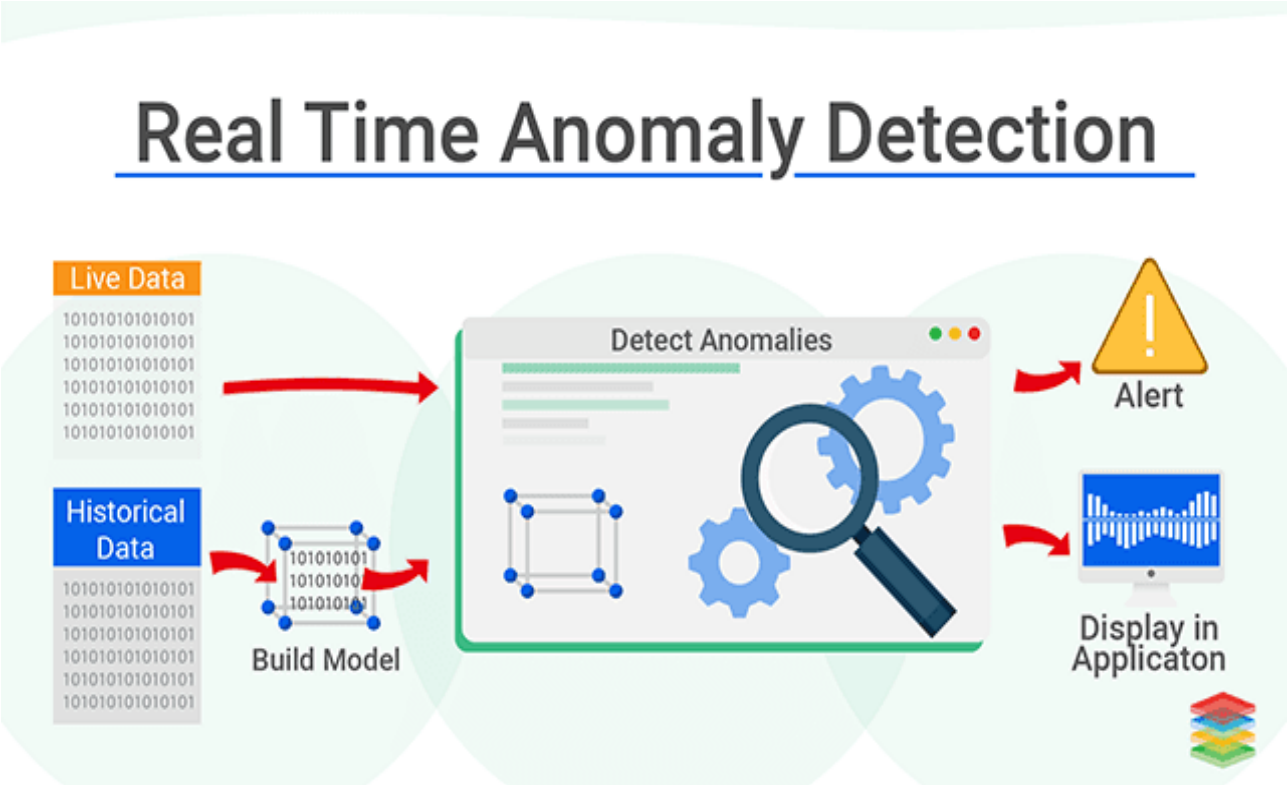


PROGETTO METODI INFORMATICI PER L'ANALISI DEI PROCESSI A.A: 2018/2019

"Analisi Degli Algoritmi Node2Vec e Graph2Vec"

Studenti

Emilio Casella matr.204898

Davide Costa matr.205167

Antonio Gagliostro matr.207059

Docente

Prof. Guzzo Antonella

Introduzione

Un algoritmo di apprendimento automatico è un algoritmo che è in grado di imparare dai dati forniti in ingresso. Una possibile definizione formale è la seguente:

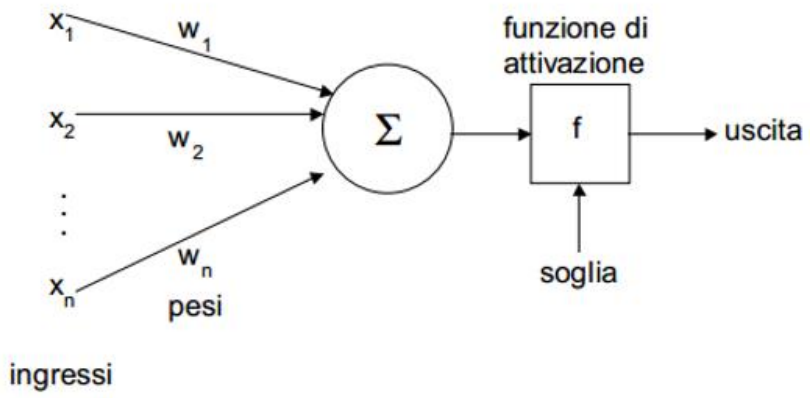
“Un algoritmo di apprendimento è in grado di imparare da una certa esperienza E rispetto a una classe di task T e misura di performance P, se la performance nei task in T, misurata secondo P, aumenta con l’esperienza E”.

Entrando più nel dettaglio, un task T sarà descritto in termini di come il sistema elabora un esempio (classificazione, classificazione con input mancanti, anomaly detection etc.); una performance P valuterà l’abilità dell’algoritmo in esame (specifica per il tipo di task); infine l’esperienza E sarà supervisionata se ad ogni esempio sul dataset di apprendimento è associata un etichetta, altrimenti sarà non visionata e si cercherà di riscostruire, tramite l’osservazione di variabili aleatorie, la distribuzione generatrice.

Vari algoritmi di machine learning adoperano un approccio di questo tipo, come la regressione lineare, lo Stochastic Gradient Descent etc.

Concentrandosi in particolar modo sulle reti neurali, si è passati da un iniziale obbiettivo di modellazione dei sistemi neurali biologi ad un ottimo strumento per tecniche di apprendimento avanzate.

Partendo dalla struttura biologica, si è arrivati a un modello computazionale del neurone, figura sottostante.



I segnali xi che viaggiano lungo gli assoni vengono combinati con i dendriti di altri neuroni in base al peso wi. L’idea alla base è che i pesi W dei dendriti possano essere addestrati e quindi utilizzati per controllare il comportamento dei neuroni associati. I contributi dei vari dendriti in ingresso vengono quindi sommati se il valore di tale somma supera una certa soglia, il neurone può attivarsi e generare quindi un segnale in uscita attraverso il suo assone.

Questa attivazione del neurone viene modellata tramite una funzione di attivazione f, che rappresenta la frequenza con cui viene generato un segnale in uscita: sono disponibili in letteratura una grande varietà di funzioni di attivazione che permettono di modellare i vari comportamenti che si vogliono associare al neurone.

In sintesi, il neurone “compie” un prodotto scalare tra input e pesi, “aggiunge” un valore di bias e applica la funzione di attivazione.

In neuroni sono divisi in N strati, N-1 strati nascosti (hidden layer).

Una rete di questo tipo, quindi, può essere rappresentata attraverso un grafo; dove gli output di certi neuroni diventano gli input di altri neuroni (non sono ammessi cicli).

L'idea alla base di un nuovo approccio è quella di mappare nodi o interi grafi (o sottografi) come punti in un vettore di piccole dimensioni, in modo tale da ottimizzare la rappresentazione del grafo originale e renderlo il più veritiero possibile.

Questa nuova struttura sarà data in pasto ad altre funzioni di apprendimento, utilizzando un approccio "data-driven" per incorporamenti che codificano la struttura di un grafo.

Embeddings sui grafi

Questa tecnica trasferisce le proprietà, del grafo in esame, ad un vettore o un insieme di vettori; consentendo di conservare info sulla topologia e altre info rilevanti su grafi e vertici (o anche sottografi).

Si può dividere l'embeddings in 2 gruppi:

- Vertex embeddings: si codifica ogni vertice (nodo) con la sua rappresentazione vettoriale; questo incorporamento viene usato quando si desidera eseguire la visualizzazione o la previsione a livello di vertice, ad es. visualizzazione di vertici sul piano 2D o la previsione di nuove connessioni basate su similitudini di vertici.

- Graph embeddings: si rappresenta l'intero grafo con un singolo vettore per fare previsioni a livello di grafo o quando si vuole confrontare o visualizzare tutti i grafi, ad es. confronto di strutture chimiche.

L'embeddings è necessario in quanto un semplice grafo, che utilizza uno specifico gruppo si argomenti matematici fisici o statistici, è limitato in questo contesto di utilizzo; al contrario dei vettori, che possono essere rimodellati con vari approcci e hanno un set di operazioni decisamente più veloci.

Un altro motivo è che i grafi hanno bisogno di una matrice di adiacenza per i propri nodi e le proprie connessioni che, specialmente per quelli con un grande numero di nodi, rappresenta un costo esponenziale in termini di memoria.

Al contrario, i vettori racchiudono le info di un gruppo di nodi, in un array di esigue dimensioni.

Un embeddings, per essere considerato un buon livello di rappresentazione, deve soddisfare i seguenti requisiti:

- Rappresentare necessariamente la topologia del grafo, i suoi nodi, le sue connessioni e i suoi vicini. Le performance di visualizzazione o previsione dipenderanno dal grado di accuratezza;

- La taglia del grafo non deve influenzare le performance del processo di embeddings;

- Bisogna trovare un buon compromesso tra la taglia del grafo e la complessità spaziale causata dalle innumerevoli informazioni da preservare.

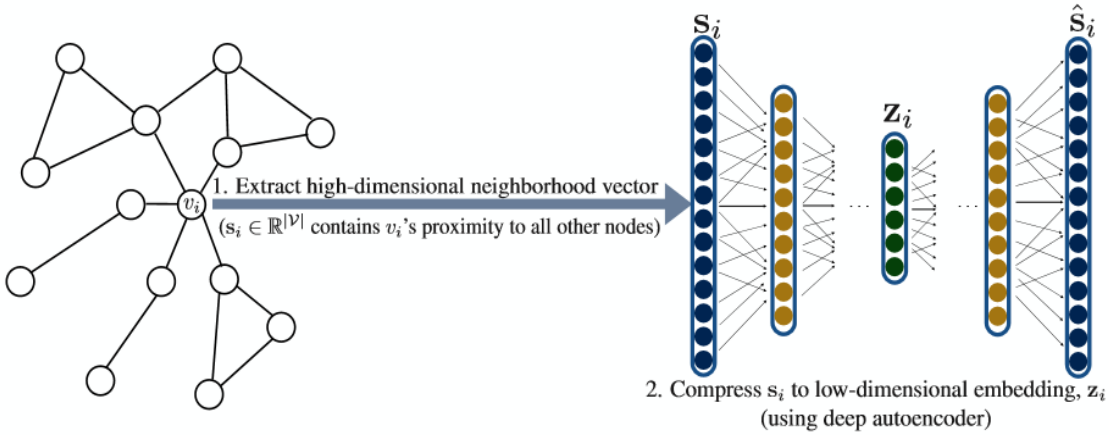


Figure 6: To generate an embedding for a node, v_i, the neighborhood autoencoder approaches first extract a high-dimensional neighborhood vector $\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$, which summarizes v_i's proximity to all other nodes in the graph. The $\mathbf{s}_i$ vector is then fed through a deep autoencoder to reduce its dimensionality, producing the low-dimensional $\mathbf{z}_i$ embedding.

Una rete che rappresenta al meglio queste caratteristiche è una rete di autoencoder.

Gli autoencoder sono reti neurali con lo scopo di generare nuovi dati dapprima comprimendo l'input in uno spazio di variabili latenti e, successivamente, ricostruendo l'output sulla base delle informazioni acquisite. Questa tipologia di network è composta da due parti:

- Encoder: la parte della rete che comprime l'input in uno spazio di variabili latenti e che può essere rappresentato dalla funzione di codifica h=f(x);
- Decoder: la parte che si occupa di ricostruire l'input sulla base delle informazioni precedentemente raccolte. È rappresentato dalla funzione di decodifica r=g(h).

L'autoencoder nel suo complesso può quindi essere descritto dalla funzione d(f(x)) = r dove r è quanto più simile all'input originale x.

Perché copiare l'input in output?

Quello che speriamo è che, allenando l'autoencoder a copiare l'input, lo spazio di variabili latenti h possa assumere delle caratteristiche a noi utili.

Questo può essere ottenuto imponendo dei limiti all'azione di codifica, costringendo lo spazio h a dimensioni minori di quelle di x. In questo caso l'autoencoder viene chiamato undercomplete.

Allenando lo spazio undercomplete, portiamo l'autoencoder a cogliere le caratteristiche più rilevanti dei dati di allenamento. Se non le diamo sufficienti vincoli, la rete si limita al compito di copiare l'input in output, senza estrapolare alcuna informazione utile sulla distribuzione dei dati.

Ciò può accadere anche quando la dimensione del sottospazio latente ha la stessa grandezza dello spazio di partenza.

Ad oggi, riduzione del rumore e riduzione della dimensionalità per la visualizzazione dei dati sono considerati le applicazioni più interessanti degli autoencoder.

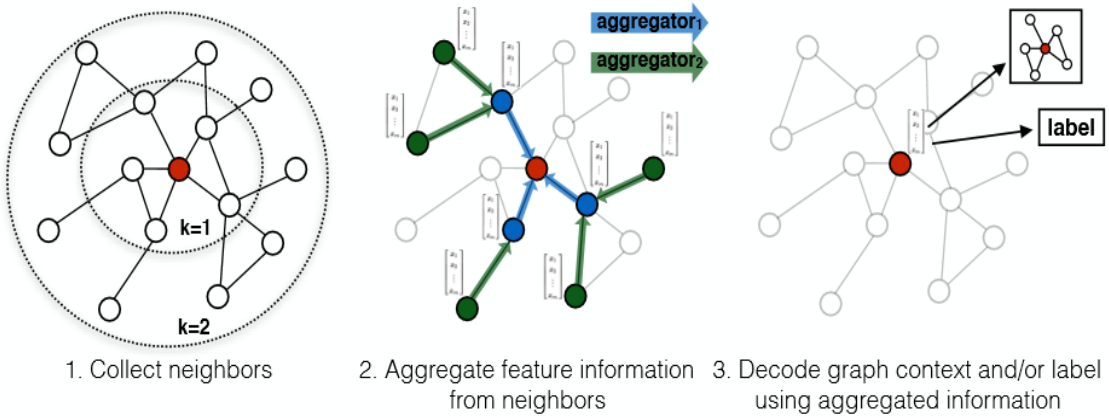
Con l'appropriato settaggio della dimensionalità e dei relativi vincoli sulla dispersione del dato, attraverso gli autoencoder si possono ottenere proiezioni in sottospazi di interesse maggiore rispetto a metodi lineari.

Gli autoencoder si allenano automaticamente attraverso dati di esempio, rendendo facile allenare la rete in modo da performare bene su simili tipologie di input, senza la necessità di generalizzare.

La compressione effettuata su dati simili a quelli utilizzati nel training set avrà dei buoni risultati, viceversa, su dati diversi sarà poco efficace.

Queste reti neurali sono allenate a preservare quante più informazioni possibili, quando sono inserite nell'encoder e successivamente nel decoder; ma devono anche far sì che le nuove rappresentazioni acquisiscano differenti tipi di proprietà.

Idealmente è possibile allenare con successo una qualsiasi architettura basata su autoencoder scegliendo opportunamente i parametri e la capacità di ciascun encoder-decoder in base alla complessità del dato da modellare.



1. Collect neighbors | 2. Aggregate feature information from neighbors | 3. Decode graph context and/or label using aggregated information

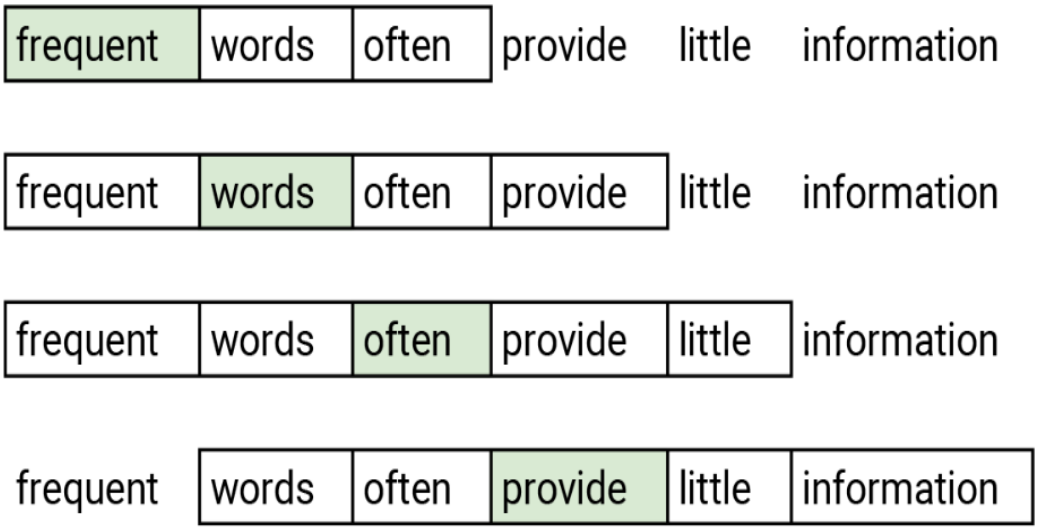
Word2vec

Word2vec consiste in una traduzione di parole o frasi in vettori composti da numeri reali.
Questi modelli non sono altro che delle reti neurali a due livelli che sono addestrate (tramite un approccio non supervisionato) per ricostruire i contesti linguistici delle parole. Word2Vec prende come input un grande frammento di testo e costruisce uno spazio vettoriale, tipicamente di diverse centinaia di dimensioni, in cui ogni parola è univocamente assegnata ad un corrispondente vettore nello spazio, seguendo un certo criterio. Infatti i vettori (rappresentanti le parole) vengono posizionati nello spazio cosicché le parole che risultino "simili" all'interno del frammento di testo, siano collocate vicine tra loro nello spazio stesso.
L'idea è quella di riuscire ad incapsulare relazioni differenti tra le parole, come ad esempio sinonimi, contrari o analogie. Word2Vec fa uso di un *trucco* tipicamente usato nel machine learning: addestrare una semplice rete neurale - con un singolo livello di nodi di hidden - ad effettuare un certo task, che però poi non sarà quello per cui verrà effettivamente usata la rete neurale. Un esempio noto è quello che riguarda la compressione di un vettore: si fa uso, dell'apprendimento non supervisionato per addestrare la rete neurale a comprimere (nel livello di hidden), un vettore dato come input, successivamente decompresso nuovamente (nel livello di ouput) riottenendo l'originale. Dopo l'addestramento, il livello di output (ovvero lo step di decompressione) può essere scartato mantenendo solo i livelli di input e di hidden.

Esistono due metodologie per poter creare il modello:

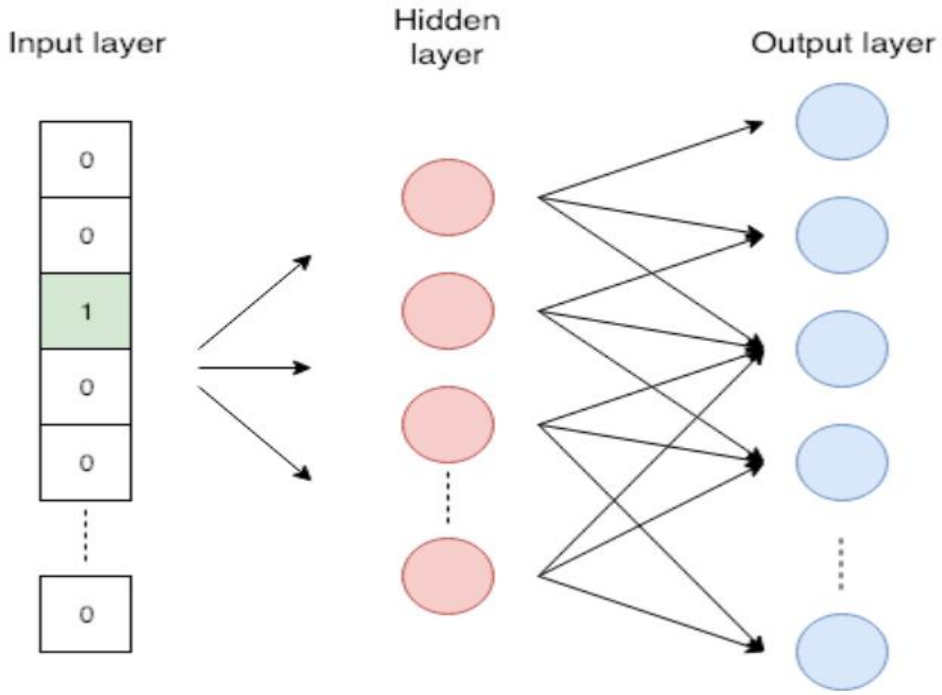
• **Skip-Gram model:** in cui si prendono coppie di parole dal testo e si addestra una rete neurale con un livello di nodi di hidden sulla base del finto task in cui, a partire da una parola in input, la rete restituisce la distribuzione di probabilità delle parole vicine (all'interno del testo) all'input. In altre parole, la rete restituirà alte probabilità per le parole che tipicamente compaiono "vicine" alla parola in input. Essendo un finto task però, ad essere realmente importanti saranno i pesi tra i nodi di input e i nodi di hidden che saranno usati come word embedding. Quindi se lo strato di hidden possiede 300 neuroni, la rete restituirà un vettore di dimensione pari a 300, per ciascuna parola.

• **A continuous bag of words (CBOW):** anch'essa fa uso di una rete neurale con un livello di nodi di hidden. Il finto task in questo caso è basato sul predire una parola centrale, a partire da un insieme di parole di input (appartenenti allo stesso contesto); saranno poi i pesi tra il livello di input e di hidden ad essere usati come word embeddings per i termini dati in ingresso alla rete.

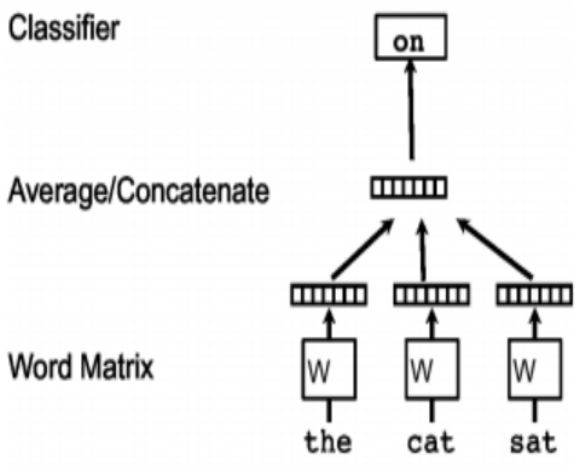


The word colored with green is given to the network. It is optimized to predict the word in the neighborhood with higher probability. In this example, we consider words that are the most two places away from the selected words.

La rete neurale skip-gram vista nella figura seguente ha un livello di input, un livello nascosto e un livello di output; accetta le parole con una codifica one-hot ovvero un vettore con lunghezza uguale a quella del dizionario delle parole e con tutti zeri tranne un elemento posto ad uno (punto in cui appare una parola codificata nel dizionario). Il livello nascosto non ha alcuna funzione di attivazione, il suo output presenta un incorporamento della parola. Il livello di output è un classificatore softmax che predice le parole di vicinato.



Skip-gram neural network



Node2Vec

L'algoritmo node2vec appartiene al gruppo di algoritmi vertex embeddings.

La sua base è caratterizzata da un approccio di tipo DeepWalk, ovvero un'applicazione del classico random walk alle reti di apprendimento.

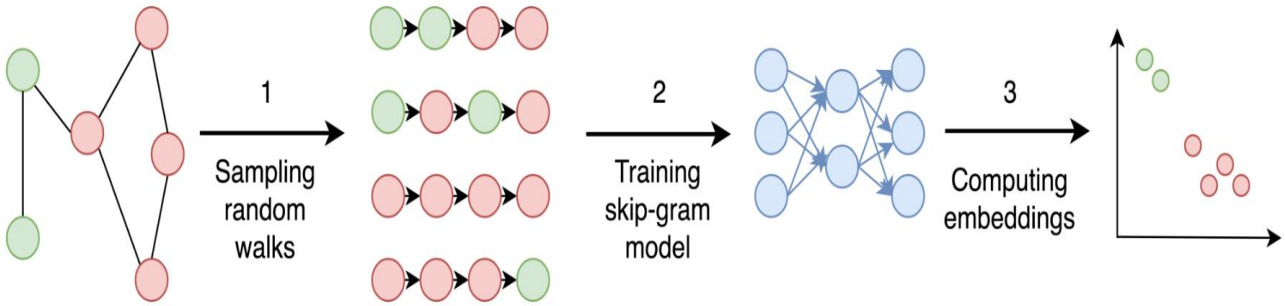
Per prima cosa ci si muove selezionando un nodo e ci si sposta su un vicino casuale dal nodo selezionato, in un determinato numero di passi.

Il metodo consiste sostanzialmente in tre passaggi:

Campionamento: un grafico viene campionato con passeggiate casuali. Vengono eseguiti pochi passi casuali da ciascun nodo. I buoni cammini random compiono circa 30 passi.

Allenamento skip-gram: le passeggiate casuali sono paragonabili alle frasi nell'approccio word2vec. La rete skip-gram prende in input un nodo del random walk come un vettore one-hot e massimizza la probabilità di prevedere i nodi vicini. È in genere addestrato per prevedere circa 20 nodi vicini - 10 nodi a sinistra e 10 nodi a destra.

Computing embeddings: E' l'output di un livello nascosto della rete; DeepWalk lo calcola per ciascun nodo nel grafo.



Phases of DeepWalk approach

Il problema principale del DeepWalk, però, risiede nel fatto di non riuscire a preservare una buona "memoria" dei nodi vicini a causa del suo modo di operare.

Risulta, quindi, un algoritmo molto flessibile che garantisce un buon controllo sullo spazio di ricerca in esame tramite l'utilizzo di semplici parametri che orientano le strategie del cammino nella rete.

Gli algoritmi di ricerca più utilizzati per esplorare i nodi vicini sono il Breadth-first (il vicinato è ristretto ai nodi più vicini a quello in esame) e il Depth-first (esplora i nodi per distanze incrementali rispetto al nodo in esame).

Applicando un approccio basato sul random walk a quest'ultimi, si riescono ad ottenere buoni vantaggi in termini computazionali e spaziali.

3.2.3 *The node2vec algorithm*

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
  π = PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  for $iter = 1$ **to** r **do**
    for all nodes $u \in V$ **do**
      $walk$ = node2vecWalk(G', u, l)
      Append $walk$ to $walks$
  f = StochasticGradientDescent(k, d, $walks$)
  return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
  Inititalize $walk$ to $[u]$
  for $walk\_iter = 1$ **to** l **do**
    $curr = walk[-1]$
    V_{curr} = GetNeighbors($curr, G'$)
    s = AliasSample(V_{curr}, π)
    Append s to $walk$
  return $walk$

Ricapitolando, Node2vec è una modifica di DeepWalk con una piccola differenza nelle “passeggiate casuali”. Ha i parametri dei parametri che chiamiamo, ad esempio, P e Q.

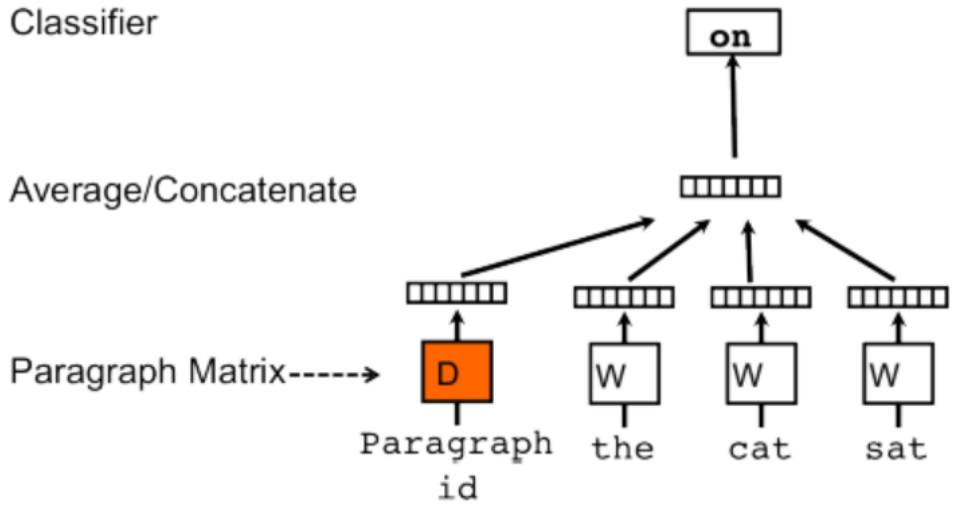
Il parametro Q definisce quanto è probabile che la “camminata casuale” rilevi la parte sconosciuta del grafo, mentre il parametro P definisce quanto è probabile che la camminata casuale ritorni al nodo precedente. Il parametro P controlla in maniera microscopica i nodi attorno a quello in esame, mentre Q controlla in maniera macroscopica il vicinato.

Doc2Vec

In maniera analoga agli scopi del Word2Vec, l’obiettivo del Doc2Vec è di creare una rappresentazione numerica di un intero documento (o paragrafo, nel nostro caso di un tweet) a prescindere dalla sua lunghezza. Il principio usato è semplice ed intelligente:
si fa uso del modello word2vec e si aggiunge un altro vettore, detto **Paragraph ID**. Quindi dopo aver addestrato la rete neurale, si avranno oltre che i word vectors (la rappresentazione vettoriale delle parole) anche un document vector (la rappresentazione vettoriale del documento).
Anche nel caso del Doc2Vec esistono due metodologie analoghe rispettivamente al
CBOW e lo Skip-Grim:

• **Distributed Memory version of Paragraph Vector (PV-DM):** che agisce come una memoria che ricorda cosa manca esattamente all'interno del contesto in questione o come l'argomento del paragrafo. Mentre i word vectors rappresentano il concetto di una parola, il document vector intende rappresentare il concetto di un documento.

• **Distributed Bag of Words version of Paragraph Vector (PV-DBOW):**
che risulta essere più veloce e che consuma meno memoria (contrariamente a quanto accade allo skip-gram nel word2vec) in quanto non c'è alcun bisogno di salvarsi i word vectors (rappresentazioni vettoriali delle parole). Dopo l'addestramento infatti, basta fornire il paragraph ID (detto anche tag) per ricevere il document vector.



Graph2vec

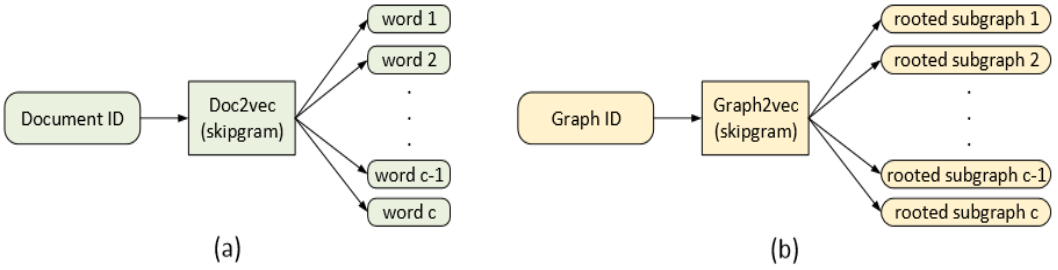
Graph2vec si basa su doc2vec che utilizza una rete di skip-gram. Ottiene un ID del doc sull'input ed è addestrato per massimizzare la probabilità di prevedere parole casuali tramite il doc stesso.

Gli approcci Graph2vec consistono in tre fasi:

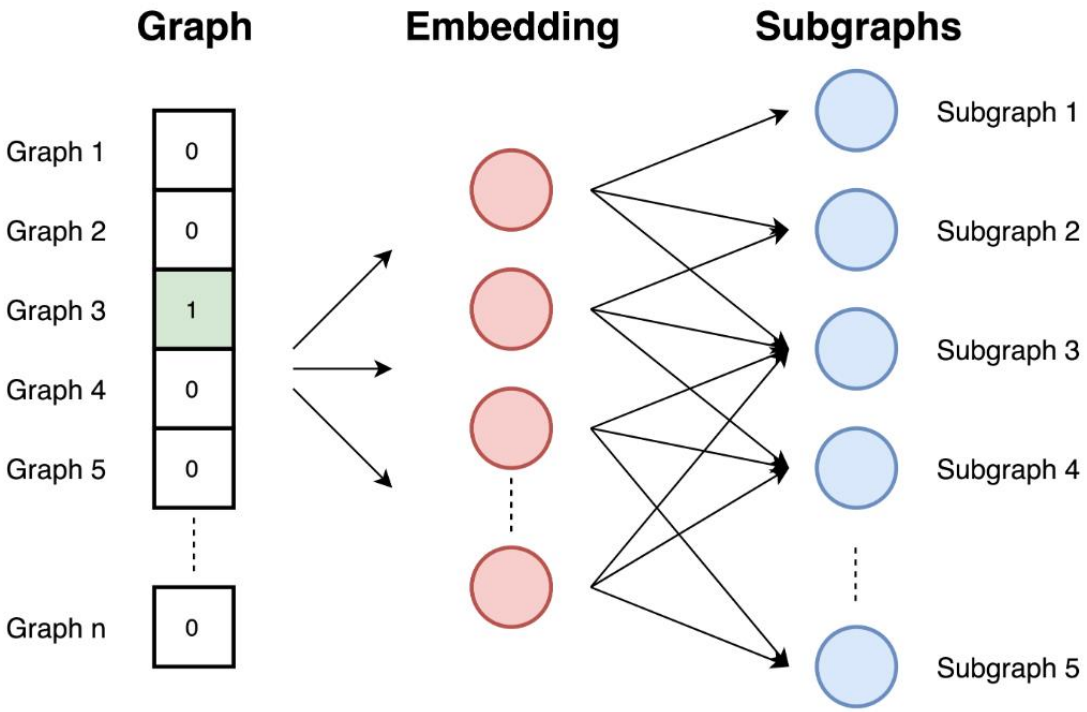
- Campionamento e ri-etichettatura di tutti i sotto-grafi dal grafo: il sotto-grafo è un insieme di nodi che appaiono attorno al nodo selezionato. I nodi nel sotto-grafo non sono superiori al numero selezionato di passi.

- Allenare il modello skip-gram: I grafi sono simili ai doc. Poiché i doc sono un insieme di word, i grafi sono un insieme di sotto-grafi. In questa fase viene addestrato il modello skip-gram e deve massimizzare la probabilità di prevedere un sotto-grafo esistente nel grafo in input. Il grafo di input viene fornito come un vettore one-hot.

- Calcolo di embeddings: Si fornisce un ID tramite un vettore one-hot in input. L'embeddings consiste nel risultato prodotto dal livello nascosto.



Poiché l'attività prevede previsioni di sotto-grafi, i grafi con sotto-grafi simili e struttura simile hanno incorporamenti simili.



Algorithm 1: GRAPH2VEC $(\mathbb{G}, D, \delta, \mathfrak{e}, \alpha)$

input : $\mathbb{G} = \{G_1, G_2, ..., G_n\}$: Set of graphs such that each graph $G_i = (N_i, E_i, \lambda_i)$ for which embeddings have to be learnt
D: Maximum degree of rooted subgraphs to be considered for learning embeddings. This will produce a vocabulary of subgraphs, $SG_{vocab} = \{sg_1, sg_2, ...\}$ from all the graphs in $\mathbb{G}$
δ: number of dimensions (embedding size)
$\mathfrak{e}$: number of epochs
α: Learning rate

output: Matrix of vector representations of graphs $\Phi \in \mathbb{R}^{|\mathbb{G}| \times \delta}$

```
1  begin
2      Initialization: Sample Φ from R^{|G| × δ}
3      for e = 1 to 𝔢 do
4          𝔊 = SHUFFLE (𝔾)
5          for each G_i ∈ 𝔊 do
6              for each n ∈ N_i do
7                  for d = 0 to D do
8                      sg_n^(d) := GETWLSUBGRAPH(n, G_i, d)
9                      J(Φ) = −log Pr (sg_n^(d) | Φ(G))
10                     Φ = Φ − α ∂J/∂Φ
11     return Φ
```

Business Process e Machine learning

Rilevare anomalie nei propri processi interni è un compito oneroso per ogni azienda ma necessario, poiché potrebbe essere indicatore di frodi e inefficienze varie. All'interno del dominio della business intelligence, la rilevazione di anomalie classiche non viene ricercata molto frequentemente.

Una delle principali sfide nel process mining è la gestione dell'elevata dimensionalità dei dati.

Un approccio alternativo viene sviluppato tramite gli autoencoders, non basandoli su conoscenze a priori e compiendo operazioni di addestramento su set di dati contenenti anomalie.

Le tecniche di process mining classiche forniscono metodologie di rilevazione di anomalie nell'esecuzione di un processo che richiedono l'esistenza di un modello di riferimento.

Tramite l'utilizzo di tecniche di machine learning, se non è disponibile quest'ultimo, si va alla scoperta di un modello di riferimento dal registro eventi stesso; utilizzando una soglia per gestire comportamenti rari, in modo che il modello rilevato sia una buona rappresentazione del normale comportamento del processo. Pertanto, questi può essere utilizzato come modello di riferimento per il controllo di conformità.

Un presupposto chiave nel rilevamento di anomalie è che le esecuzioni anomale si verificano meno frequentemente delle esecuzioni normali. Questa distribuzione “distorta” può essere sfruttata quando si applicano le tecniche di rilevamento delle anomalie.

Questa metodologia funziona secondo i seguenti principi:

- Nessuna conoscenza preliminare del processo
- I dati di addestramento contengono già anomalie
- Nessun modello di riferimento necessario
- Nessuna etichetta necessaria (cioè nessuna conoscenza di anomalie)
- L'algoritmo deve rilevare l'attività esatta in cui si è verificata l'anomalia

Stato dell’arte per architetture di process mining

Vediamo ora come applicare tutti i concetti visti precedentemente alla creazione di architetture adibite alla rappresentazione di event logs e process models.

Act2vec

Act2vec è una architettura di apprendimento utilizzata per la rappresentazione delle attività.

Supponiamo di avere i dati in input sotto forma di un registro ad eventi. In linea con l'approccio Word2vec nell'elaborazione del linguaggio naturale, possiamo rappresentarle considerando le attività come word in un corpus, con il corpus che è il registro eventi nel nostro caso.

L'architettura si basa sul principio secondo cui una parola può essere prevista dal suo contesto (cioè le parole che appaiono prima e dopo la parola chiave

Le tracce saranno quindi considerate come frasi ed eventi come parole. Pertanto, l’algoritmo apprende rappresentazioni di attività generiche che non sono adattate, ad esempio, alla previsione dell'attività successiva o alla previsione del tempo rimanente delle istanze.

Si noti che, nonostante le tracce in un registro eventi siano di natura sequenziale, si sceglie di definire il contesto di un'attività in base all'insieme non ordinato di attività precedenti e successive; ottenendo un'architettura di apprendimento che può essere utilizzata per comprendere il contesto di due diverse attività. A seconda del loro contesto, la loro rappresentazione sarà simile o molto diversa. Mentre le rappresentazioni delle attività potrebbero essere utilizzate per particolari attività relative al BPM.

Per accelerare il processo di addestramento della rete neurale, in quanto, in ogni aggiornamento della rete, solo una piccola percentuale dei suoi parametri vengono aggiornati, modificando i pesi in base al

campionamento del vettore di output (un vettore one-hot).

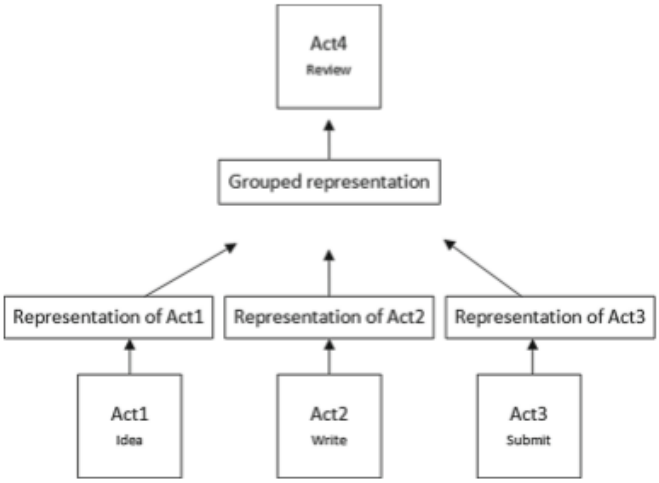


Fig. 1. The *act2vec*-architecture for learning vector representations of activities. The context consisting of activities "idea", "write", and "submit" is used to predict activity "review".

L'architettura Act2vec può essere estesa in diversi modi, utilizzando altri attributi correlati che potrebbero includere altri dati disponibili nel registro eventi, nella definizione del contesto di un'attività o nel concetto da prevedere. Queste dimensioni aggiuntive dei dati possono essere considerate come input aggiuntivi dell'architettura proposta sopra, rendendo possibile esplorare progetti gerarchici o multistrato più complessi.

I modelli a cui si ispira questa tipologia di architettura presuppongono che gli input contestuali siano indipendenti (in genere, il raggruppamento delle rappresentazioni degli input contestuali viene eseguito mediante la media o la somma di questi, sebbene anche qui sia possibile utilizzare un approccio di concatenazione, che risolve parzialmente il presupposto di indipendenza). Nell'analisi del processo, sembra logico presumere che l'ordine sia importante e, quindi, da tenere in considerazione.

Nonostante questo, è stato dimostrato che architetture di incorporamento della rete neurale che trascurano l'ordine, come gli autoencoder, forniscono risultati solidi utilizzando l'iperparametrizzazione predefinita.

Trace2vec

Seguendo l'analogia tra attività e word, anche le tracce possono essere considerate frasi. L'apprendimento di rappresentazioni distribuite di frasi, paragrafi o documenti è stato introdotto nel dominio dell'elaborazione del linguaggio naturale sotto forma dell'approccio doc2vec, come spiegato precedentemente.

Questa idea può essere adottata anche per le tracce, dando origine all'architettura Trace2vec.

Dato che questa architettura include una rappresentazione di tracce (basata sull'identificatore di tracce), consentirà l'apprendimento congiunto di rappresentazioni di attività e tracce (alle quali siamo principalmente interessati).

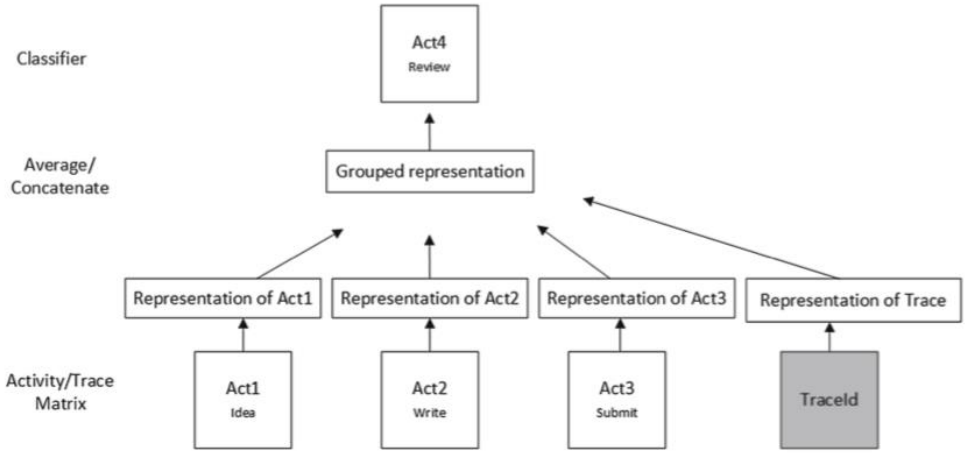


Fig. 2. The *trace2vec*-architecture for learning vector representations of traces. The context consisting of activities "idea", "write", and "submit", as well as the trace-id, is used to predict activity "review".

Qui, l'idea è di usare un solo paragraph vector in input, per prevedere una piccola finestra di parole nel paragrafo sul lato di output.

Si noti che, oltre ad utilizzare queste architetture basate su doc2vec, sarebbe anche possibile derivare rappresentazioni a livello di traccia facendo uso di architetture di aggregatori. Una tale architettura aggregatrice aggregato le rappresentazioni di attività.

Log2vec

Diverse attività di apprendimento basate sui dati all'interno del dominio BPM si basano sulle rappresentazioni di un intero processo (rappresentato da un registro eventi o da un modello). Come tale, in linea con Trace2vec, un progetto architetturale può essere concepito per apprendere rappresentazioni distribuite dei registri. Un metodo semplice potrebbe essere quello di sostituire la rappresentazione di una traccia con una rappresentazione di un registro.

Dato che un'architettura di questo tipo, non sarebbe in grado di incorporare informazioni a livello di traccia e quindi considererà un event log come un insieme di attività non raggruppate; il che ha senso dal punto di vista dei processi aziendali per includere anche le informazioni di traccia. Per fare ciò, potremmo semplicemente utilizzare l'identificatore delle tracce come prima, tuttavia, dato che i processi aziendali potrebbero condividere varianti di esecuzione simili, proponiamo di includere un identificatore artificiale relativo a istanze di processo distinte nell'architettura (ovvero un identificatore di "variante" di traccia).

Più in particolare, tutte le tracce dei diversi event log in esame vengono unite in un registro ad eventi basato su quell'input, permettendo di calcolare un identificatore di istanza del processo distinto.

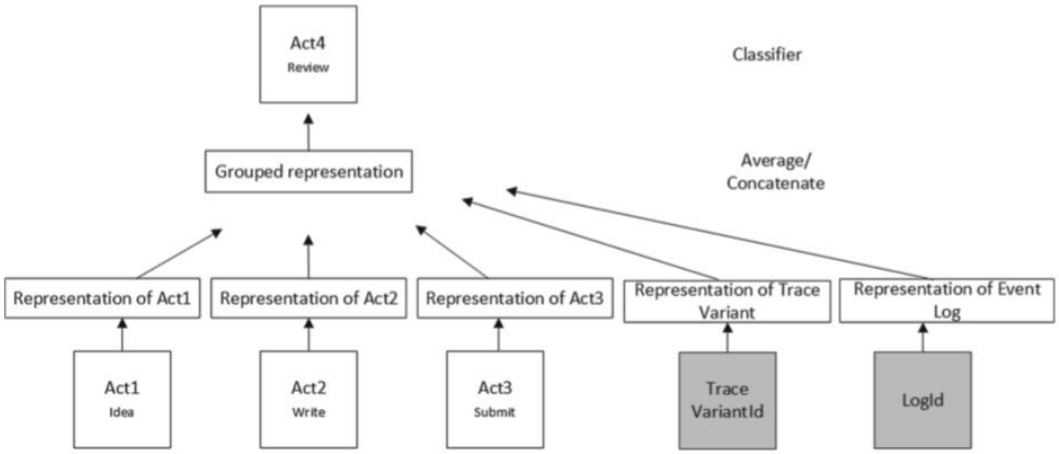


Fig. 3. The *log2vec*-architecture for learning vector representations of logs. The context consisting of activities "idea", "write", and "submit", an identifier for each trace variant, as well as the log-id from which the words are sampled, is used to predict activity "review".

Model2vec

L'obbiettivo è quello di rappresentare "process model" come vettori di piccole dimensioni.

Basandosi sulle architetture sopra discusse, un'estensione banale potrebbe essere quella di ottenere una rappresentazione generica simulando prima i modelli (tenendo traccia dei dati, degli eventi prodotti durante la simulazione) e applicando successivamente Log2vec, per apprendere una rappresentazione del modello di processo.

Tuttavia, dato che si può sostenere che i modelli non sono rappresentati esclusivamente dalle varianti di esecuzione, che sono in grado di produrre, si aprono opportunità per applicare diverse tecniche di apprendimento, utili alla rappresentazione di una struttura generale.

Si utilizza un approccio random walk, come illustrato in precedenza, prendendo in considerazione solo indirettamente il comportamento effettivo del modello, ma ponendo maggiormente l'accento sul layout grafico e le relazioni tra gli elementi di modellazione, compresi quelli che non sono direttamente correlati alla rappresentazione di un'attività (come gateway, luoghi o altri costrutti).

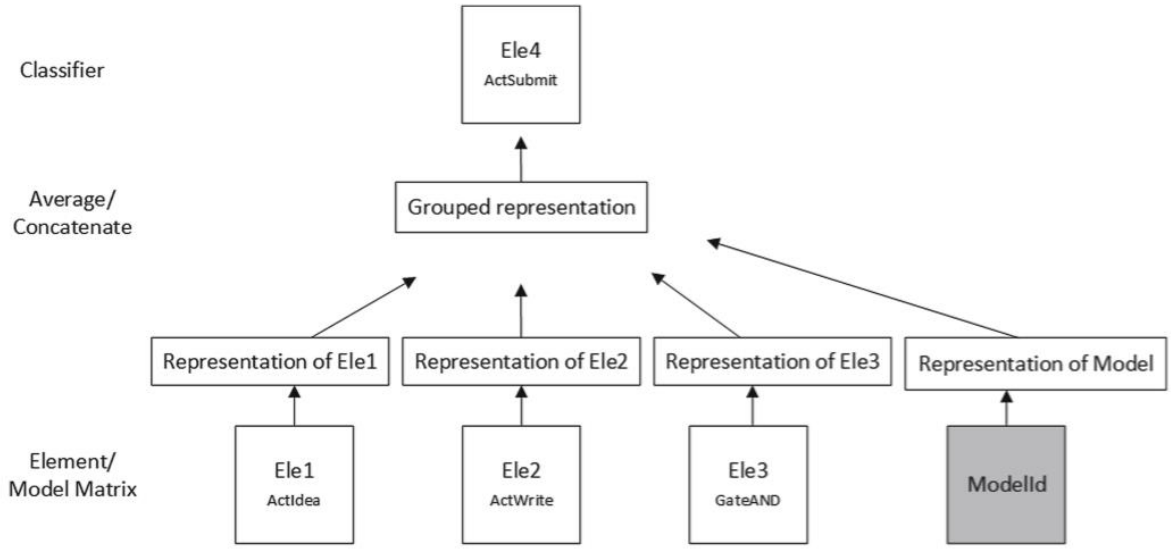


Fig. 4. The *model2vec*-architecture for learning vector representations of logs. The context of elements resulting from either simulation or random walk ("idea", "write", "And-gateway"), as well as the model-id for the model on which the simulation is performed or the walk is sampled, are used to predict activity "submit".

Model2vec (e anche log2vec quando combinato con la simulazione) consente un approccio alternativo al processo classico di confronto dei modelli.

Il confronto tra modelli basati su Model2vec potrebbe sfruttare la dimensione strutturale e grafica dei modelli stessi, con approcci esistenti fortemente incentrati sulla somiglianza comportamentale. Nel caso in cui questa dimensione strutturale / grafica dovesse essere de-enfatizzata, prevediamo che Log2vec o una combinazione di entrambi, potrebbe offrire l'opportunità di affrontare il confronto tra modelli in modo diverso.

Il log sotto modello XES

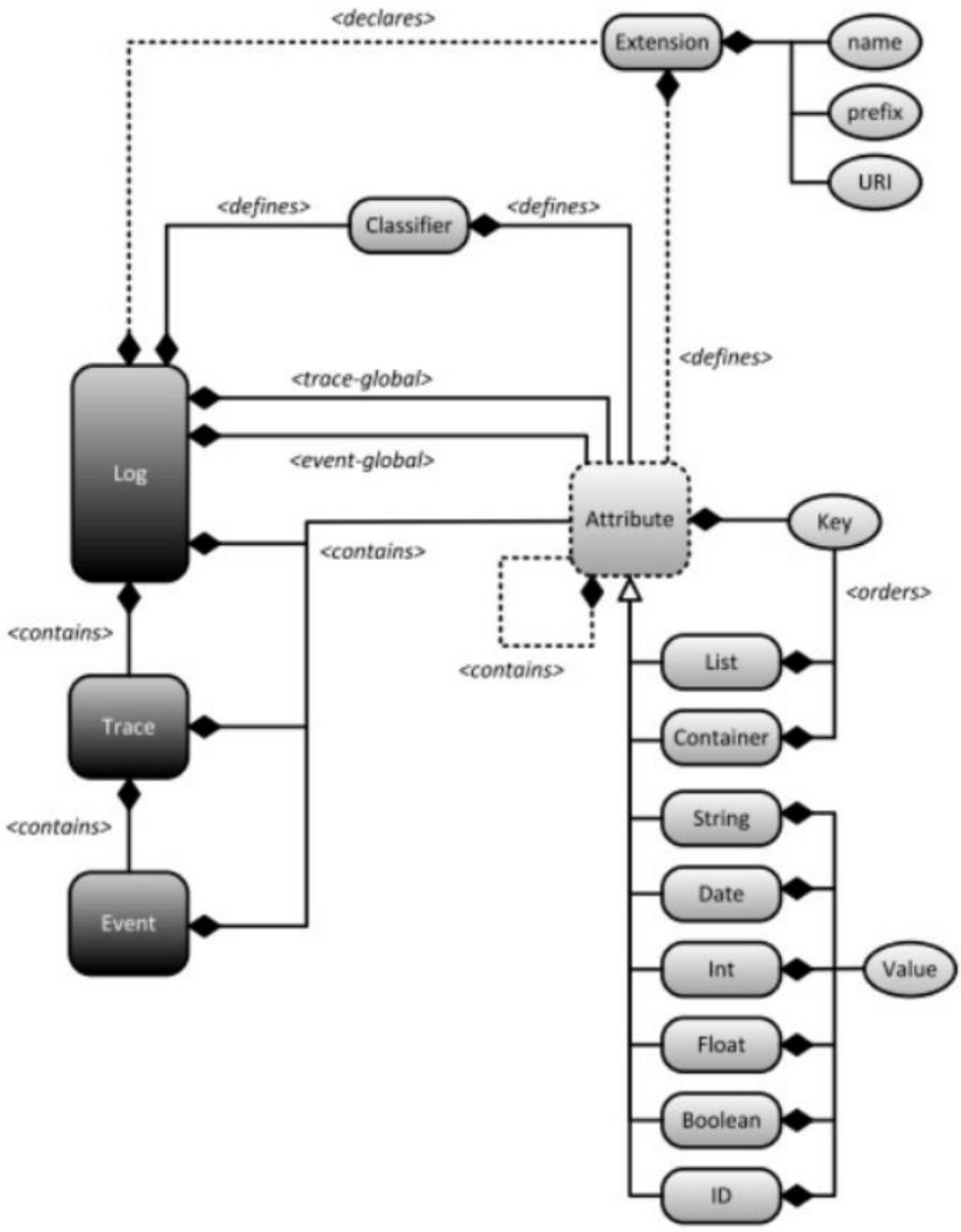
E' bene fare una piccola analisi, prima di passare ad altro, del log che sarà preso in esame.

La gerarchia di base di un documento XES segue la struttura universale delle informazioni del registro degli eventi:

- Log: al livello superiore è presente un oggetto di log, che contiene tutte le informazioni sugli eventi che sono legati ad un processo specifico. Il nome del tag per l'oggetto log nella serializzazione XML di XES è:<log>. Attributi del tag <log> XML sono xes.version e xes.features;

- Trace: un registro contiene un numero arbitrario (può essere vuoto) di oggetti di traccia. Ogni traccia descrive l'esecuzione di un caso specifico, o il caso, del processo registrato. Il nome del tag per l'oggetto trace nella serializzazione XML di XES è: <trace>. Non sono definiti attributi XML per il tag <trace>;

- Event: ogni traccia contiene un numero arbitrario di oggetti evento. Gli eventi rappresentano la granularità atomica delle attività che sono state osservate durante l'esecuzione di un processo. Come tale, un evento ha una durata. Il nome del tag per l'oggetto event nella serializzazione XML di XES è: <event> Non sono definiti attributi XML per il tag <event>.



Le tre tipologie di oggetti di log, appena elencate, non contengono informazioni.
Esse definiscono solamente la struttura del documento. Tutte le informazioni, in un registro eventi, vengono memorizzate negli attributi.
Gli attributi descrivono il loro elemento genitore (log, trace, ecc.) Tutti gli attributi hanno una chiave basata su stringa. Lo standard XES richiede che l'attributo chiave:

- non contenga avanzamenti riga, ritorni a capo e tabs. Può contenere spazi iniziali o finali, o più spazi;

- sia univoco nel contenitore che lo racchiude (ad esempio, un solo attributo con la chiave _id_ per trace). L'unica eccezione a questa regola sono le chiavi all'interno di un elenco di inclusione. Poiché l'elenco impone un ordine su queste chiavi, hanno bisogno di non essere uniche.

Logs, trace ed event contengono ciascuno un numero arbitrario di attributi. Ci sono sei tipi di attributi elementari, ciascuno definito dal tipo di valore di dati che rappresenta:

- String: gli attributi String contengono informazioni letterali di lunghezza arbitraria che non sono generalmente tipizzate. Nella rappresentazione XML di XES, i valori degli attributi String vengono memorizzati come xs:string come tipologia di dati;

- Date: gli attributi Date contengono informazioni su uno specifico punto nel tempo (con precisione millisecondi). Nella rappresentazione XML di XES, i valori degli attributi Date vengono memorizzati come xs:dateTime come tipologia di dati;

- Int: gli attributi Int contengono un numero intero discreto (con 64bit). Nella rappresentazione XML di XES, i valori degli attributi Int vengono memorizzati come xs:long come tipologia di dati;

- Float: gli attributi Float contengono un numero continuo in virgola mobile continuo (con 64 bit). Nella rappresentazione XML di XES, i valori dell'attributo Float vengono memorizzati come xs:double come tipologia di dati;

- Boolean: gli attributi Boolean contengono un valore booleano che può essere vero o falso. Nella rappresentazione XML di XES, i valori degli attributi Boolean vengono memorizzati come xs:boolean come tipologia di dati;

- ID: gli attributi ID contengono informazioni sull'id che è generalmente un universally unique identifier (UUID). Nella rappresentazione XML di XES, i valori degli attributi ID vengono memorizzati come xs:string come tipologia di dati;

Accanto a questi attributi elementari, ci sono due tipi di attributi di raccolta:

- List: gli attributi List contengono un numero qualsiasi (può essere vuoto) di attributi figli. Questi attributi figlio sono ordinati e le loro chiavi non hanno bisogno di essere univoche. Il valore di un attributo List è derivato dai valori dei suoi attributi figli;

- Container: gli attributi Container contengono un numero qualsiasi (può essere vuoto) di attributi figli. Gli attributi figli non sono ordinati. Il valore di un attributo Container deriva dai valori dei suoi attributi figli.

Oltre a questi tipi di attributi standard esistono le estensioni, che consentono di aggiungere ulteriori informazioni in un event log. Alcune estensioni sono state standardizzate, in quanto, certe informazioni, risultavano necessarie per quasi ogni event log. Queste estensioni standardizzate sono le seguenti e ciascuna si riferisce ad un tipo diverso di dato:

- Concept: è l'identificativo delle attività la cui esecuzione ha generato l'evento;

- Lifecycle: serve ad indicare se è un evento è solo iniziato (start) oppure si è già concluso (complete);

- Organizational: è utile quando gli eventi sono stati realizzati da una persona che in qualche modo fa parte di una struttura organizzativa. Questa estensione comprende la risorsa, il ruolo e il gruppo;

- Time: definisce la data e l'ora esatta in cui sono stati registrati gli eventi. Il timestamp nella registrazione degli eventi è fondamentale per la maggior parte delle analisi;

- Semantic: è il riferimento ai concetti del modello in una ontologia.

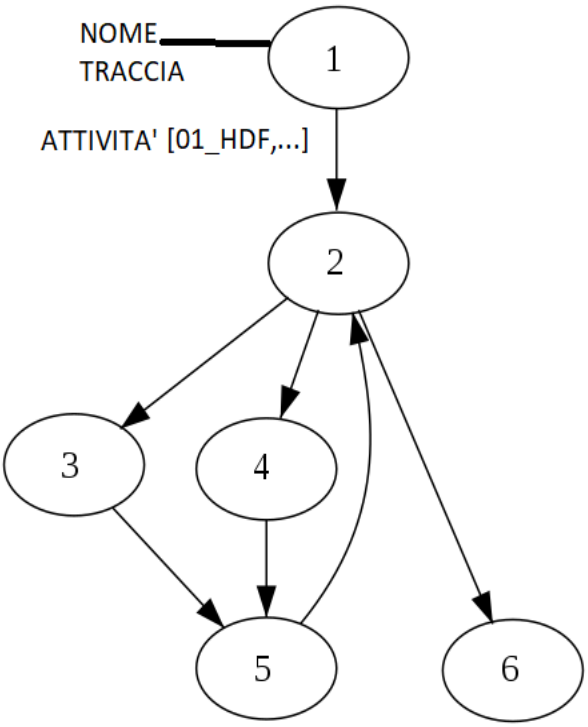
Implementazione

Verranno illustrate le parti principali del codice prodotto.

Le prime due classi, sviluppate in Python, che si occupano di costruire i modelli in esame, sono Node2Vec e Graph2Vec.

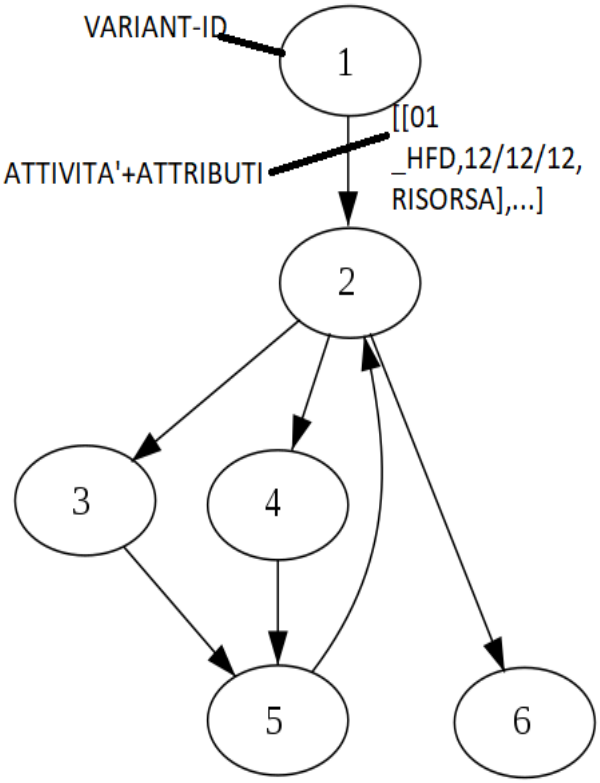
Per l'implementazione di Node2vec si utilizza un pacchetto specifico del linguaggio, basato su un apprendimento di Word2Vec di tipo skip-gram. Tramite questa implementazione, vengono prodotti due modelli. Il primo, produce un grafo orientato caratterizzato dalla corrispondenza "traccia-insieme di attività che la compongono", ispirandosi all'algoritmo Trace2vec. In particolar modo, ogni nodo conterrà il valore che identifica una traccia e i collegamenti tra i vari nodi, saranno creati sulla base delle attività contenute nella traccia stessa.

Precisamente, i nodi A e B saranno collegati tra loro se entrambi in possesso di almeno un'attività in comune. Per limiti di risorse hardware e del pacchetto Node2vec stesso, limitato per grandi reti con molte info, il collegamento tra due nodi sarà unico, anche se i due dovessero avere più attività in comune, con relativa etichettatura.



```
def learnT(logName,vectorsize):
    graph=nx.Graph()
    buildGT(graph,logName)
    node2vec = Node2Vec(graph, dimensions=vectorsize, walk_length=30, num_walks=200,workers=8)
    model = node2vec.fit(window=10, min_count=1, batch_words=4)
    model.save('output/'+'N2VVST'+str(vectorsize)+'.model')
```

Il secondo modello prodotto segue la falsa riga del primo. I nodi, del grafo orientato, conterranno l'id variant della traccia in esame, i collegamenti tra i nodi sono sviluppati in base al valore dell'attività svolta, con attributi relativi alla risorsa utilizzata e al timestamp relativo all'attività, ispirandosi all'algoritmo Act2vec. Il set di attività-attributi relativi ad ogni id-trace è limitato, così come il set di collegamenti ed etichettatura è limitato, anche qui, a causa dell'hardware e dell'implementazione stessa di Node2vec.



```
def learnV(logName,vectorsize):
    graph=nx.Graph()
    buildG(graph,logName)
    node2vec = Node2Vec(graph, dimensions=16, walk_length=30, num_walks=200,workers=8)
    model = node2vec.fit(window=10, min_count=1, batch_words=4)
    model.save('output/'+'N2VVS'+str(vectorsize)+'.model')
```

Gli altri metodi (si rimanda al codice) serviranno ad una buona costruzione dei grafi in esame. A titolo di esempio, si riporta la costruzione dell'albero per il modello "attività+attributi" di Node2Vec.

```
def buildG(graph,logName):
    max=5000
    attivita=[]
    for element in loadXES.Vget_sentences_XES(logName + ".xes"):
        attivita.append(element)
    NodiAttivita={}
    i=0
    c=0
    for variant in loadXES.get_variant_names(logName+".xes"):
        if(c<=max):
            NodiAttivita[variant]=attivita[i]
            i=i+1
            c=c+1
        else:
            break
    for key in NodiAttivita.keys():
        graph.add_node(key)
    for var1 in NodiAttivita.keys():
        for var2 in NodiAttivita.keys():
            if(var1!=var2):
                x = NodiAttivita[var1]
                y = NodiAttivita[var2]
                for el in x:
                    if(c<max):
                        if(y.__contains__(el)):
                            graph.add_edge(var1,var2,attr=el)
                            c=c+1
                            break
                    else:
                        break
```

I modelli prodotti saranno "allenati", sulla base dei grafi ottenuti seguendo i ragionamenti precedentemente elencati, utilizzando il metodo fit di Node2vec.

L'implementazione del modulo Graph2vec, segue la filosofia precedentemente elencata per Node2Vec. I grafi orientati prodotti, però, saranno implementati non più attraverso la classe Networkx, ma a livello logico; sotto forma di dizionari contenenti come chiave il trace id variant oppure la traccia stessa, e come valore associato alla chiave, l'insieme di attività o insieme di attività+attributi. Ogni elemento in comune andrà a costituire un collegamento nel grafo, con relativa etichettatura, ma solo e puramente a livello logico, in modo da alleggerire in maniera considerevole, il costo di costruzione dell'albero ed elaborazione del modello.

In questo modo, si può utilizzare molto agevolmente, la classe Doc2Vec implementa in gensim, permettendo inoltre, una facile lettura del codice. Il nodo sarà quindi una chiave, e due chiavi saranno considerate collegate, se avranno almeno una attività o attributo in comune, come avveniva precedentemente. Tutto questo permetterà, come sarà illustrato in seguito, di ottenere performance nettamente migliori, anche a causa della natura stessa dell'algoritmo implementato (Graph2vec è un Doc2Vec). Si ricorda che il set di attività o attributi utilizzati, è limitato dall'hardware in uso.

```
def learnT(logName,vectorsize):
    documents=buildGT(logName)
    model = gensim.models.Doc2Vec(documents, dm=0, alpha=0.025, vector_size=vectorsize, window=8, min_alpha=0.025,min_count=0, workers=4)
    nrEpochs=4
    for epoch in range(nrEpochs):
        if epoch % 2 == 0:
            print ('Now training epoch %s'%epoch)
        model.train(documents,total_examples=len(documents), epochs=nrEpochs)
        model.alpha -= 0.002
        model.min_alpha = model.alpha
    model.save('output/'+'G2VVST'+str(vectorsize)+'.model')

def learnV(logName,vectorsize):
    documents=buildG(logName)
    model = gensim.models.Doc2Vec(documents, dm=0, alpha=0.025, vector_size=vectorsize, window=8, min_alpha=0.025,min_count=0, workers=4)
    nrEpochs=4
    for epoch in range(nrEpochs):
        if epoch % 2 == 0:
            print ('Now training epoch %s'%epoch)
        model.train(documents,total_examples=len(documents), epochs=nrEpochs)
        model.alpha -= 0.002
        model.min_alpha = model.alpha
    model.save('output/'+'G2VVS'+str(vectorsize)+'.model')
```

"L'allenamento" del grafo è fatto sfruttando il metodo train, implementato nella classe Doc2Vec.

```
def buildG(logName):
    grafoS=[]
    attivita=[]
    for element in loadXES.Vget_sentences_XES(logName + ".xes"):
        attivita.append(element)
    NodiAttivita={}
    i=0
    for variant in loadXES.get_variant_names(logName+".xes"):
            NodiAttivita[variant]=attivita[i]
            i=i+1
    for key in NodiAttivita:
        t=TaggedDocument((np.concatenate(NodiAttivita[key])),[key])
        grafoS.append(t)
    return grafoS
```

L'analisi dei modelli ottenuti, sarà fatta dal modulo MyModel2Vec. Ispirandosi al Model2Vec spiegato precedentemente, vengono implementati i cluster relativi al Trace2Vec e Act2Vec, che andranno a formare una parte del modulo Log2Vec, suggerito come analisi per il cluster prodotto da Model2Vec. Si specifica, che il modulo prodotto, non implementa la parte relativa all'identificazione di vari modelli con ID specifici, in quanto non utile al fine dell'esperimento in oggetto.

Per il cluster sui modelli ottenuti, si utilizzano gli algoritmi KMeans e Hierward. Si mostra, a titolo di esempio, il cluster relativo al variant-id, molto simile a quello prodotto dal cluster sulle tracce. Per le due implementazioni complete, si rimanda al codice.

```
def clusterV(logName,vectorsize,nameM,clusterType):
    corpus = loadXES.get_doc_XES_tagged(logName+'.xes')
    vectors = []
    NUM_CLUSTERS = 5
    conta=[]
    if(nameM=='G2VVS'):
        model= gensim.models.Doc2Vec.load('output/'+nameM+str(vectorsize)+'.model')
        print("inferring vectors")
        for variant in range(len(corpus)):
            if (corpus[variant].words not in conta):
                inferred_vector = model.infer_vector(corpus[variant].words)
                vectors.append(inferred_vector)
                conta.append(corpus[variant].words)
    elif(nameM=='N2VVS'):
        model=gensim.models.KeyedVectors.load('output/'+nameM+str(vectorsize)+'.model')
        print("model vectors")
        for variant in loadXES.get_variant_names(logName+".xes"):
            if(model.wv.__contains__(variant) and variant not in conta):
                model_vector=model.wv.get_vector(variant)
                vectors.append(model_vector)
                conta.append(variant)
    print("done")
```

```
if(clusterType=="KMeans"):
    kclusterer = KMeansClusterer(NUM_CLUSTERS, distance=nltk.cluster.util.cosine_distance, repeats=25)
    assigned_clusters = kclusterer.cluster(vectors, assign_clusters=True)
elif(clusterType=="HierWard"):
    ward = AgglomerativeClustering(n_clusters=NUM_CLUSTERS, linkage='ward').fit(vectors)
    assigned_clusters = ward.labels_
else:
    print(clusterType, " is not a predefined cluster type. Please use 'KMeans' or 'HierWard', or create a definition for ", clusterType)
    return
clusterResult= {}
```

```
  if(nameM=='G2VVS'):
      variant_list = loadXES.get_variant_names(logName + ".xes")
      fatte=[]
      for variant in range(len(conta)):
          if(variant_list[variant] not in fatte):
              clusterResult[variant_list[variant]]=assigned_clusters[variant]
              fatte.append(variant_list[variant])
      resultFile= open('output/'+nameM+str(vectorsize)+clusterType+'.csv','w')
      fatte=[]
      for variant in range(len(conta)):
          if(variant_list[variant] not in fatte):
              resultFile.write(variant_list[variant]+','+str(assigned_clusters[variant])+"\n")
              fatte.append(variant_list[variant])
      resultFile.close()
  elif(nameM=='N2VVS'):
      for variant in range(0, len(conta)):
          clusterResult[conta[variant]] = assigned_clusters[variant]
      resultFile = open('output/'+nameM+str(vectorsize)+clusterType +'.csv', 'w')
      for variant in range(0, len(conta)):
          resultFile.write(conta[variant] + ',' + str(assigned_clusters[variant]) + "\n")
      resultFile.close()
  print("done")
```

I modelli precedentemente prodotti, servono ad ottenere la rappresentazione della traccia o variant-id, caratterizzata dal set di attività o attività+ attributi, associata al Word2Vec o Doc2Vec, alla base di Node2Vec e Graph2Vec. La rappresentazione di questi vettori, cambierà anche in base alla dimensione dei vettori stessi associati al modello. I risultati dei vari clustering, saranno salvati in file .csv.

L'ultimo modulo implementato è MyLog2Vec, che conterrà gli stessi metodi di clustering di Model2Vec.

MyLog2vec, però, non utilizza i modelli prodotti da Node2Vec o Graph2Vec ma analizza il file log in esame, produce un suo modello. Il clustering prodotto, sarà utilizzato come metrica di riferimento per i test dei nostri modelli.

Il clustering è effettuato in maniera simile ai precedenti, per ulteriori info si rimanda al codice.

```
def learn(logName,vectorsize):
    documents = loadXES.get_doc_XES_tagged(logName+'.xes')
    model = gensim.models.Doc2Vec(documents, dm = 0, alpha=0.025, vector_size= vectorsize, window=3, min_alpha=0.025, min_count=0)
    nrEpochs= 4
    for epoch in range(nrEpochs):
        if epoch % 2 == 0:
            print ('Now training epoch %s'%epoch)
        model.train(documents,total_examples=len(documents), epochs=nrEpochs)
        model.alpha -= 0.002
        model.min_alpha = model.alpha
    model.save('output/'+'L2VVS'+str(vectorsize) +'.model')

def cluster(logName,vectorsize,clusterType):
    clusterT(logName,vectorsize,clusterType)
    clusterV(logName,vectorsize,clusterType)
```

Testing

Il potere diagnostico di un test è di per sé un concetto multidimensionale, in quanto include la sensibilità, la specificità, il potere predittivo positivo, il potere predittivo negativo e l'accuratezza.

I test effettuati includono vari tipi di analisi, in particolar modo sono stati:

1. L'informazione mutua (MI) di due variabili casuali è una quantità che misura la mutua dipendenza delle due variabili. La più comune unità di misura della mutua informazione è il bit, quando si usano i logaritmi in base 2. Intuitivamente, l'informazione mutua misura l'informazione che *X* e *Y* condividono: essa misura quanto la conoscenza di una di queste variabili riduce la nostra incertezza riguardo all'altra. Ad esempio, se *X* e *Y* sono indipendenti, allora la conoscenza di *X* non dà alcuna informazione riguardo a *Y* e viceversa, perciò la loro mutua informazione è zero. All'altro estremo, se *X* e *Y* sono identiche allora tutte le informazioni trasmesse da *X* sono condivise con *Y*: la conoscenza di *X* determina il valore di *Y* e viceversa. Come risultato, nel caso di identità l'informazione mutua è la stessa contenuta in *Y* (o *X*) da sola, vale a dire l'entropia di *Y* (o *X*:

chiaramente se *X* e *Y* sono identiche, hanno identica entropia).La variante utilizzata nel testing è l'informazione mutua normalizzata che varia nel calcolo del coefficiente di vincolo e di incertezza.
NMI è una normalizzazione del punteggio MI per ridimensionare i risultati tra 0 (nessuna informazione reciproca) e 1 (correlazione perfetta). In questa funzione, le informazioni reciproche sono normalizzate da una media generalizzata di H.
Questa metrica è stata utilizzata in quanto indipendente dai valori assoluti delle etichette: una permutazione dei valori delle etichette della classe o del cluster non modificherà il valore del punteggio in alcun modo; è inoltre simmetrica: il passaggio da "label_true a label_pred" restituirà lo stesso valore di punteggio. Questo può essere utile per misurare l'accordo di due strategie di assegnazione di etichette indipendenti sullo stesso set di dati quando non si conosce la vera verità di base.

2. Il Rand Index è utilizzato per misurare la somiglianza tra 2 cluster di dati; essendo correlato alla precisione. In particolare, verrà utilizzata la versione Adjusted Rand Index che stabilisce una linea di base usando la somiglianza attesa di tutti i confronti a coppie tra cluster raggruppati specificati da un modello casuale. Tradizionalmente, l'Indice Rand veniva corretto utilizzando il Modello di permutazione per i cluster (il numero e la dimensione dei cluster all'interno di un cluster sono fissi e tutti i cluster casuali sono generati mescolando gli elementi tra i cluster fissi). Tuttavia, le premesse del modello di permutazione sono frequentemente violate in molti scenari di clustering, infatti, il numero di cluster o la distribuzione delle dimensioni di tali cluster variano drasticamente, quando, ad esempio, le dimensioni di tali cluster sono dedotte dai dati. Sebbene l'indice Rand possa produrre solo un valore compreso tra 0 e +1, l'indice Rand rettificato può produrre valori negativi se l'indice è inferiore all'indice previsto.
Ricapitolando, l'indice Rand calcola una misura di somiglianza tra due cluster tenendo conto di tutte le coppie di campioni e contando le coppie assegnate nello stesso o in diversi cluster nei cluster previsti e veri; è simmetrico se si usa ARI e in questo caso verrà utilizzato per testare la bontà dell'algoritmo in esame.

$$AdjustedIndex = \frac{Index - ExpectedIndex}{MaxIndex - ExpectedIndex}$$

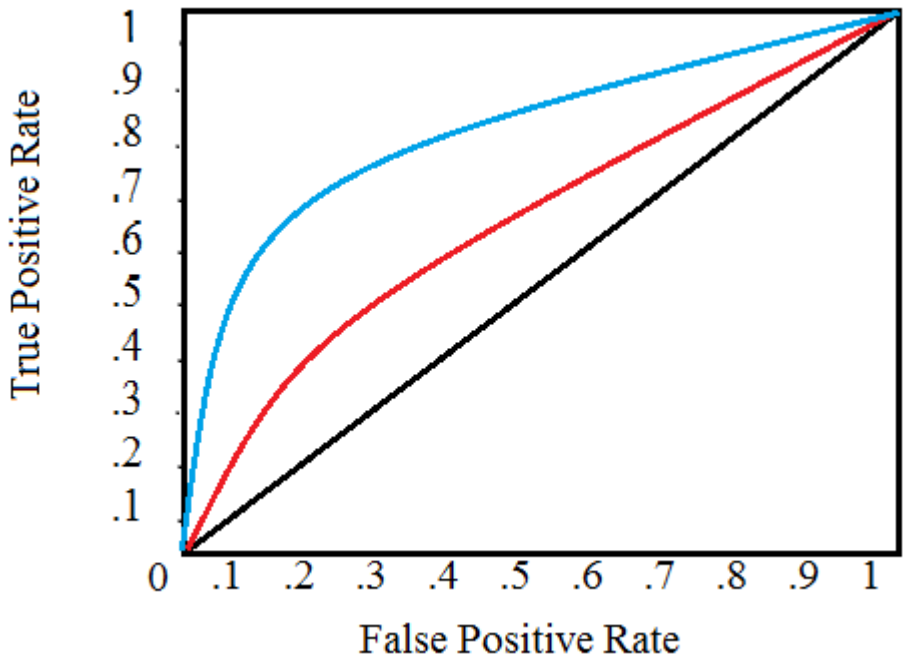
Given the contingency table:

	Y_1	Y_2	$\cdots$	Y_s	$Sums$
X_1	n_{11}	n_{12}	$\cdots$	n_{1s}	a_1
X_2	n_{21}	n_{22}	$\cdots$	n_{2s}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_r	n_{r1}	n_{r2}	$\cdots$	n_{rs}	a_r
$Sums$	b_1	b_2	$\cdots$	b_s	

the adjusted index is:

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - [\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}{\frac{1}{2}[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}] - [\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}$$

3. La curva ROC viene costruita considerando tutti i possibili valori del test e, per ognuno di questi, si calcola la proporzione di veri positivi (la sensibilità) e la proporzione di falsi positivi. La proporzione di falsi positivi si calcola con la formula standard: 1 - specificità. Congiungendo i punti che mettono in rapporto la proporzione di veri positivi e di falsi positivi (le cosiddette coordinate) si ottiene una curva chiamata curva ROC. L'area sottostante alla curva ROC (AUC, acronimo dei termini inglesi "Area Under the Curve") è una misura di accuratezza diagnostica. Se un ipotetico nuovo test discriminasse perfettamente i malati dai sani, l'area della curva ROC avrebbe valore 1, cioè il 100% di accuratezza. Nel caso in cui il nuovo test non discriminasse per niente i malati dai sani, la curva ROC avrebbe un'area di 0.5 (o 50%) che coinciderebbe con l'area sottostante la diagonale del grafico.L'area sotto la curva può assumere valori compresi tra 0.5 e 1.0. Tanto maggiore è l'area sotto la curva (cioè tanto più la curva si avvicina al vertice del grafico) tanto maggiore è il potere discriminante del test.
   Le curve ROC sono generalmente utilizzate nella classificazione binaria per studiare l'output di un classificatore. Per estendere la curva ROC e l'area ROC alla classificazione multietichetta, è necessario binarizzare l'output. Il punto in alto a sinistra della trama è il punto "ideale", un tasso di falsi positivi pari a zero e un tasso di veri positivi di uno. Questo non è molto realistico, ma significa che un'area più ampia sotto la curva (AUC) è generalmente migliore. Anche la "pendenza" delle curve ROC è importante, poiché è ideale per massimizzare il tasso positivi reali minimizzando il tasso di falsi positivi.



4. AUC score, indica l'area relativa alla curva di ROC.

Data la stretta correlazione tra curva di ROC e F1 score, quest'ultima si presta meglio ad implementare i modelli in esame, consentendo una migliore visualizzazione dei risultati sperimentali.

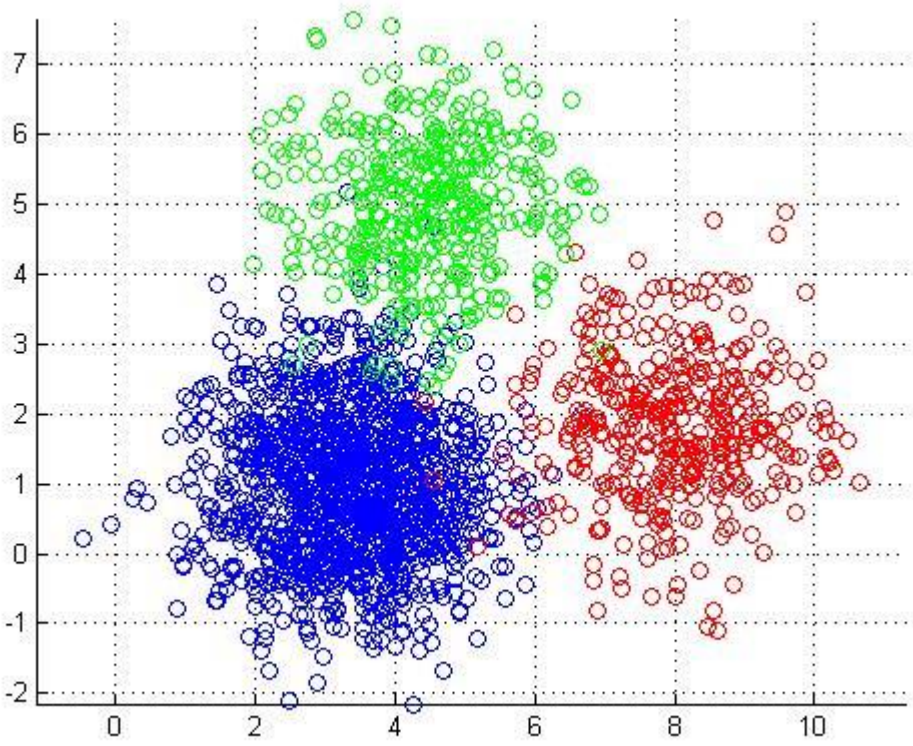
Le tipologie di algoritmi di clustering utilizzate sono due:

1) Il K-Means è un algoritmo di apprendimento non supervisionato che trova un numero fisso di cluster in un insieme di dati. I cluster rappresentano i gruppi che dividono gli oggetti a seconda della presenza o meno di una certa somiglianza tra di loro, e vengono scelti a priori, prima dell'esecuzione dell'algoritmo. Ognuno di questi cluster raggruppa un particolare insieme di oggetti, che vengono definiti data points. L'insieme dei data points analizzati definisce il set di dati, che rappresenta l'insieme di tutte le istanze analizzate dall'algoritmo. Quando si utilizza un algoritmo K-Means, per ogni cluster si definisce un centroide, ossia un punto (immaginario o reale) al centro di un cluster. L'algoritmo k-means è un algoritmo iterativo, ossia che esegue ripetutamente alcune sue fasi e fondamentalmente si può affermare che è formato dai seguenti step:

   -Inizializzazione: si definiscono i parametri di input per eseguire l'algoritmo;

   -Assegnazione del cluster: ogni data points viene assegnato al cluster (o centroide) più vicino;

   -Aggiornamento della posizione del centroide: ricalcola il punto esatto del centroide e di conseguenza ne modifica la sua posizione.



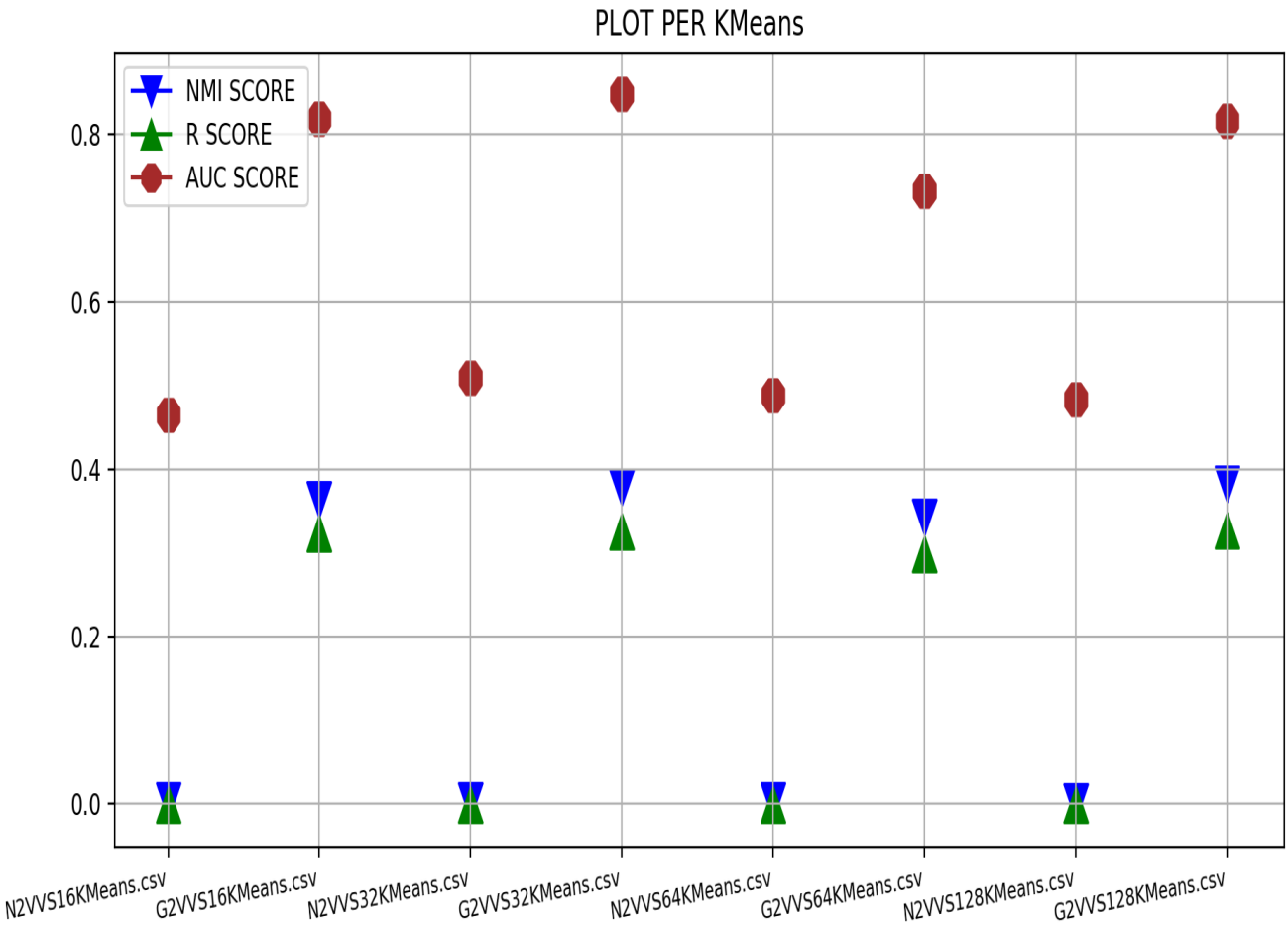
L’algortimo k-means è molto adatto per scenari in cui è possibile creare gruppi di oggetti simili da una collezione di oggetti distribuiti casualmente; ha il vantaggio di essere abbastanza veloce, in quanto sono richiesti pochi calcoli, e di conseguenza poco tempo di elaborazione al computer, per calcolare le distanze tra i data points e i centroidi ad ogni iterazione (ovviamente ciò dipende dal set di dati e dal numero di cluster presenti).D’altra parte, k-means ha un paio di svantaggi.Innanzitutto, bisogna selezionare quanti gruppi si vogliono visualizzare. Questo non è sempre banale, soprattutto per problemi di complessità maggiore. In aggiunta, K-means inizia con una scelta casuale di centroidi e pertanto può produrre risultati di clustering diversi su diverse sequenze dell’algoritmo. Pertanto, i risultati potrebbero non essere ripetibili e mancare di coerenza.

2) HierWard è il clustering gerarchico agglomerativo di Ward. Utilizza un approccio "bottom up" (dal basso verso l'alto) in cui si parte dall'inserimento di ciascun elemento in un cluster differente e si procede quindi all'accorpamento graduale di cluster a due a due. Il metodo utilizza, per calcolare le distanze tra i punti, la somma delle differenze quadrate all’interno dei cluster. E’ un approccio che minimizza la varianza, simile alla funzione obiettivo di k-means affrontata con un approccio gerarchico diverso.

Dopo queste premesse teoriche, passiamo ora ad analizzare i risultati ottenuti.
Nei modelli generati, per ogni algoritmo implementato, la dimensione dei vettori utilizzati per rappresentare le informazioni del modello, varia da 16 a 128 bit. I risultati di KMeans e HierWard in base al modello(“traccia-attività” oppure “variante traccia-attività+attributi”) con Node2Vec o Graph2Vec, sono stati salvati in file di tipo .csv. Le analisi effettuate sui clustering, sono salvate nel file risultati.csv in opportune tabelle, di facile lettura ed interpretazione. La metrica di riferimento per l’applicazione dei test, è rappresentata dai risultati ottenuti da MyLog2Vec, in grado di analizzare al meglio il file log in esame.

Di seguito i principali risultati prodotti:

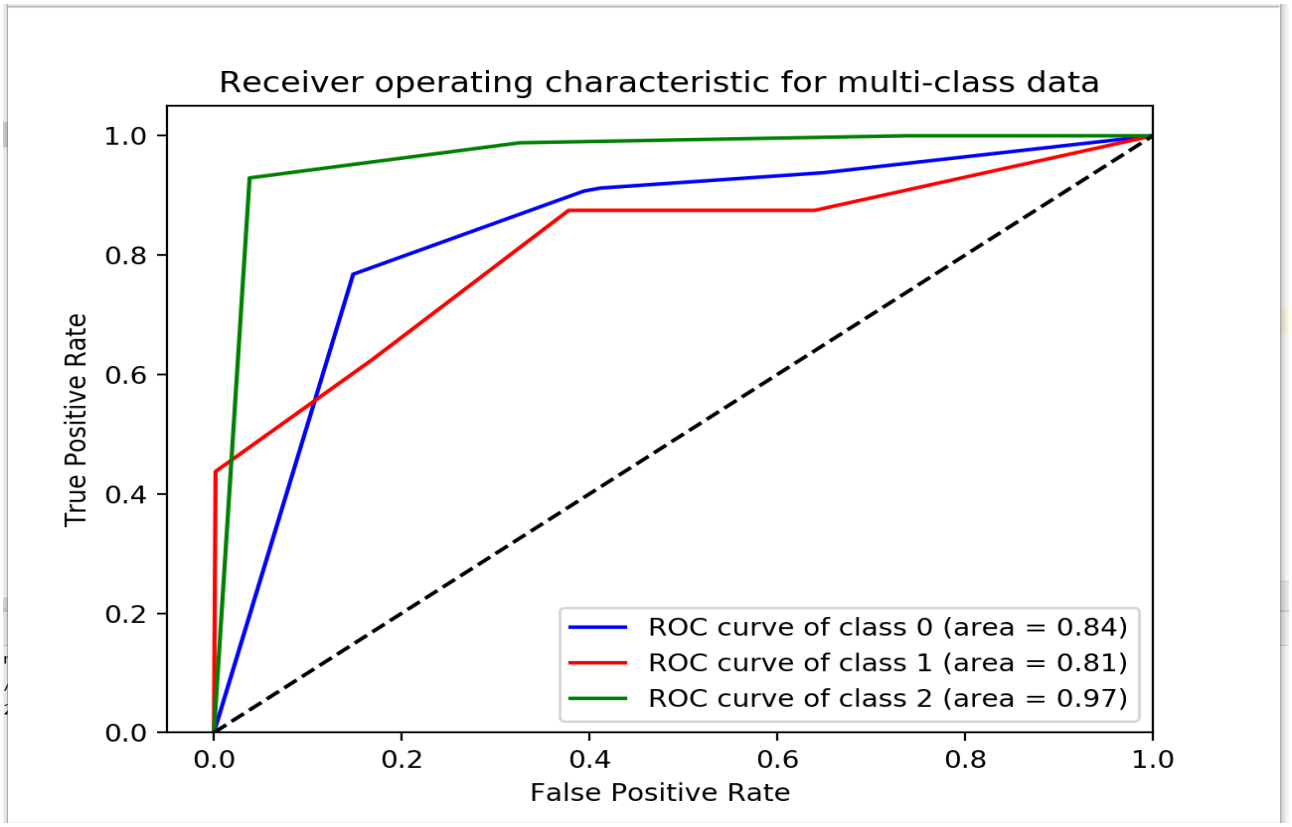
Tabella N2VVSKMeans			
	NMI	RI	AUC
N16	0,004	-0,0007	0,49
N32	0,0003	-0,0009	0,5
N64	0,003	-0,0007	0,507
N128	0,003	-0,0006	0,5
Tabella G2VVSKMeans			
	NMI	RI	AUC
G16	0,36	0,32	0,85
G32	0,38	0,33	0,87
G64	0,35	0,3	0,78
G128	0,38	0,32	0,82



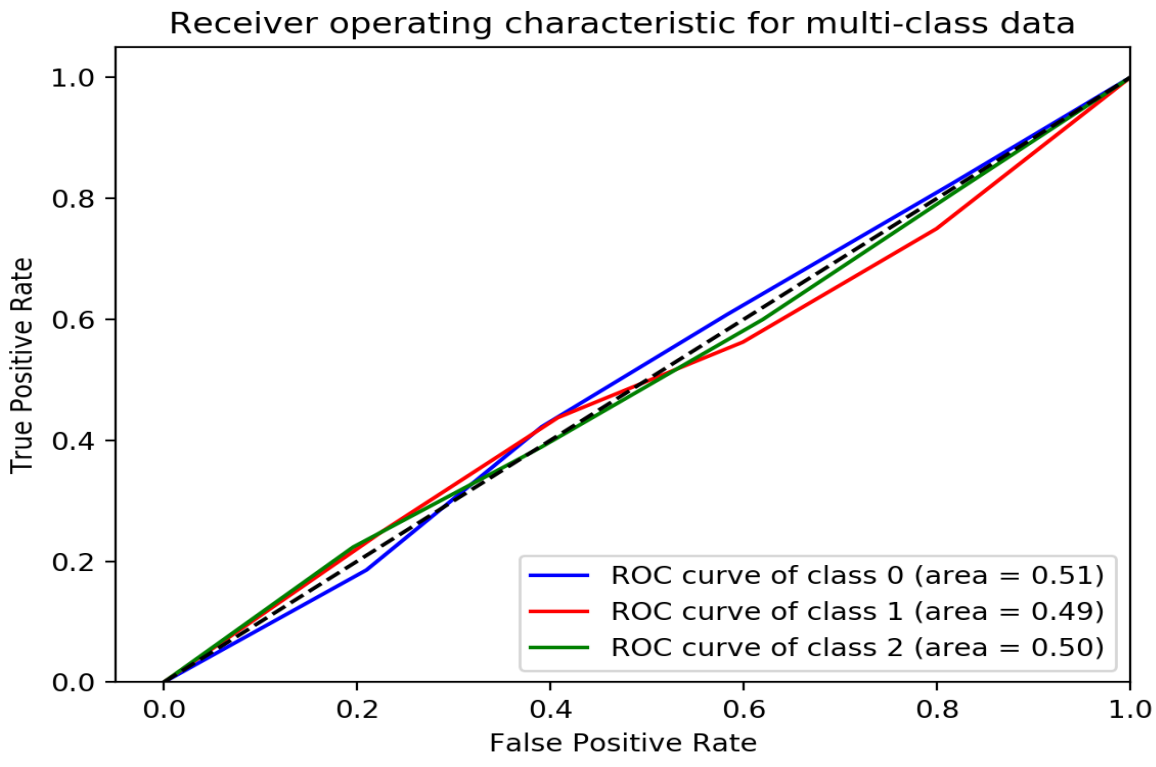
Analizzando i risultati ottenuti da K-Means, è evidente la migliore capacità di Graph2Vec nell'ottenere buoni risultati. L'indice NMI nella configurazione "variante traccia-attività+attributi" ha un range di 0,36<=x<=0,38 contro i 0,0003<=y<=0,003, così come quello RI 0,30<=x<=0,33 contro i -0,0009<=y<=-0,0006, valore negativo che indica un indice diverso da quello atteso per Node2Vec.

Come ultima conferma, si ottengono valori dell'area della curva di ROC per Graph2Vec compresi 0,78<=x<=0,87 contro i 0,49<=y<=0,507 di Node2Vec; indicando l'impossibilità del test di decidere per quest'ultimo modello, contro un'eccellente accuratezza del primo. A titolo di esempio si riportano le configurazioni delle curve ottenute.

Graph2vec



Node2Vec

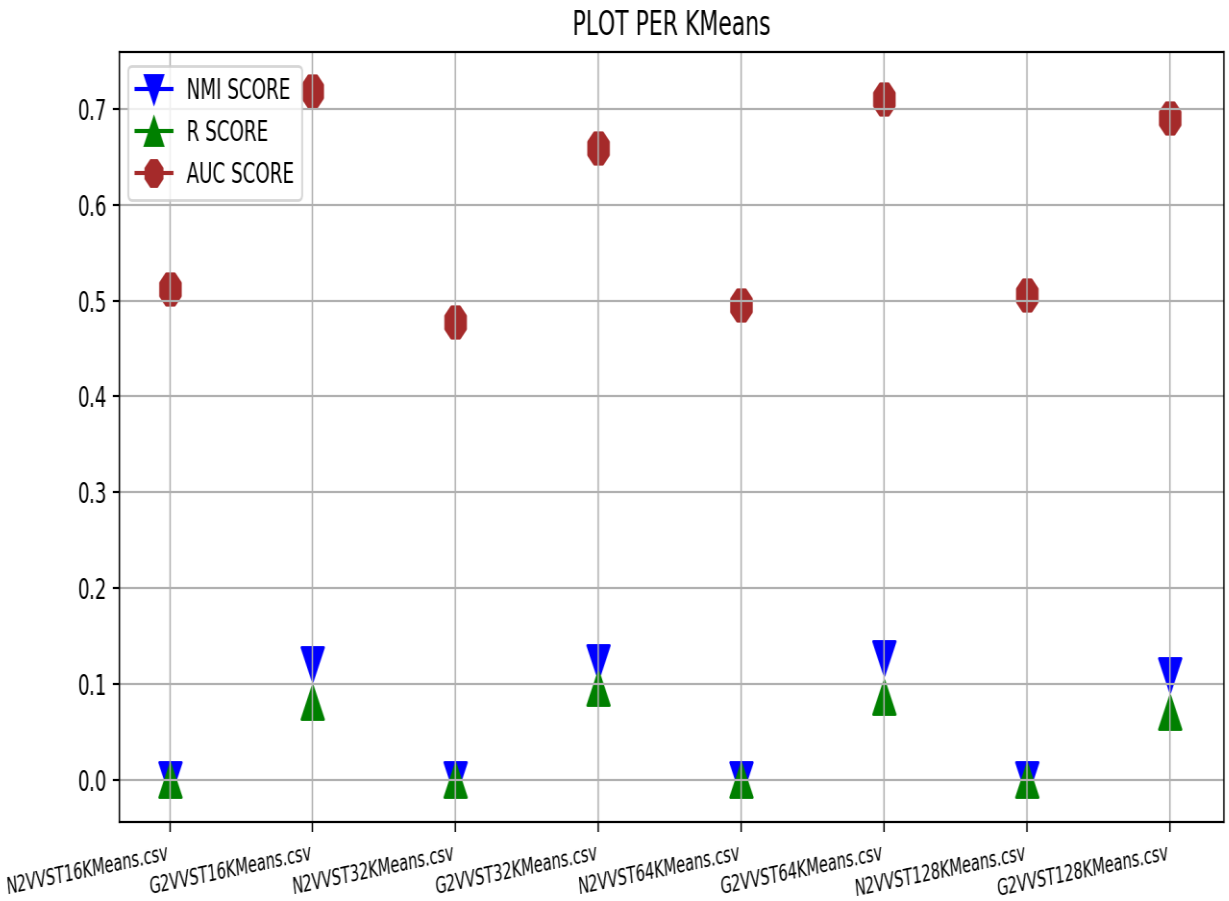


Per Node2Vec, il test della curva di ROC non è informativo, mentre per Graph2Vec è un test moderatamente accurato.

Analizziamo ora la rappresentazione del cluster “tracce-attività”.

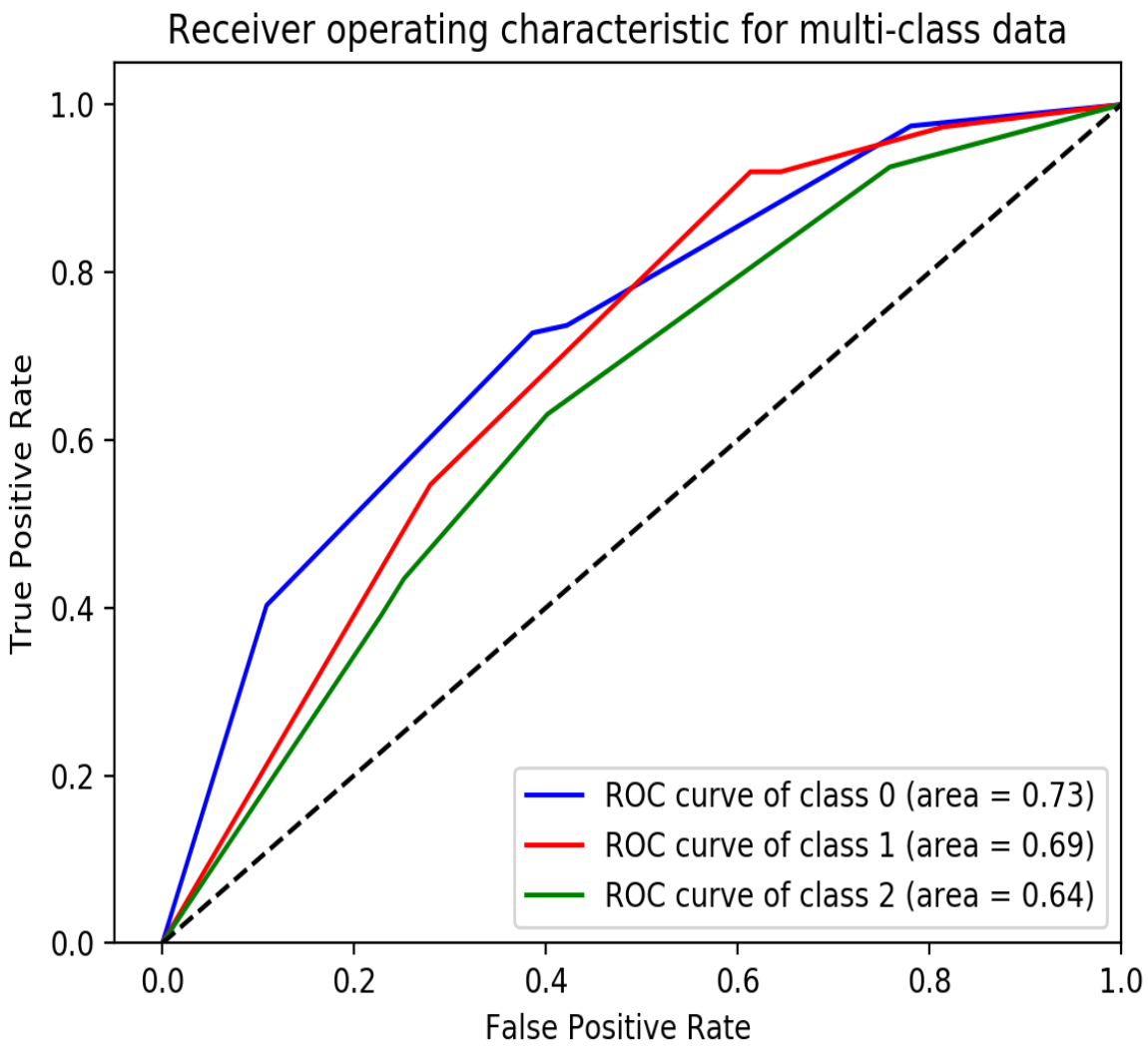
Tabella N2VVSTKMeans			
	NMI	RI	AUC
N16	0,001	0,000087	0,51
N32	0,001	0,0001	0,46
N64	0,0009	0,00008	0,49
N128	0,0011	0,0002	0,5

Tabella G2VVSTKMeans			
	NMI	RI	AUC
G16	0,12	0,08	0,67
G32	0,12	0,09	0,69
G64	0,12	0,08	0,67
G128	0,11	0,08	0,67



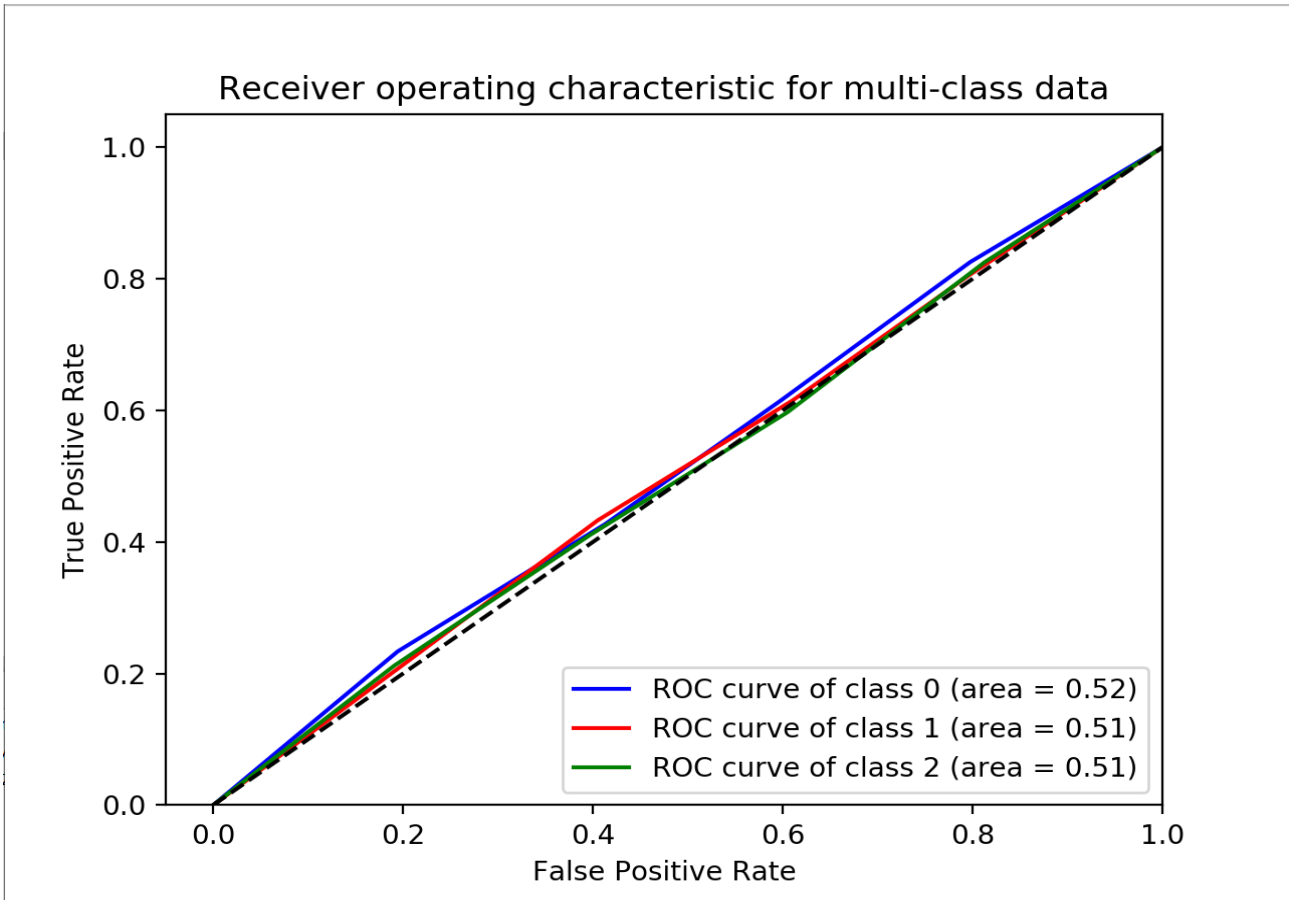
Anche qui Graph2Vec si comporta meglio, nonostante i valori NMI, RI e dell'area della curva di ROC siano più bassi. Il valore di RI qui non è negativo (non ho un valore diverso dall'atteso) per Node2Vec, ma di gran lunga inferiore rispetto a quello di Graph2Vec.

Graph2Vec



L'analisi delle curve di ROC, con i risultati AUC tabellati, mostrano chiaramente che questo test è nuovamente non informativo per Node2Vec; mentre è poco accurato per Graph2Vec, producendo comunque un risultato migliore rispetto all'algoritmo precedente.

Node2Vec



Passando alla tipologia di clustering gerarchico HierWard, si ottiene la seguente tabella:

Tabella N2VVSHW

	NMI	RI	AUC
N16	0,005	0,001	0,6
N32	0,003	-0,003	0,49
N64	0,003	-0,001	0,5
N128	0,002	-0,005	0,48

Tabella N2VVSTHW

	NMI	RI	AUC
N16	0,001	0,001	0,51
N32	0,0009	0,001	0,49
N64	0,0008	0,0004	0,49
N128	0,001	0,003	0,51

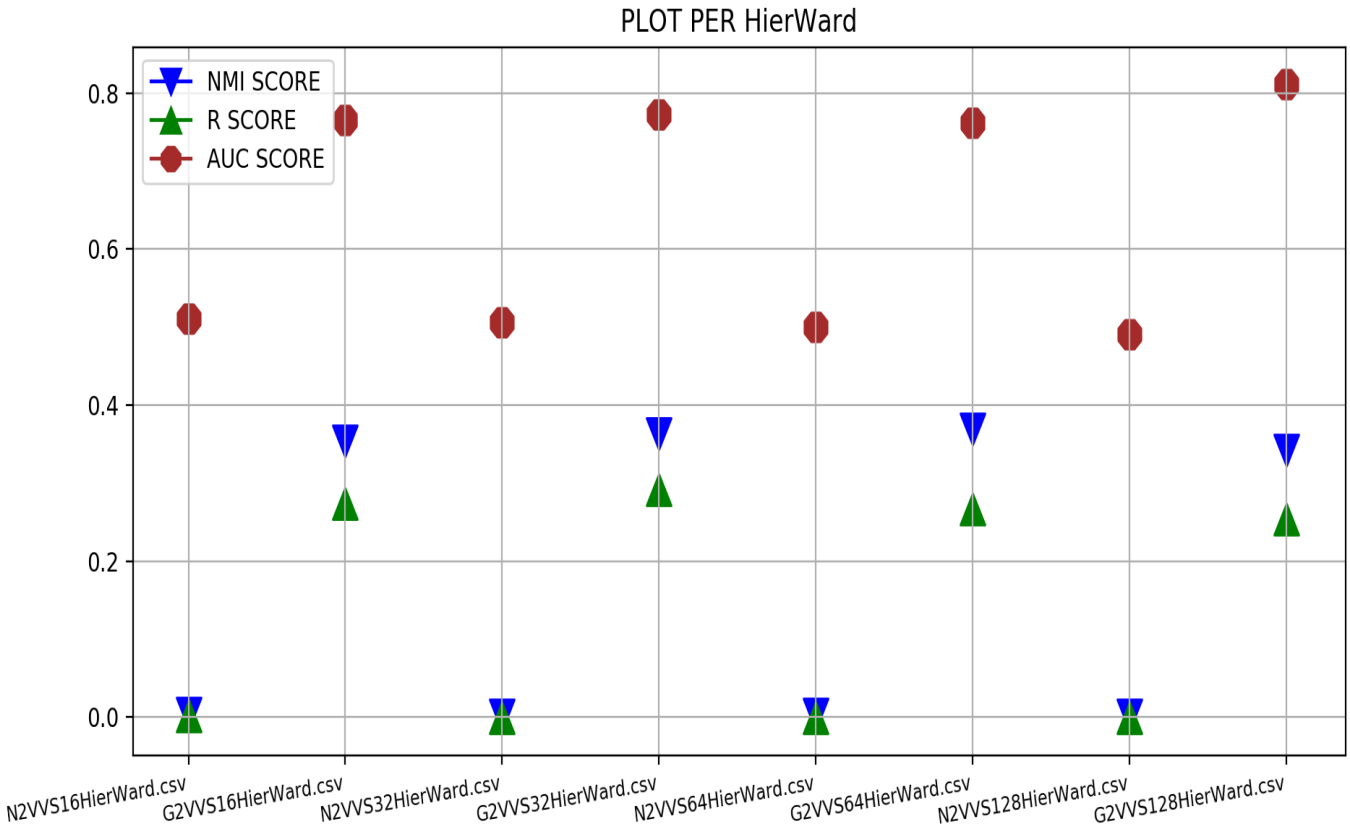
Tabella G2VVSHW

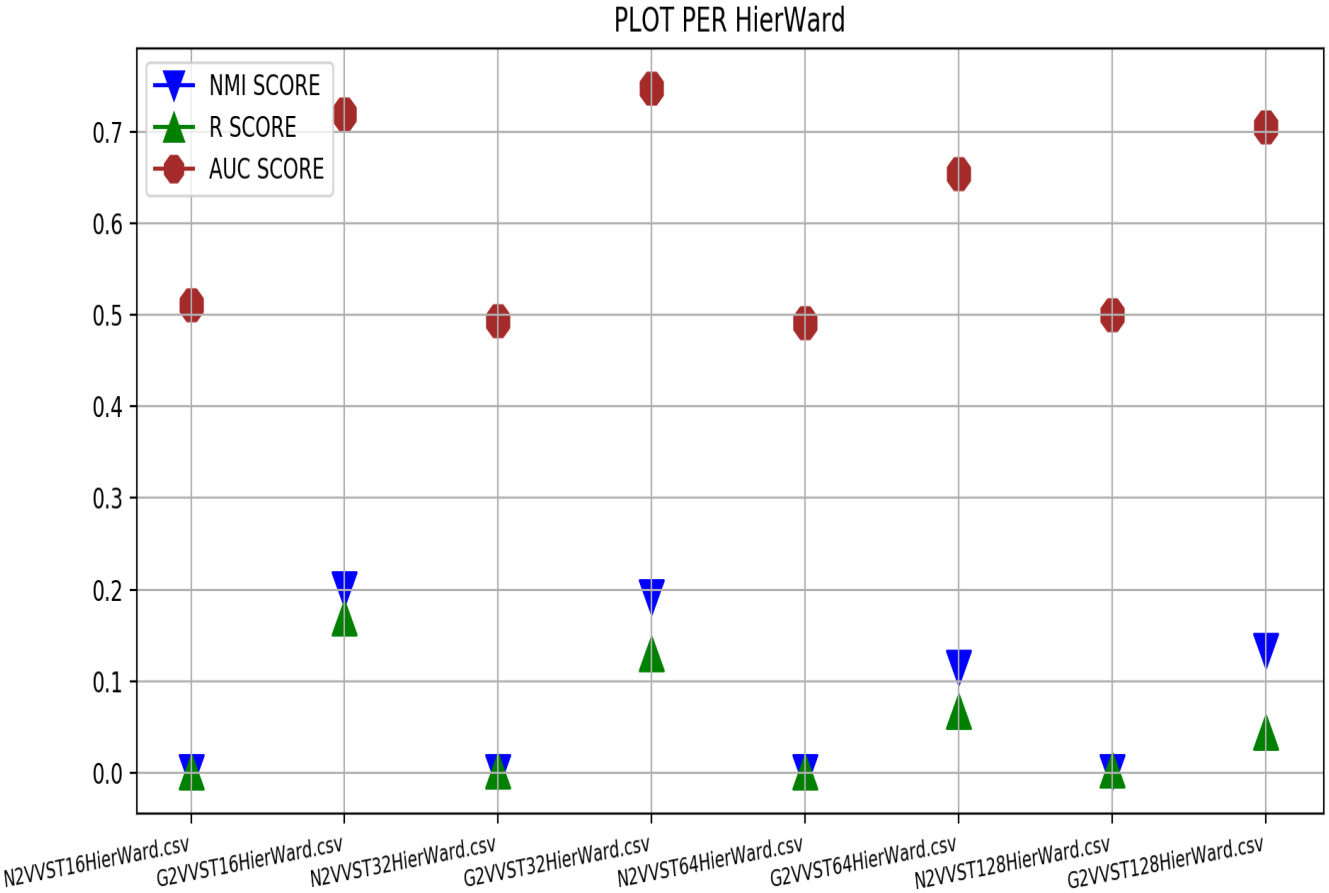
	NMI	RI	AUC
G16	0,35	0,27	0,85
G32	0,36	0,29	0,82
G64	0,37	0,28	0,83
G128	0,34	0,25	0,84

Tabella G2VVSTKHW

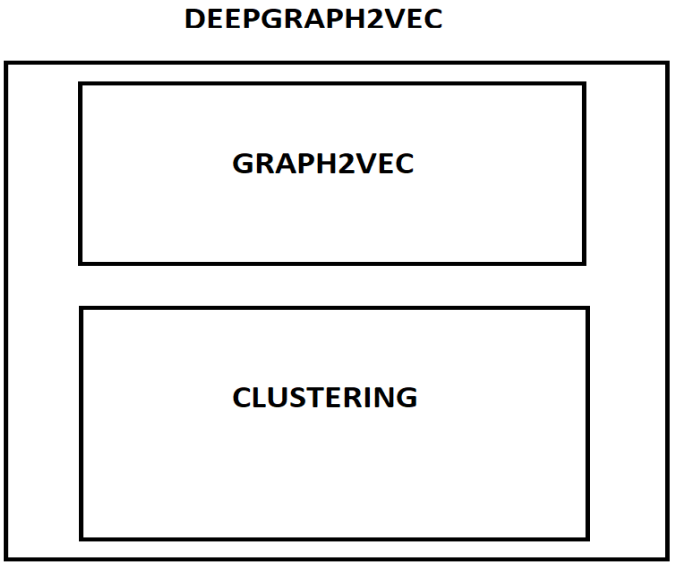
	NMI	RI	AUC
G16	0,2	0,17	0,73
G32	0,19	0,13	0,73
G64	0,11	0,07	0,67
G128	0,13	0,04	0,73

I risultati ottenuti per tutti e 4 i modelli, ricalcano la falsa riga di KMeans, non necessitando di ulteriori spiegazioni.


PLOT PER HierWard
NMI SCORE
R SCORE
AUC SCORE
0.8
0.6
0.4
0.2
0.0
N2VVS16HierWard.csv
G2VVS16HierWard.csv
N2VVS32HierWard.csv
G2VVS32HierWard.csv
N2VVS64HierWard.csv
G2VVS64HierWard.csv
N2VVS128HierWard.csv
G2VVS128HierWard.csv


PLOT PER HierWard
NMI SCORE
R SCORE
AUC SCORE
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
N2VVST16HierWard.csv
G2VVST16HierWard.csv
N2VVST32HierWard.csv
G2VVST32HierWard.csv
N2VVST64HierWard.csv
G2VVST64HierWard.csv
N2VVST128HierWard.csv
G2VVST128HierWard.csv

Possiamo concludere che la nostra architettura ideale per l'analisi sui grafi deve utilizzare, per la formazione dei modelli, Graph2Vec nettamente migliore di Node2Vec.



A titolo di completezza, si riportano in aggiunta a questi test, un'analisi del file log esaminato.

Le attività, le risorse e gli attori più utilizzati, sono stati catalogati in un dizionario, facilmente consultabile dal codice.

```
Elaboro info dal log...
Attività con relativa frequenza:
[('01_HOOFD', 36590), ('08_AWB45', 5995), ('09_AH', 4749)
Risorse con relativa frequenza:
[('560752', 5149), ('560454', 4851), ('560604', 4408), ('
Attori con relativa frequenza:
[('560604', 7971), ('560852', 6924), ('4901428', 6147), (
```

Si può ammirare la top 3 delle attività, risorse e attori che compaiono con maggiore frequenza.