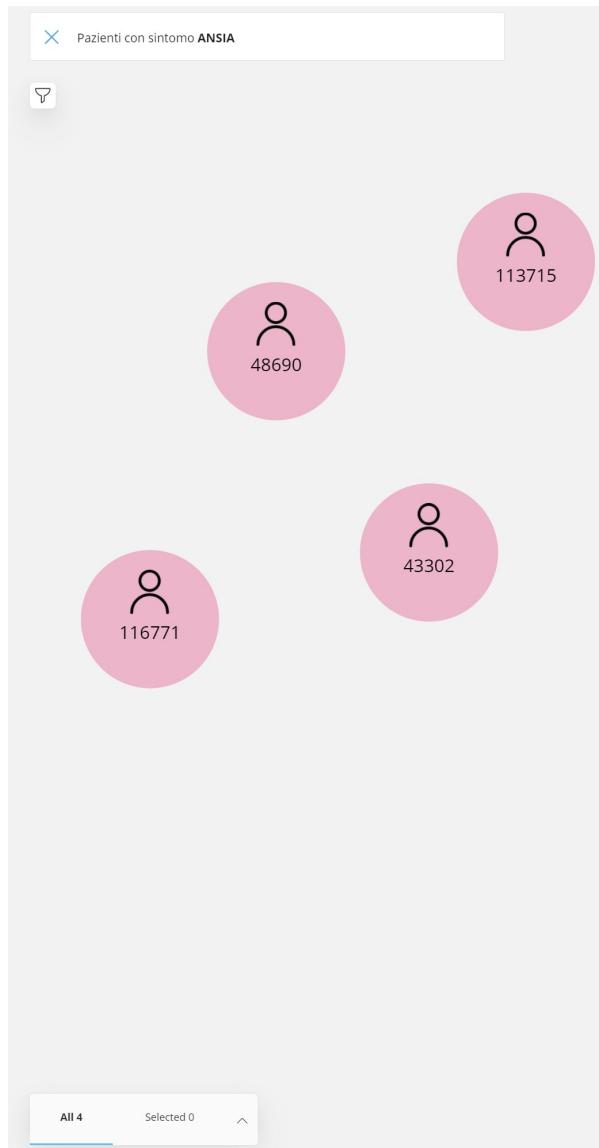


Figura 5.9: Pazienti con sintomo Ansia

5.2.1.6 Sintomi mostrati da un certo paziente

La query consente di ottenere, dato il codice del paziente, tutti i sintomi che mostra. Le informazioni restituite includono: tutte le visite in cui sono stati mostrati sintomi dal paziente e i relativi sintomi.

La frase di ricerca definita per tale query è `Sintomi del paziente $cod_paz.`

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[sott:SI_SOTTOPONE]->(v:Visita)-[r]-(s:Sintomo)
where p.codice = $cod_paz
return *
```

Come si nota dalla query Cypher, si è scelto di considerare i sintomi individuati sia in fase di anamnesi che in quella di diagnosi del paziente.

In Figura 5.10 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

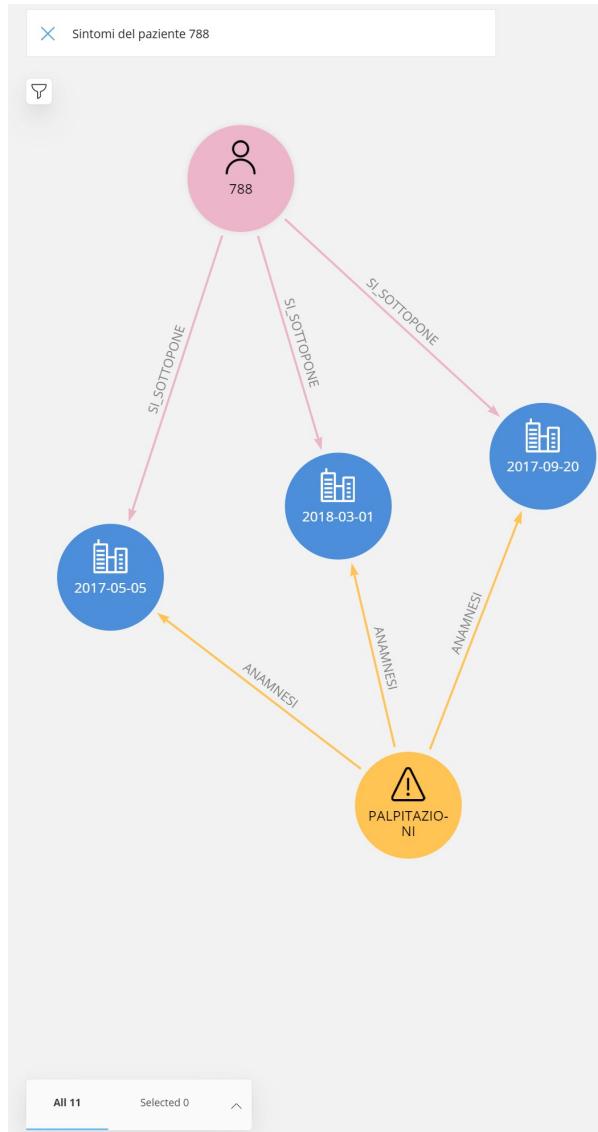


Figura 5.10: Sintomi del paziente 788

5.2.1.7 Pazienti che assumono un certo farmaco

La query consente di ottenere, dato il nome del farmaco, tutti i pazienti che lo assumono. Si è scelto in questo caso di far restituire solo i nodi Paziente.

La frase di ricerca definita per tale query è `Pazienti che assumono il farmaco $farmaco`, dove il parametro `$farmaco` è il suo nome.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[:SI_SOTTOPONE]->(v:Visita)-[:PRESCRITTO]->(f:Farmaco)  
where f.nome = $farmaco  
return p
```

In Figura 5.11 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

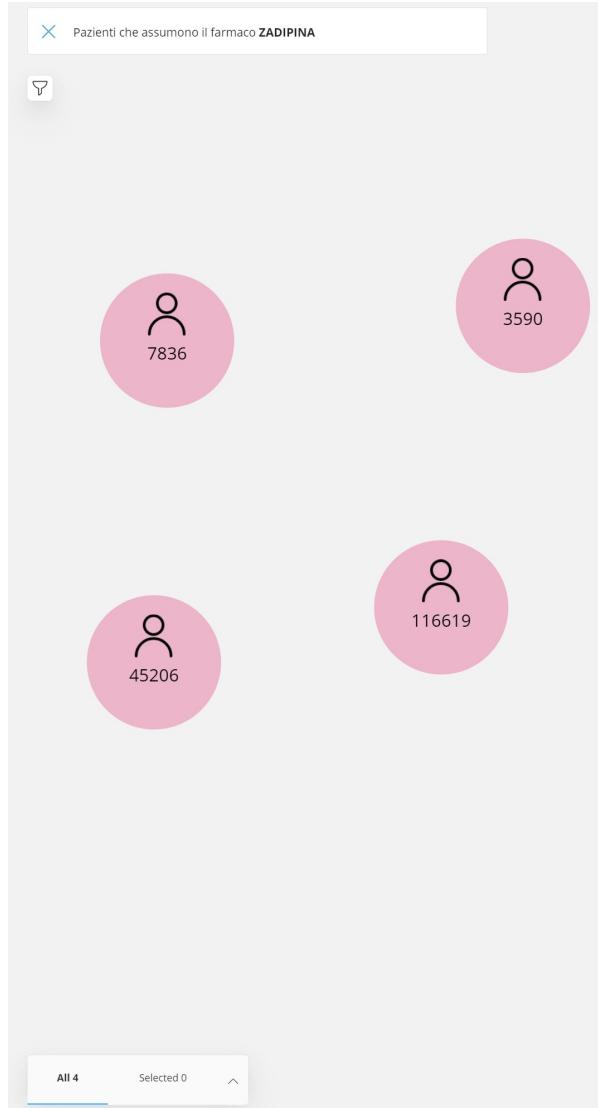


Figura 5.11: Pazienti che assumono il farmaco Zadipina

5.2.1.8 Farmaci assunti da un certo paziente

La query consente di ottenere, dato il codice del paziente, tutti i farmaci che assume. Le informazioni restituite includono: tutte le visite in cui sono stati prescritti farmaci al paziente e i relativi farmaci.

La frase di ricerca definita per tale query è Farmaci assunti dal paziente \$cod_paz.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[s:SI_SOTTOPONE]->(v:Visita)-[pre:PRESCRITTO]->(f:Farmaco)
where p.codice = $cod_paz
return *
```

In Figura 5.12 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

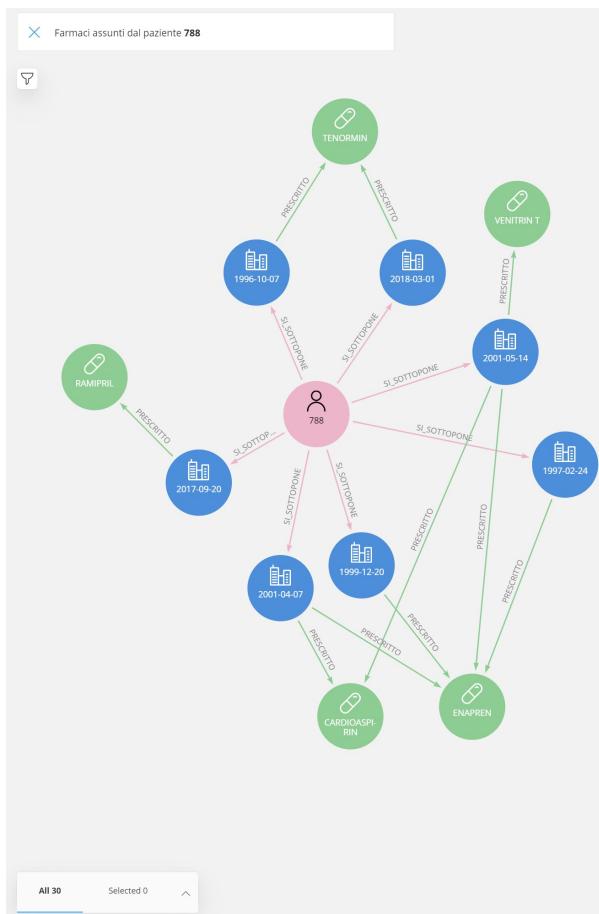


Figura 5.12: Farmaci assunti dal paziente 788

5.2.1.9 Visite del paziente tra due date

La query consente di ottenere, dato il codice del paziente e due date, tutte le visite a cui egli si è sottoposto nell'arco di tempo specificato.

La frase di ricerca definita per tale query è Visite del paziente \$cod_paz tra \$date1 e \$date2.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
MATCH (v:Visita) <-[s:SI_SOTTOPONE] - (p:Paziente)
WHERE date($date1) <= v.data <= date($date2) and p.codice=$cod_paz
RETURN *
```

In Figura 5.13 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

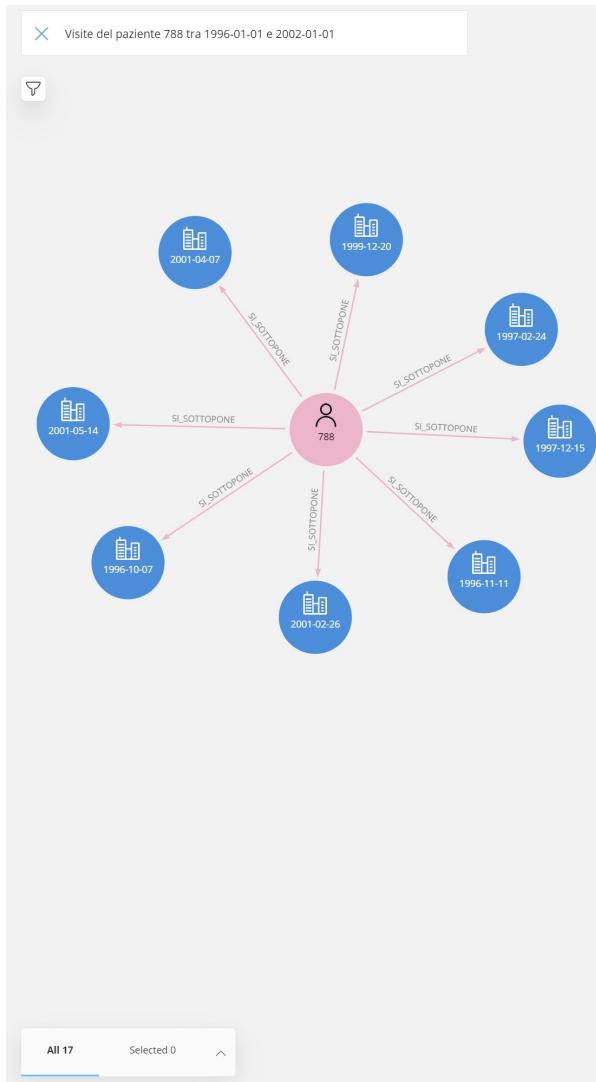


Figura 5.13: Visite del paziente 788 tra 1996-01-01 e 2002-01-01

5.2.2 Query Medici

In questa sezione si riportano le query realizzate in Neo4j Bloom per l'esplorazione e l'analisi di informazioni legate ai medici.

5.2.2.1 Pazienti visitati da un certo medico

La query consente di ottenere, dato il medico, tutti i pazienti che esso ha visitato. Le informazioni restituite includono: tutte le visite effettuate dal medico e tutti i pazienti che ha visitato.

La frase di ricerca definita per tale query è **Pazienti visitati dal medico \$medico**, dove il parametro **\$medico** può essere sia il codice identificativo del medico, sia il nome del medico.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (p:Paziente)-[s:SI_SOTTOPONE]->(v:Visita)<-[e:EFFETTUA]-(m:Medico)
where m.nome = $medico or m.codice = $medico
return *
```

In Figura 5.14 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

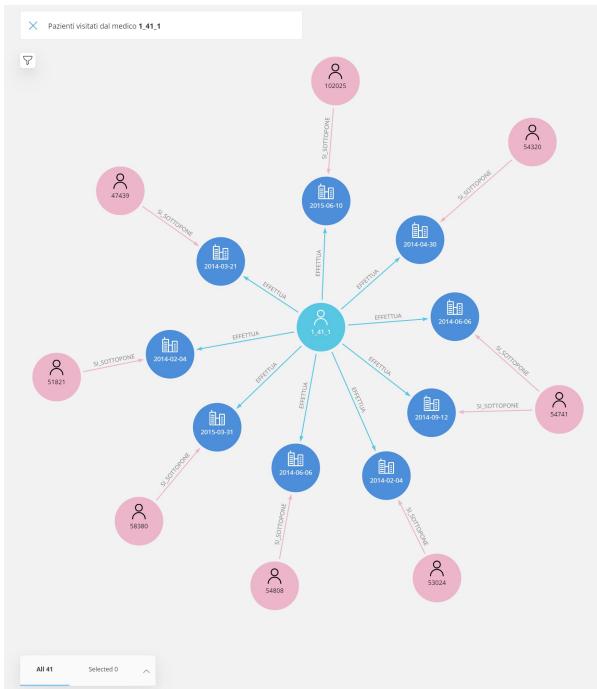


Figura 5.14: Pazienti visitati dal medico 1_41_1

5.2.2.2 Farmaci prescritti da un certo medico

La query consente di ottenere, dato il medico, tutti i pazienti che esso ha visitato. Si è scelto in questo caso di far restituire solo i nodi Farmaco.

La frase di ricerca definita per tale query è **Farmaci prescritti dal medico \$medico**, dove il parametro **\$medico** può essere sia il codice identificativo del medico, sia il nome del medico.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (m:Medico) -[e:EFFETTUA]-> (v:Visita) -[p:PRESCRITTO]-> (f:Farmaco)
where m.codice = $medico or m.nome = $medico
return f
```

In Figura 5.15 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

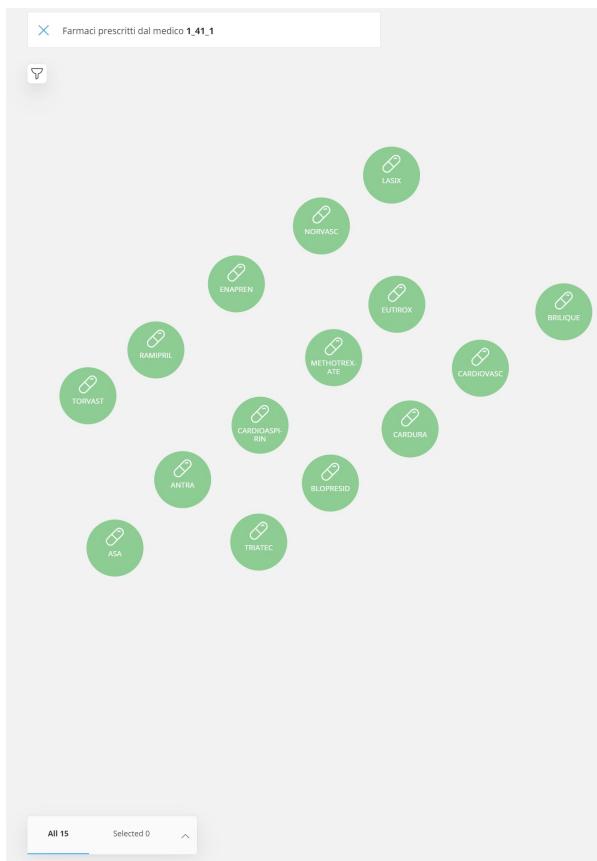


Figura 5.15: Farmaci prescritti dal medico 1_41_1

5.2.2.3 Visite di un certo medico tra due date

La query consente di ottenere, dato il medico e due date, tutte le visite che ha effettuato nell'arco temporale specificato. Si è scelto in questo caso di far restituire solo i nodi Farmaco.

La frase di ricerca definita per tale query è Visite del medico \$medico tra \$date1 e \$date2, dove il parametro \$medico può essere sia il codice identificativo del medico, sia il nome del medico.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (v:Visita)-[e:EFFETTUA]-(m:Medico)  
where date($date1) <= v.data <= date($date2) and (m.codice=$medico or m.nome=$medico)  
return *
```

In Figura 5.16 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

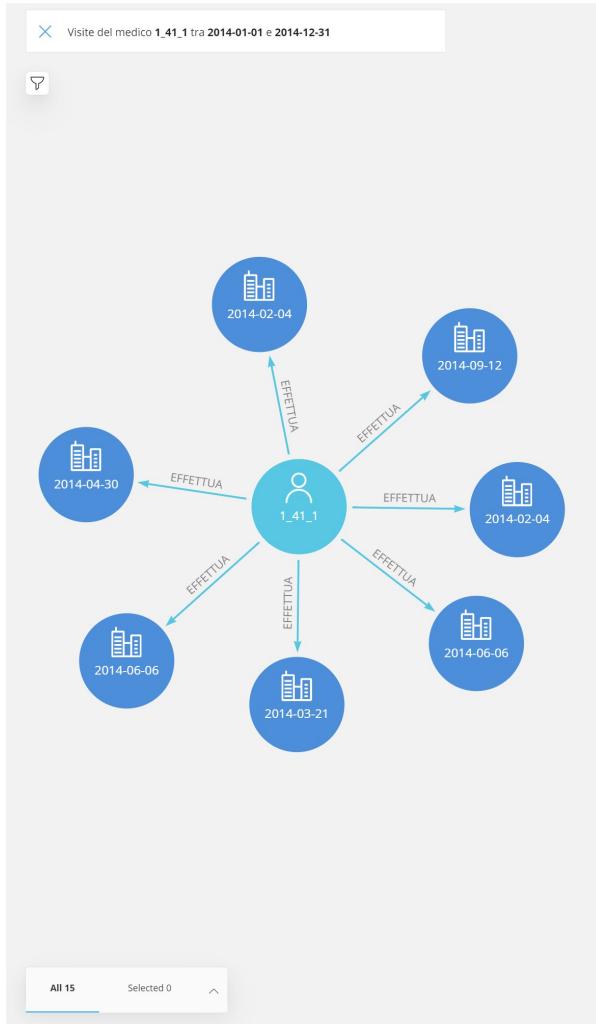


Figura 5.16: Visite del medico 1_41_1 tra 2014-01-01 e 2014-12-31

5.2.3 Query su informazioni generali

In questa sezione si riportano le query realizzate in Neo4j Bloom per l'esplorazione e l'analisi di informazioni generali contenute nel grafo costruito.

5.2.3.1 Visite tra due date

La query consente di ottenere, date due date, tutte le visite effettuate nell'arco temporale specificato. Si è scelto in questo caso di far restituire solo i nodi Visita.

La frase di ricerca definita per tale query è Visite tra \$date1 e \$date2.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (v:Visita)
where date($date1) <= v.data < date($date2)
return v
```

In Figura 5.17 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.



Figura 5.17: Visite effettuate tra 2020-12-01 e 2020-12-2

5.2.3.2 Malattie dei pazienti a cui è somministrato un certo farmaco

La query consente di ottenere, dato il nome del farmaco, tutte le malattie di cui sono affetti i pazienti che assumono tale farmaco. Le informazioni restituite includono: tutte le visite in cui è stato somministrato il farmaco e le malattie associate alla visita.

La frase di ricerca definita per tale query è **Malattie dei pazienti a cui è somministrato il farmaco \$farmaco**.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (m:Malattia)-[r]->(v:Visita)-[p:PRESCRITTO]->(f:Farmaco {nome: $farmaco})
return *
```

Come si nota dalla query Cypher, si è scelto di considerare le malattie individuate sia in fase di anamnesi che in quella di diagnosi del paziente. È bene inoltre precisare che non è possibile concludere che il farmaco ricercato sia utilizzato per curare tutte le malattie restituite dalla query.

In Figura 5.18 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

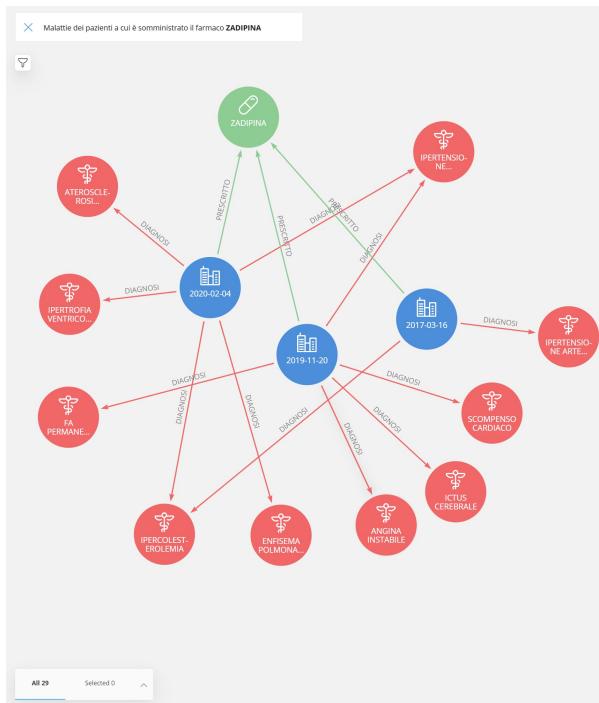


Figura 5.18: Malattie dei pazienti a cui è somministrato il farmaco Zadipina

5.2.3.3 Farmaci prescritti ai pazienti con una certa malattia

La query consente di ottenere, dato il nome della malattia, tutti i farmaci assunti dai pazienti affetti da essa. Le informazioni restituite includono: tutte le visite in cui è stata

individuata la malattia e tutti i farmaci associati ad ogni visita.

La frase di ricerca definita per tale query è Farmaci prescritti ai pazienti con malattia \$malattia.

Di seguito si riporta la query Cypher che consente di reperire le informazioni sopra indicate.

Query Cypher

```
match (m:Malattia {nome: '$malattia'}) -[r]-(v:Visita)-[p:PRESCRITTO]-> (f:Farmaco)
return *
```

Come si nota dalla query Cypher, si è scelto di considerare le malattie individuate sia in fase di anamnesi che in quella di diagnosi del paziente. È bene inoltre precisare che non è possibile concludere che la malattia ricercata sia curata con tutti i farmaci restituiti dalla query.

In Figura 5.19 è possibile osservare il risultato di una interrogazione del database utilizzando la frase di ricerca definita.

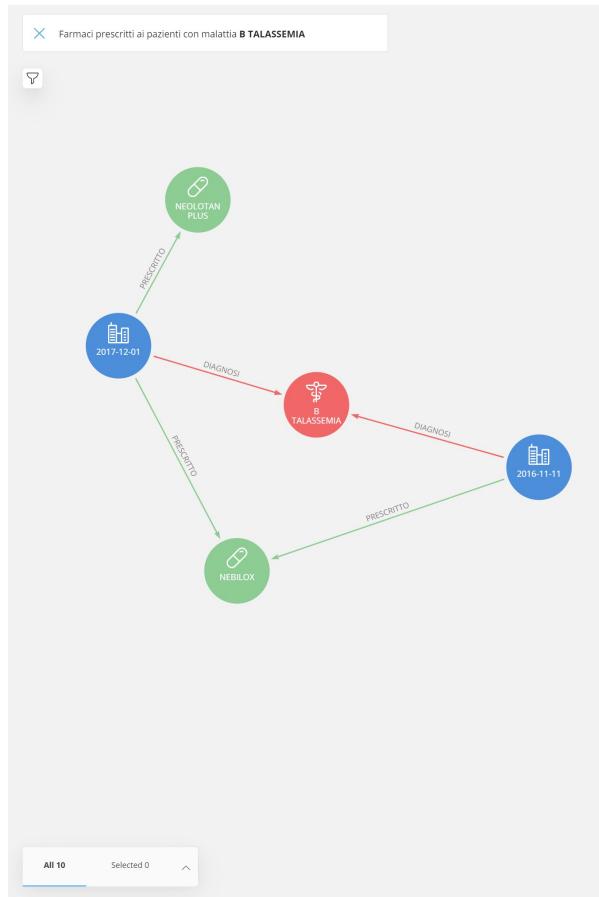


Figura 5.19: Farmaci prescritti ai pazienti con malattia B Talassemia

Bibliografia

- [1] Iannucci Antimo d'Andrea Fabio; Di Chiara Guido. *Repository GitHub - Cardiological Examinations Graph*. URL: <https://github.com/GuidoDC97/ElaboratoFinale-BDABI>.
- [2] Agenzia Italiana del Farmaco. *AIFA - Liste dei Farmaci*. URL: <https://www.aifa.gov.it/liste-dei-farmaci>.
- [3] Jing Li et al. “A Survey on Deep Learning for Named Entity Recognition”. In: *IEEE Transactions on Knowledge and Data Engineering* (2020), pp. 1–1. DOI: [10.1109/TKDE.2020.2981314](https://doi.org/10.1109/TKDE.2020.2981314).
- [4] Xuezhe Ma e Eduard Hovy. *End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF*. 2016.
- [5] Charlotte Skardon. *Repository GitHub - Connettore Neo4j-PowerBI*. URL: <https://github.com/cskardon/Neo4jDataConnectorForPowerBi>.