L'Intelligenza Artificiale per la Comorbidità

Salvatore Basilicata, Giovanni Casaburi

Email: s.basilicata@studenti.unisa.it, g.casaburi16@studenti.unisa.it

Università di Salerno, Dipartimento di Informatica

February 25, 2025

Contents

1 Introduzione

1.1 Problema

La comorbidità indica la situazione in cui un paziente è affetto da più malattie, le quali possono influenzarsi reciprocamente. Essa è al centro di varie ricerche e fornire degli strumenti adatti a comprenderne gli aspetti non solo porterebbe ad una maggiore preparazione da parte del corpo sanitario, ma potrebbe anche svelare quali sono le relazione che tendono a crearsi tra malattie diverse o identificare una sorta di pattern ricorrenti tra malattie.
Nei capitoli successivi ci soffemmeremmo sullo sviluppo di un sistema di Intelligenza Artificiale che, data la storia clinica del paziente, tenterà di predirre le malattie che saranno sviluppate nella storia futura del paziente.

1.2 Struttura del report

Il testo è suddiviso in diverse sezioni, ciascuna delle quali affronta un aspetto specifico del lavoro svolto:

- **Analisi di una soluzione esistente: MedMiner** – Viene presentata e discussa una soluzione preesistente nel contesto di riferimento, analizzandone punti di forza e limitazioni.
- **Progettazione e implementazione della nostra soluzione** – Descriviamo il nostro approccio, evidenziando le scelte metodologiche e tecniche adottate.
- **Conclusioni e sviluppi futuri** – Riassumiamo i risultati ottenuti e proponiamo possibili estensioni del lavoro.

2 MedMiner, la piattaforma contro la comorbidità

Una piattaforma che cerca di preparare ed aiutare i medici sull'analisi della comorbidità è **MedMiner**, una piattaforma web sviluppata da due nostri colleghi Gianfranco Barba e Tullio Mansi, per i dettagli implementativi e strutturali, lasciamo un riferimento alle loro tesi. In questa articolo, ci soffemmeremo come hanno creato la rete di comorbidità e sulla loro Intelligenza Artificiale per la previsione di una malattia della comorbidità. Di seguito è mostrato il link GitHub della repository : MedMiner

2.1 Grafo della comorbidità

La piattaforma MedMiner ha come scopo principale la creazione e visualizzazione del grafo della comorbidità di un paziente in particolare. In esso, il medico potrà visualizzare tutte le malattie che ha avuto il paziente e l'istante temporale in cui si sono verificate.

Per la creazione di tale grafo hanno utilizzato un dataset suddiviso in quattro macro-aree :

- **Sezione precrizioni** : in essa sono raccolte tutte le feature che riguardano le precrizione emesse dal medico riguardanti le diagnosi;
- **Sezione fragilità** : in essa sono presenti tutte le informazioni sulle condizioni fisiche e sociali dei pazienti per cui sono state emesse le precrizioni;
- **Sezione anagrafica pazienti** : sono contenute tutte le informazioni anagrafiche dei pazienti;
- **Sezione medici** : in questa sezione sono presenti le informazioni riguardanti il medico che ha effettuato la diagnosi.

Per la creazione del dato delle comorbidità sono state usate solo le sezioni relative al paziente alle prescrizioni, mentre la sezione medica è stata completamente ignorata insieme alla sezione fragilità, poichè risulta incompleta e con numerosi dati mancanti. Per le altre operazioni di data cleaning e feature selection, suggeriamo di leggere direttamente la loro tesi.
Una versione modificate e ridotta del dataset, formata da 100 mila righe è stata utilizzata per l'addestramento e la validazione di un modulo d'intelligenza artificiale, in particolare una Hetero Graph Neural Network (HGNN).

2.2 Modulo IA

I nostri colleghi hanno sviluppato una Hetero Graph Neural Network(HGNN) di due tipologie: SAGEConv e GatConv. La prima tipologia è quella più banale dove ogni nodo per costruire una rappresentazione grafica delle entità usa un'aggregazione dei nodi vicini. Il modello GatConv invece utilizza un ulteriore meccanismo, quello dell'attenzione in modo da assegnare dei pesi differenti ai nodi vicini in base a degli attributi che ne descrivono la rilevanza. Il nostro studio si è focalizzato sulla GatConv, perchè ci è sembrato il modello più adatto allo studio della comorbidità, poichè ogni malattia ha un impatto diverso nel paziente, inoltre tale meccanismo permette di considerare anche quelle malattie che fanno da hub(sono le malattie che principalmente collegano le patologie tra loro) nel grafo della comorbidità.

2.3 Revisione del lavoro

In questo studio abbiamo effettuato inizialmente una revisione completa del lavoro svolto dai nostri colleghi per capire quali fossero gli aspetti negativi della loro implementazione nonostante le performance del modello abbiano raggiunto un punteggio molto elevato (94% accuratezza). Tali risultati ci hanno interessato molto e abbiamo effettuato questa

revisione in primis per capire come funzionasse il modello, in secondo per capire se tali performance erano effettive. Partiamo dalla creazione del grafo della comorbidità.

2.3.1 Architettura del grafo di comorbidità

Per la creazione del grafo da passare in input al modello, hanno utilizzato una versione ridotta del dataset di partenza e selezionato solo le feature più rilevanti:

- Codice Fiscale paziente : il codice fiscale anonimizzato del paziente a cui è stata effettuata la diagnosi;
- Codice ICD9_CM : è il codice identificativo della malattia che è stata diagnosticata al paziente;
- Data Prescrizione : la data in cui è stata emessa la prescrizione;
- Anno di nascita : l'anno in cui è nata il paziente, utile per ricavarsi l'età di quando gli è stata diagnosticata la malattia;
- Sesso : il sesso del paziente;
- Descrizione malattia : una descrizione relativa alla diagnosi della malattia.

Il grafo che hanno creato presenta una struttura eterogenea perchè è formato sia da nodi che rappresentano i pazienti, sia da nodi che rappresentano malattie, inoltre hanno creato due tipologie di relazioni: una relazione che collega i pazienti alle malattie che gli sono state diagnosticate, ed una che collega un nodo paziente ad un altro nodo paziente che rappresenta l'impronta temporale.

```
# 5. Convertiamo le liste in tensori finali
data['paziente'].x = torch.stack(paziente_x)
data['malattia'].x = torch.stack(malattia_x)
data['paziente', 'diagnosi', 'malattia'].edge_index = torch.tensor(
    edge_index_paziente_malattia, dtype=torch.long).t().contiguous()
data['paziente', 'diagnosi', 'malattia'].edge_attr = torch.tensor(
    edge_attr_paziente_malattia, dtype=torch.float)
data['paziente', 'next', 'paziente'].edge_index = torch.tensor(
    edge_index_paziente_temporale, dtype=torch.long).t().contiguous()
data['paziente', 'next', 'paziente'].edge_attr = torch.tensor(
    edge_attr_paziente_temporale, dtype=torch.float)
```

Il paziente nel loro grafo rappresentava la prescrizione e non il paziente vero e proprio. Per ogni paziente unico nel dataset viene creato un sottografo per quel paziente dove vengono collegati i nodi pazienti con i nodi malattia, inoltre il sottografo sarà formato da più pazienti perchè rappresentano gli istanti temporali nel grafo. Ogni arco che collega un nodo paziente ad un nodo malattia ha un attributo che rappresenta il numero di volte che è stata diagnosticata quella malattia al paziente, mentre gli archi che collegano un nodo paziente ad unaltro hanno un attributo che ha il timestamp della data della prescrizione.

```
    # 4. Creiamo i sotto-grafi per paziente
for paziente in paziente_diagnosi_malattia['CODICE_FISCALE_ASSISTITO'].
    unique():
    paziente_data = paziente_diagnosi_malattia[
        paziente_diagnosi_malattia['CODICE_FISCALE_ASSISTITO'] ==
        paziente]
    paziente_data = paziente_data.sort_values(by='DATA_PRESCRIZIONE')

    malattie_contratte = {}  # Dizionario per tracciare le malattie e i
        pesi degli archi
    nodi_paziente_creati = []
    data_prescrizione_precedente = None

    for idx, row in paziente_data.iterrows():
        malattia_idx = row['ICD9_CM_CODE']
        data_prescrizione = row['DATA_PRESCRIZIONE']

        # Calcolo dell'et  alla data della prescrizione
        et   = data_prescrizione.year - row['ANNO_NASCITA']

        # Creazione nodo paziente con et  e sesso codificato
        paziente_feature = torch.tensor([et  , row['SESSO_CODE']], dtype
            =torch.float)
        paziente_x.append(paziente_feature)

        paziente_idx = paziente_id_counter
        paziente_id_counter += 1

        # Aggiunta nodo malattia solo se nuova malattia
        if malattia_idx not in malattie_contratte:
            malattia_feature = torch.tensor([row['ICD9_CM_CODE'], row['
                ETICHETTA_CODE']], dtype=torch.float)
            malattia_x.append(malattia_feature)
            malattie_contratte[malattia_idx] = 1  # Prima occorrenza
        else:
            # Incremento del peso dell'arco se la malattia    ricomparsa
            malattie_contratte[malattia_idx] += 1

        # Aggiungere un arco per la nuova malattia (o incrementare il
            peso)
        edge_index_paziente_malattia.append([paziente_idx, malattia_idx
            ])
        edge_attr_paziente_malattia.append(malattie_contratte[
            malattia_idx])

        # Aggiungere archi per le malattie contratte in precedenza
        for malattia_prec_idx in malattie_contratte:
            if malattia_prec_idx != malattia_idx:
```

```
                edge_index_paziente_malattia.append([paziente_idx,
                    malattia_prec_idx])
                edge_attr_paziente_malattia.append(malattie_contratte[
                    malattia_prec_idx])

        # Collegamento al nodo paziente precedente
        if nodi_paziente_creati:
            edge_index_paziente_temporale.append([nodi_paziente_creati
                [-1], paziente_idx])
            if data_prescrizione_precedente is not None:
                #differenza_temporale = (data_prescrizione -
                    data_prescrizione_precedente).days
                timestamp_prescrizione = data_prescrizione.timestamp()
                edge_attr_paziente_temporale.append(
                    timestamp_prescrizione)

        # Aggiorna nodi paziente creati e la data della prescrizione
        nodi_paziente_creati.append(paziente_idx)
        data_prescrizione_precedente = data_prescrizione
```

Una volta che hanno creato il grafo originale, hanno proseguito con l'aggiunta di archi negativi al grafo in modo tale che il modello impari a generalizzare. La quantità di archi negativi che viene generata è pari al 50% rispetto agli archi postivi. Tale generazione ha degli aspetti che potrebbero non rendere adeguato l'addestramento del modello. Innanzitutto, gli archi negativi a differenza di quelli positivi non hanno attributi, questo è un caratteristica critica poichè il modello potrebbe imparare che se leggendo quell'arco vede che non è presente l'attributo, allora vuol dire che esso è negativo e lo scarta subito, cosa che non può essere permessa. Un'altra considerazione che abbiamo fatto è che gli archi negativi vengono generati in maniera completamente casuale e non vengono distinti in hard-negative o normal negatives, gli hard negatives sono cruciali poichè essi rappresentano dei reali archi che si potrebbero creare tra il paziente ed una malattia.

```
# Creiamo le etichette per gli archi positivi (1 per ogni arco positivo)
positive_edge_label = torch.ones(positive_edge_index.size(1))

# Generiamo archi negativi per la relazione paziente-malattia
neg_edge_index = negative_sampling(positive_edge_index, num_nodes=(data[
    'paziente'].x.size(0), data['malattia'].x.size(0)), num_neg_samples=
    int(positive_edge_index.size(1) * 0.5))
```

2.3.2 Generazione dei train, validation e test set

Nella generazione dei vari set da passare al modello, hanno suddiviso il dataset di partenza in tre parti: test, validation e test set. Nel generare questi set, inizialmente effetuavano una permutazione random delle prescrizioni di un paziente (chiamati nodi paziente) ed

in base alla permutazione generavano i vari set. Questo approccio utilizzato ci è sembrato inefficiente poichè nella selezione delle istanze che andavano a far parte dei vari set si rischia di tagliare uno stesso paziente, rischiando innanzittuto di metterlo sia nel train set che nel validation set, ma rischiando anche di ridurre le informazioni che forniamo all'addestramento durante il modello. Sarebbe meglio addestrare il modello fino ad un'istante temporale di quel paziente per poi validarlo sul resto dell'istante temporale così riesce ad avere il quadro generale fino a quell'istante e a prevedere secondo esso in modo efficiente.

```
def train_test_split_edges(data, test_ratio=0.15, val_ratio=0.15):
pazienti = torch.unique(final_edge_index[0])
perm_pazienti = torch.randperm(pazienti.size(0))
pazienti = pazienti[perm_pazienti]
num_test = int(pazienti.size(0) * test_ratio)
num_val = int(pazienti.size(0) * val_ratio)
test_pazienti = pazienti[:num_test]
val_pazienti = pazienti[num_test:num_test + num_val]
train_pazienti = pazienti[num_test + num_val:]
```

2.3.3 Leaky Validation

Nella funzione di validazione per calcolare le metriche del modello abbiamo notato che il modello soffre di Leaky Validation. Per confermare questo basta considerare la riga riguardante la generazione del *z_dict*. Quando viene generato questo dizionario passiamo al modello l'intero dataset, con tutti gli archi, quindi il modello prima di effettuare la validazione, ha già visto tutti i nodi e tutti gli archi, quindi al momento della validazione sa già tutto e quindi si limita a leggere quello che viene passato come arco di validazione e controlla se nel grafo originale se tale arco è presente o meno. Quindi la causa principale delle alte performance del modello è dovuto a questo piccolo aspetto durante la fase di validazione.

```
def evaluate(model, data, val_edges, val_labels, edge_attr_dict,
    is_test=False):
model.eval()
with torch.no_grad():
    # Forward pass per il validation set o il test set
    z_dict = model(data.x_dict, data.edge_index_dict, edge_attr_dict
        =edge_attr_dict)
    val_pred = model.decode(z_dict, val_edges)

#Momento dell'invocazione
test_accuracy, test_auc, precision, f1, fpr, tpr = evaluate(model,
    data, test_edges, test_labels, edge_attr_dict, is_test=True)
```

3 La nostra implementazione

3.1 Divisione train e validation set

Idealmente, l'obiettivo sarebbe di riuscire a prevedere, per un certo paziente, quali sono le malattie che compariranno nella sua storia futura. La task di validazione è stata formulata in questo modo, ma quella di train si è basata su quest'idea:" Dato un paziente in un certo momento temporale t, denotato da tutte le malattie fino al momento t, prevedi le malattie che compariranno al tempo t+1", perché sequenzialmente si è voluto far apprendere la causalità in modo più informato. Quindi abbiamo diviso i pazienti in due gruppi: 70% per il train e 30% per la validazione. Poichè la task riguarda l'allenamento supervisionato, abbiamo suddiviso il train set in due insiemi: il primo insieme contiene, per ogni paziente e per ogni data_prescrizione unica (momento t) di quel paziente, una lista ordinata temporalmente di tutte le malattie fino al tempo t; per il train set, il secondo insieme contine,per ogni paziente e per ogni data_prescrizione, le malattie al tempo t+1, mentre il validation set contiene tutte le malattie della storia futura del paziente (>t+1). Questa suddivisione è giustificata dal fatto che, durante l'allenamento vogliamo acquisire tutte le informazioni sulla causalità, supponendo che le malattie che compariranno al prossimo istante siano influenzate dalle malattie precendenti. Invece durante la validazione vogliamo vedere come si comporta nel predirre tutte le malattie future che realmente avrà.

3.2 Architettura del modello

Il modello che abbiamo scelto si basa su due tecnologie principali: Gatv2Conv ed RNN. Il meccanismo di funzionamento è il seguente: dato il grafo totale di comorbidità tra malattie (dei pazienti di train) e un insieme di malattie di un paziente, dare in output un vettore di dimensione 1238 (numero di malattie), tale che ogni indice indica uno score assegnato alla malattia corrispondente a quell'indice (score alti saranno indicativi di un'alta probabilità che il paziente possa avere quelle malattie, grazie alla funzione di loss, che vedremo successivamente).

3.2.1 Grafo di Comorbidità

Per costruire il grafo di comorbidità abbiamo creato nodi e archi in questo modo: per ogni malattia abbiamo creato un nodo e tra due malattie abbiamo creato un arco se esite un paziente (ovviamente preso dall'insieme di train_set) in cui quelle malattie si presentano insieme. Ma ciò potrebbe non bastare per creare un grafo resiliente, infatti due malattie potrebbero comparire insieme in un solo paziente e non avere nessun nesso di causalità. Quindi abbiamo cercato un modo per far sì che le malattie che comparissero molto spesso insieme avessero un peso maggiore. Per questo abbiamo utilizzato l'indice di Jaccard

per l'attributo degli archi: il peso di ogni arco è calcolato come indicato dalla seguente formula dove:

- N_i : indica il numero di pazienti in cui la malattia i appare;
- N_j : indica il numero di pazienti in cui la malattia j appare;
- $N_{i,j}$: indica il numero di pazienti in cui le malattie i e j si presentano insieme;

$$P = \frac{N_{i,j}}{N_i + N_j - N_{i,j}} \tag{1}$$

3.2.2 Inizializzazione del Modello

```
class ComorbidityPredictor(nn.Module):
    def __init__(self, num_diseases, emb_dim, num_gat_layers=2, dropout
        =0.2):
        super().__init__()
        # Embedding malattie globali
        self.disease_emb = nn.Embedding(num_diseases, emb_dim)

        # GNN per il grafo comorbidit
        self.gnn_layers = nn.ModuleList()
        for _ in range(num_gat_layers):
            self.gnn_layers.append(GATv2Conv(emb_dim, emb_dim, edge_dim
                =1))
        self.dropout = nn.Dropout(dropout)

        # encoder temporale per lo storico del paziente
        self.rnn = nn.GRU(emb_dim, emb_dim, batch_first=True)

        # testa di predizione
        self.classifier = nn.Linear(emb_dim, num_diseases)
```

Il modello è inizializzato con 2 Gatv2Conv layer e dropout a 0.2, con 1238 embedding di malattie (ognuna di dimensione 128), un RNN con input di dimensione 128 (corrispondente ad una singola malattia) con hidden size sempre di 128 e un classificatore lineare (input di dimensione 128 e output di 1238, numero di malattie).

```
    def forward(self, patient_sequences, global_graph):
        #Patient_sequences    la prima met , global graph    la
            comorbidit  di tutte le prime met  dello storico dei
            paazienti
        # Step 1: aggiorna i disease embedding con il global graph
        x = self.disease_emb.weight
        for gnn_layer in self.gnn_layers:
```

```
            x = F.relu(gnn_layer(x, global_graph.edge_index,edge_attr=
                global_graph.edge_attr))
            x = self.dropout(x)
        global_embs = x

        # Step 2: codifica lo storico del paziente
        seq_embs = global_embs[patient_sequences]  # lookup per gli
            embedding delle malattie
        outputs, h_n = self.rnn(seq_embs)  # h_n: [1, batch_size,
            emb_dim]

        # Step 3: predici le malattie future
        logits = self.classifier(h_n.squeeze(0))
        return logits
```

Nel passo forward sono passati in input delle sequenze di malattie di pazienti fino ad arbitrari istanti e il grafo di comorbidità. (Si noti che se la batch fosse di dimensione 1, si avrebbe una sequenza di malattie di un paziente). Innanzitutto vengono aggiornati gli embedding delle malattie attraverso Gatv2Conv (utilizzando il grafo statico di comorbidità e i pesi degli archi). Per ogni paziente, tra gli embedding delle malattie ottenuti, quelli corrispondenti alla sequenza di malattie passate in input, sono a loro volta passati in input alla RNN secondo l'ordine temporale originale. L'hidden state finale ottenuto della RNN viene dato in input al classificatore e viene restituito l'output risultante: per ogni paziente del batch, un vettore con uno score per ogni malattia esistente nel dataset.

3.3 Training e Validazione

In questa parte verrà descritta la metodologia e le tecniche utilizzate per la fase di training e validazione, descrivendo anche la funzione di Loss utilizza e l'architettura del modello.

3.3.1 Funzione di Loss

Per il training abbiamo utilizzato la BPR loss, perchè, dato l'ouput del modello (che per ogni paziente presente nel batch e ogni malattia assegna un determinato score), idealmente vorremo che lo score attribuito alle malattie future del paziente sia superiore allo score di ogni malattia che non sia presente nello storico complessivo del paziente, e idealmente si avrebbe:

$$\mathcal{L}_{BPR}(u^*) = \frac{1}{|V_{pos}(u^*)| + |V_{neg}(u^*)|} \sum_{(u*,v_{pos}) \in V_{pos}(u^*)} \sum_{(u^*,v_{neg}) \in V_{neg}(u^*)} -\log\left(\sigma(\hat{x}_{u^*,v_{pos}} - \hat{x}_{u^*,v_{neg}})\right) \tag{2}$$

Dove:

- u^*: Indica il paziente;

- $V_{pos}(u^*)$: L'insieme delle coppie paziente, malattia $(u^*, v * pos)$ tali che il paziente ha realmente quella malattia
- $V_{neg}(u^*)$: L'insieme delle coppie paziente, malattia $(u^*, v * pos)$ tali che il paziente NON ha quella malattia
- $\hat{x}_{u^*i}$: Lo score predetto per la malattia V_{pos} e l'utente u^*.
- $\hat{x}_{u^*j}$: Lo score predetto per la malattia V_{neg} e l'utente u^*.

3.3.2 Creazione di malattie "negative"

Per la creazione di malattie negative abbiamo utilizzato un doppio approccio: generazione random e generazione degli hard negatives. Tale scelta viene stabilità da un valore booleano che empiricamente abbiamo settato a 1/10. Infatti generare sempre archi negativi difficili andava ad influire in modo negativo sulle prestazioni. Abbiamo pensato che questo accade perché un modello imparerà meglio se apprende da esempi di ogni possibile "dificoltà", quindi, per la maggior parte delle iterazioni di train abbiamo utilizzato negativi casuali. Per quanto riguarda le hard, bisognava trovare un modo per generare malattie negative che siano plausibili. Le relazioni di causalità che intercorre tra una malattia e l'altra non sono esplicite nei dati, ad esculisione delle malattie che condividono le cifre iniziali del codice ICD9_CM (ad esempio il diabete e le sue complicazioni). Però potrebbero esserci malattie appartenenti ad aree diverse che sono fortemente correlate.
Dunque, per poter generare delle malattie negative plausibili, utilizziamo il grafo di comorbidità del singolo paziente. Per ogni malattia prendiamo tutti i suoi nodi vicini (due malattie sono vicine se compaiono nello stesso paziente almeno una volta), i quali sono salvati in un dizionario(neighbor_dict), tutti i vicini di ogni nodo malattia sono potenziali candidati ad essere negative perchè hanno una forte correlazione con essa. Una volta presi tutti i vicini, eliminiamo quelli che sono presenti già nel target come malattie, poichè se esse sono presenti vuol dire che sono positive. Una volta scelti i candidati, ad ognuno di loro viene assegnato un probabilità di essere presi come negativi che è proporzianle alla correlazione che hanno con la malattia, più è alta quest'ultima, maggiore sarà la probabilità che venga scelta. Se alla fine di tutto questo processo non si trova un candidato valido, si procede alla generazione random dei negativi.

```
def sample_pos_neg_pairs(logits, targets, num_diseases, neighbor_dict,
   use_hard, batch_allowed_indices):

    if use_hard == False:
        batch_size = logits.size(0)
        pos_scores = []
        neg_scores = []

        for b in range(batch_size):
            pos_diseases = torch.nonzero(targets[b]).squeeze(-1)
```

```
            allowed_indices = batch_allowed_indices[b]

            for pos_disease in pos_diseases:
                pos_score = logits[b, pos_disease]


                num_neg_samples = 1
                if len(allowed_indices) == 0:
                    neg_diseases = torch.randint(0, num_diseases, (
                        num_neg_samples,))
                else:
                    rand_indices = torch.randint(0, len(allowed_indices)
                        , (num_neg_samples,))
                    neg_diseases = allowed_indices[rand_indices]

                neg_scores_batch = logits[b, neg_diseases]
                mean_neg_score = neg_scores_batch.mean()

                pos_scores.append(pos_score)
                neg_scores.append(mean_neg_score)
        return torch.stack(pos_scores), torch.stack(neg_scores)
    else:
        # Archi negativi difficili
        pass
        batch_size = logits.size(0)
        pos_scores = []
        neg_scores = []

        for b in range(batch_size):
            # Convert positive diseases to a list of integers for quick
                lookup
            pos_diseases_tensor = torch.nonzero(targets[b]).squeeze(-1)
            pos_diseases_list = pos_diseases_tensor.tolist()

            for pos_disease_tensor in pos_diseases_tensor:
                pos_disease = pos_disease_tensor.item()
                pos_score = logits[b, pos_disease]

                #ottieni i negativi difficili dal grafo di comorbidit
                candidates = neighbor_dict.get(pos_disease, [])
                # Filtra le malattie gi  presenti
                valid_candidates = [(n, w) for (n, w) in candidates if n
                    not in pos_diseases_list]

                if valid_candidates:
                    #estrai gli indici e i pesi
                    neg_indices, neg_weights = zip(*valid_candidates)
                    weights_tensor = torch.tensor(neg_weights, dtype=
```

```
                    torch.float)

                #normalizza i pesi in probabilit  e fai il sampling
                probs = weights_tensor / weights_tensor.sum()
                sampled_idx = torch.multinomial(probs, num_samples
                    =1).item()
                neg_disease = neg_indices[sampled_idx]
            else:
                # Fallback a random sampling se non ci sono
                    candidati validi
                neg_disease = torch.randint(0, num_diseases, (1,)).
                    item()
                while neg_disease in pos_diseases_list:
                    neg_disease = torch.randint(0, num_diseases,
                        (1,)).item()

            neg_score = logits[b, neg_disease]
            pos_scores.append(pos_score)
            neg_scores.append(neg_score)

    pos_scores = torch.stack(pos_scores) if pos_scores else torch.
        tensor([])
    neg_scores = torch.stack(neg_scores) if neg_scores else torch.
        tensor([])
    return pos_scores, neg_scores
```

3.3.3 Training

Per l'allenamento del modello è stato utilizzato l'optimizer AdamW ed è stato scelto, attraverso varie prove empiriche, di utilizzare 3 epoche con batch size di 100: per ogni epoca, permutiamo la lista di train (che contiene le varie liste di malattie per paziente e fino a un certo istante t)e iterativamente diamo in input al modello le varie batches che contengono le malattie di input corrispondenti a quel batch. L'output del modello, insieme ai labels, viene utilizzato per generare malattie negative e, infine, si usa la BPR loss con backpropagation per far si' che gli score di quelle malattie del vettore dato in output dal modello corrispondenti alle malattie positive (label) siano superiori a quelle delle malattie negative che non compaiono mai nella storia del paziente.

3.3.4 Validazione

Per la validazione è stata utilizzato un approccio che permettesse di misurare il raggiungimento dell'obiettivo: "Dato un paziente e un certo numero di malattie, si riesce a prevedere le malattie che il paziente avrà in futuro?". Per rispondere abbiamo utilizzato la precisione e recall cumulative. Cioè, per ogni paziente e data_prenotazione (che sia presente nello storico paziente), sono state passate in input le malattie fino a data_prenotazione. Dato il

vettore di output (ricoridamo che è un vettore di score di tutte le malattie), si sono presi i top k score, dove k è il numero di malattie che il paziente avrà da data_prenotazione in poi, e le malattie corrispondenti sono state confrontate con le k malattie che il paziente realmente avrà.

3.3.5 Risultati

Poichè in media le malattie future da predirre sono 41 e in totale ci sono 1238 malattie, un modello random avrebbe 3,3% di precision e recall, valori confermati anche da una simulazione di 100 assegnazoni casuali per il vettore degli score delle malattie e aver utilizzato le stesse metriche utilizzate nella validazione per valutare le performance. Il nostro modello riesce a raggiungere il 26$\pm$1% di precision e recall, avendo delle performance comunque superiori (8.7x) rispetto ad un modello random, confermando una parziale ma concreta correlazione tra le malattie passate e le malattie future di un paziente.