**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:

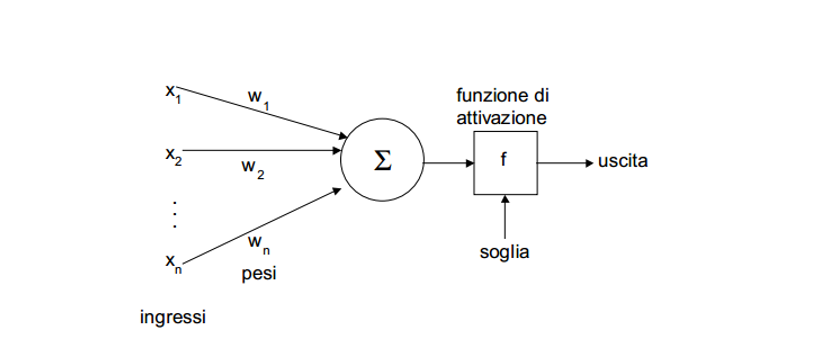
“Un algoritmo di apprendimento è in grado di imparare da una certa esperienza E rispetto a una classe di task T e misura di performance P, se la performance nei task in T, misurata secondo P, aumenta con l’esperienza E”.

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.

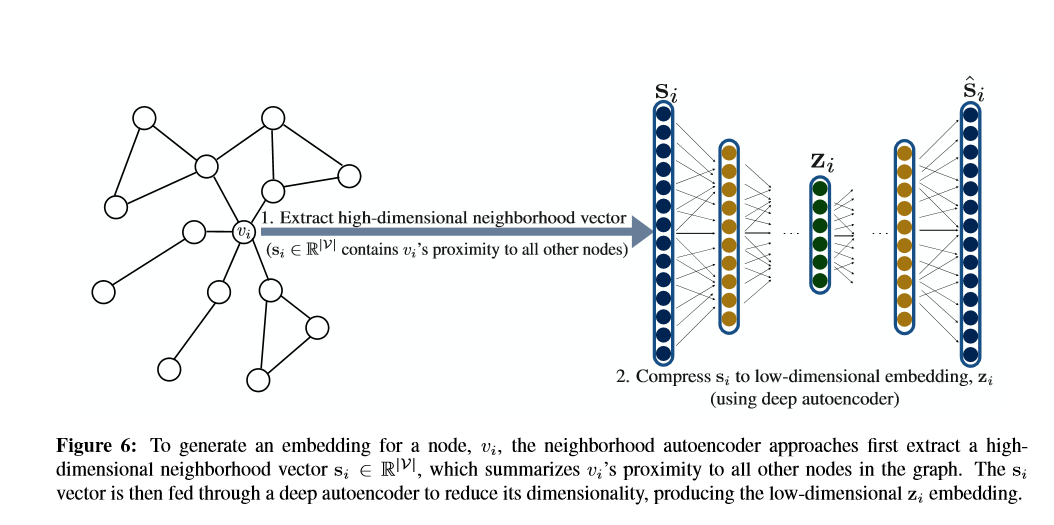
L’embeddings è necessario in quanto un semplice grafo, che utilizza uno specifico gruppo si argomenti matematici fisici o statistici, è limitato in questo contesto di utilizzo; al contrario dei vettori, che possono essere rimodellati con vari approcci e hanno un set di operazioni decisamente più veloci.

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.



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.

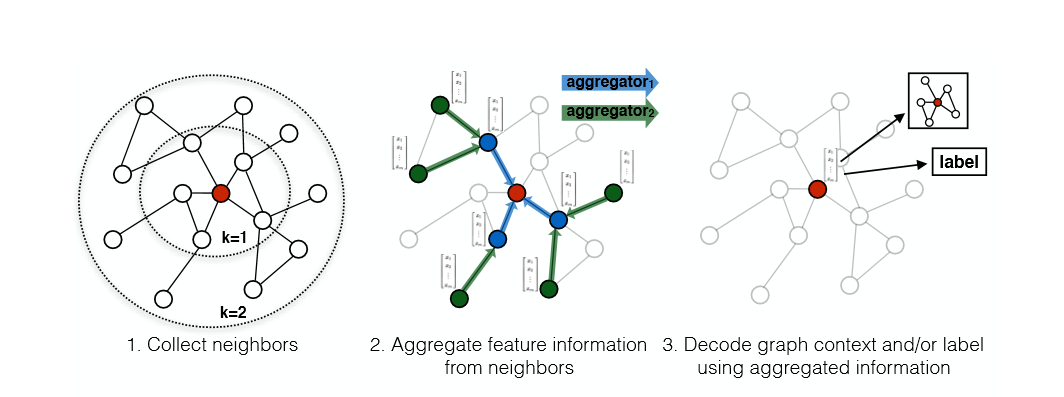
Con l’appropriato settaggio della dimensionalità e dei relativi vincoli sulla dispersione del dato, attraverso gli autoencoder si possono ottenere proiezioni in sottospazi di interesse maggiore rispetto a metodi lineari.

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 nell’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.



**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),di un vettore dato come input, successivamente decompresso nuovamente (nel livello

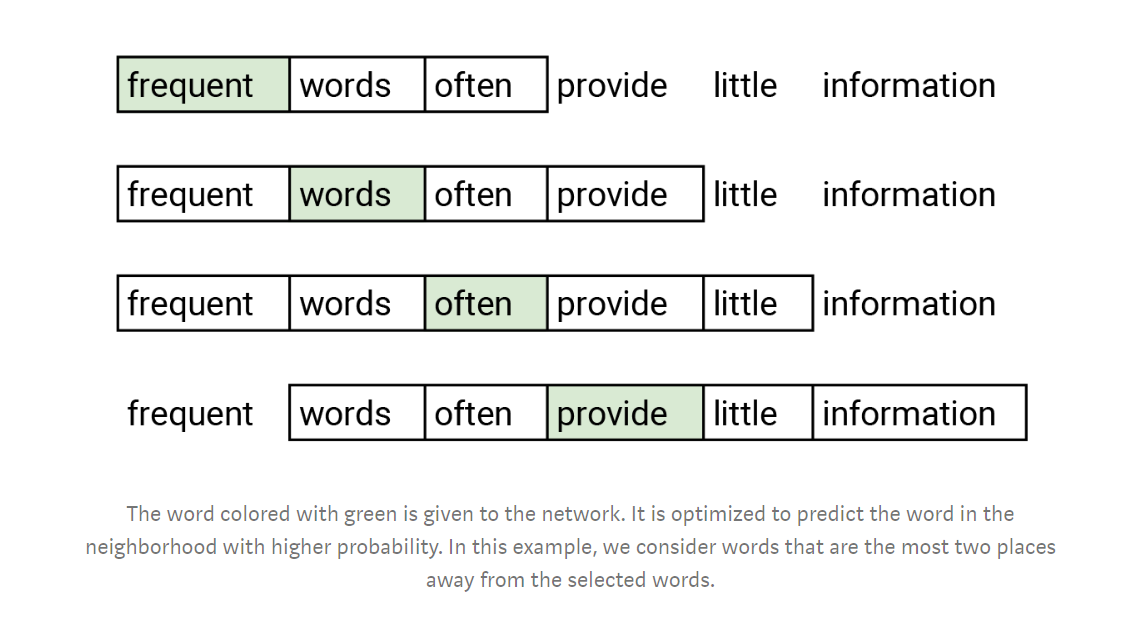
di ouput) riottenendo l’originale. Dopo l’addestramento, il livello di output (ovvero lo step di decompressione) può essere scartato mantenendo solo i livelli di input e di hidden.

Esistono due metodologie per poter creare il modello:

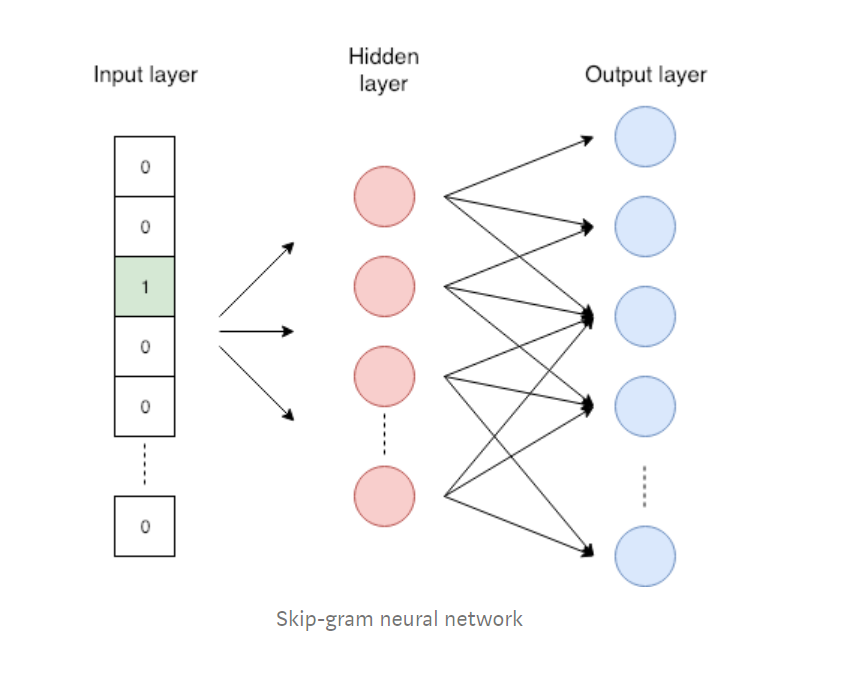
• **Skip-Gram model:** in cui si prendono coppie di parole dal testo (secondo un certo criterio che vedremo dopo) e si addestra una rete neurale con un livello di nodi di hidden sulla base del finto task in cui, a partire da una parola in input, la rete restituisce la distribuzione di probabilità delle 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

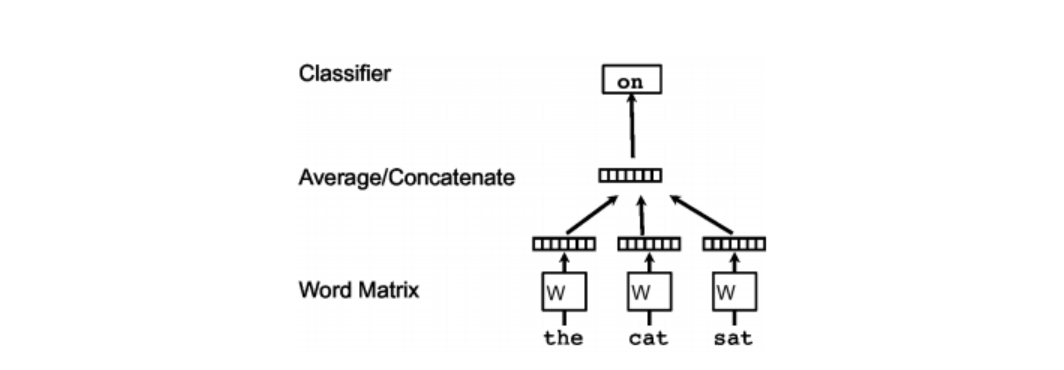
possiede 300 neuroni, la rete restituirà un vettore di dimensione pari a 300, per ciscuna parola.

• **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.



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.





**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.

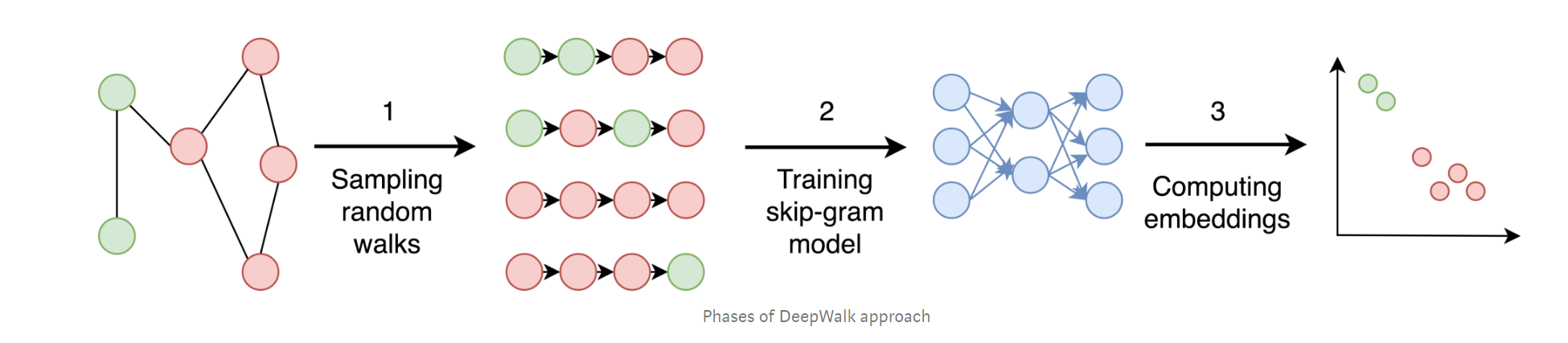
Per prima cosa ci si muove selezionando un nodo e ci si sposta su un vicino casuale dal nodo selezionato, in un determinato numero di passi.

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 camminati 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.



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.

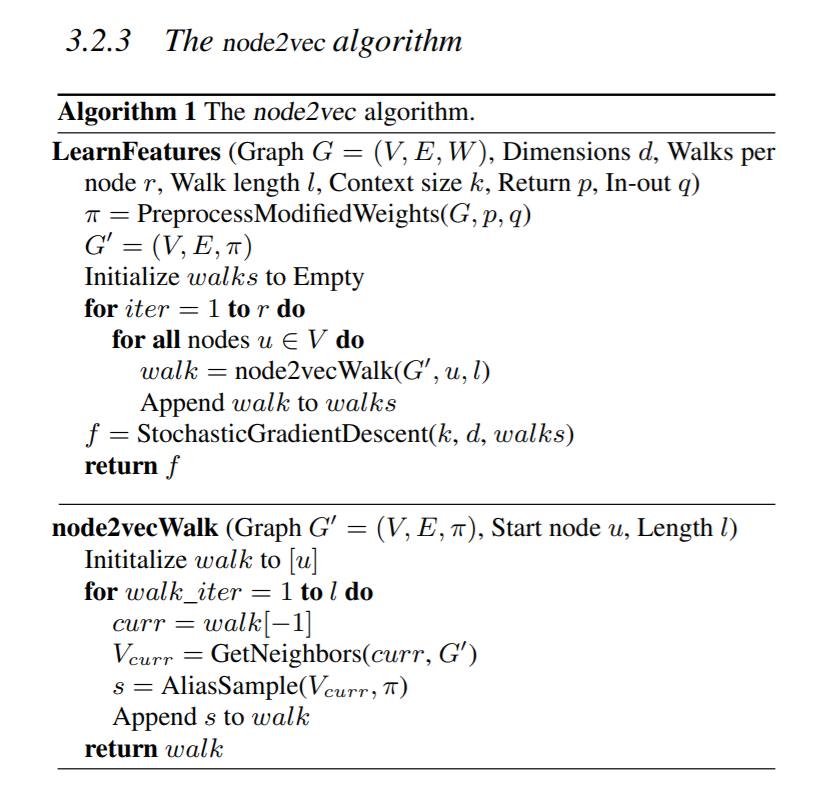
Per risolvere questo, entra in gioco il Node2Vec, un framework di apprendimento semi-supervisionato basato sulla discesa stocastica del gradiente, tipica degli algoritmi di machine learning.

Utilizzando la giusta rappresentazione dei nodi vicini in una rete, l’algoritmo riesce ad apprendere il modo di organizzare i vicini basato sulle regole della rete stessa; che avviene utilizzando un set di “passeggiate casuali” che esplorano in maniera molto efficiente i nodi vicini a quello in esame.

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.



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).

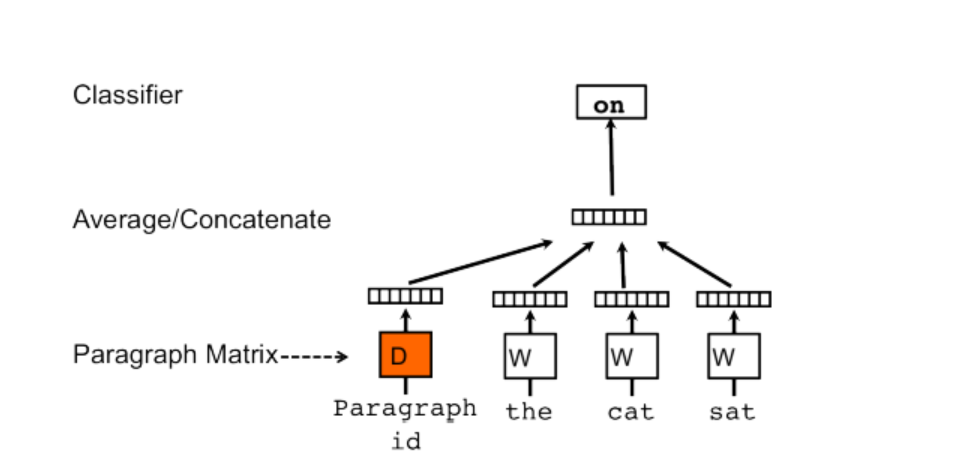
Anche nel caso del Doc2Vec esistono due metodologie analoghe rispettivamente al

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.

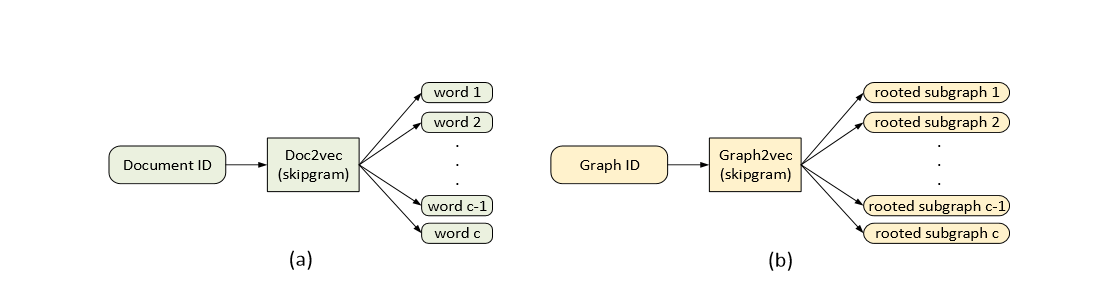


**Graph2vec**

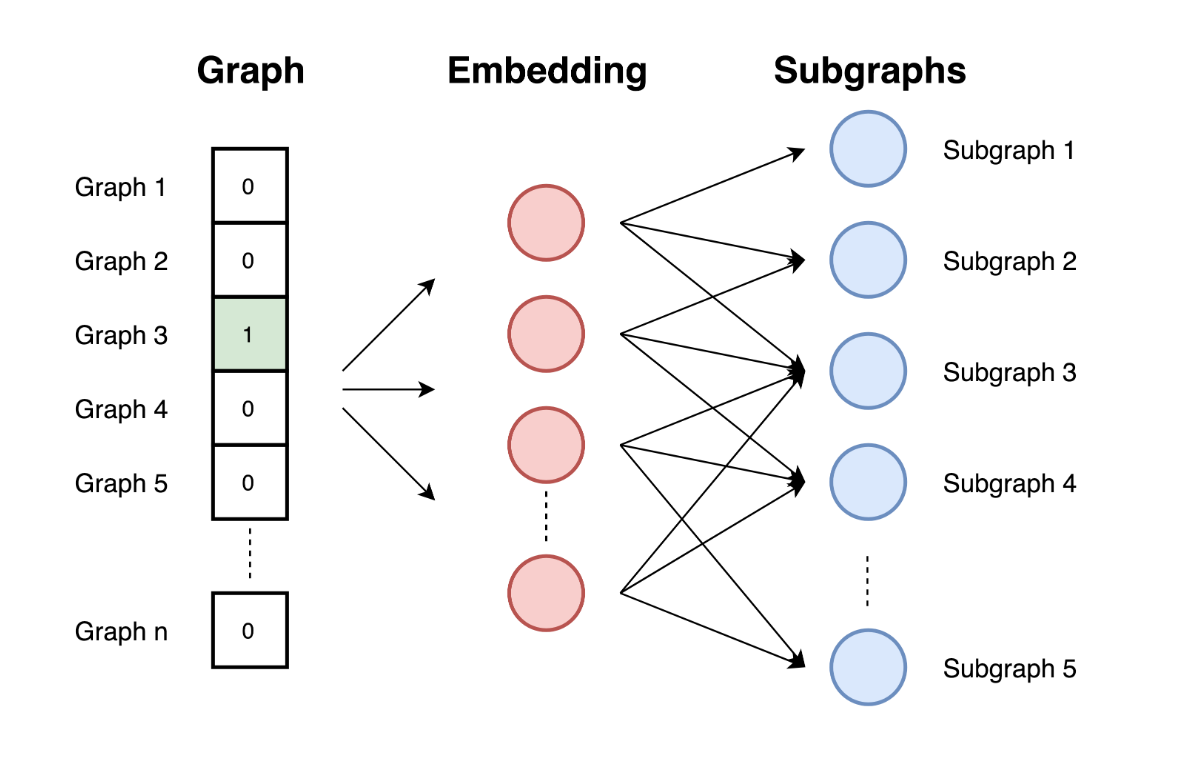
Graph2vec si basa su doc2vec che utilizza una rete di skip-gram. Ottiene un ID del doc sull'input ed è addestrato per massimizzare la probabilità di prevedere parole casuali tramite il doc stesso.

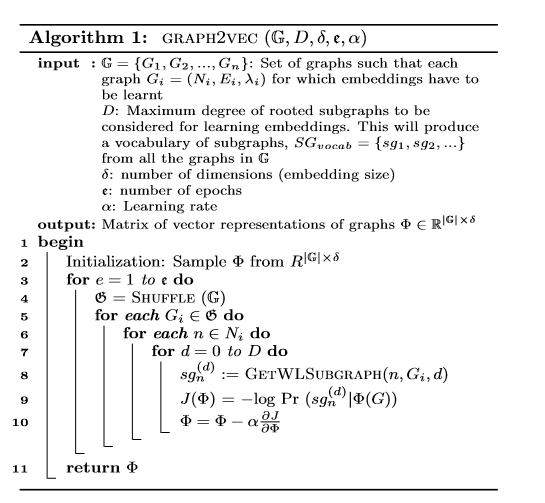
Gli approcci Graph2vec consistono in tre fasi:

* 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.





**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**

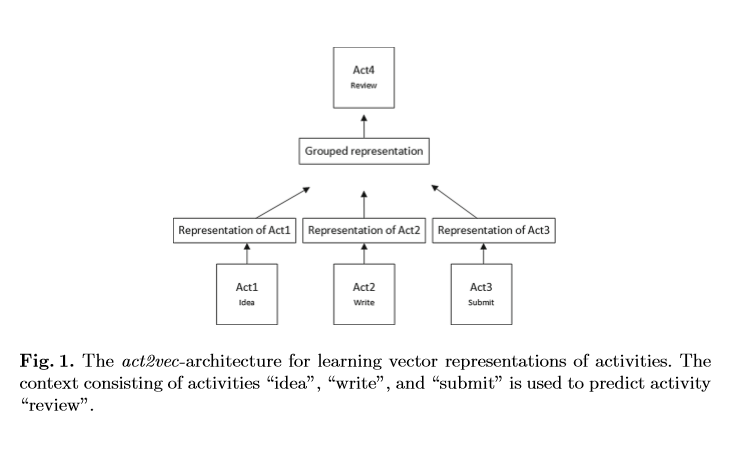
Act2vec è una architettura di apprendimento utilizzata per la rappresentazione delle attività.

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). 

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.

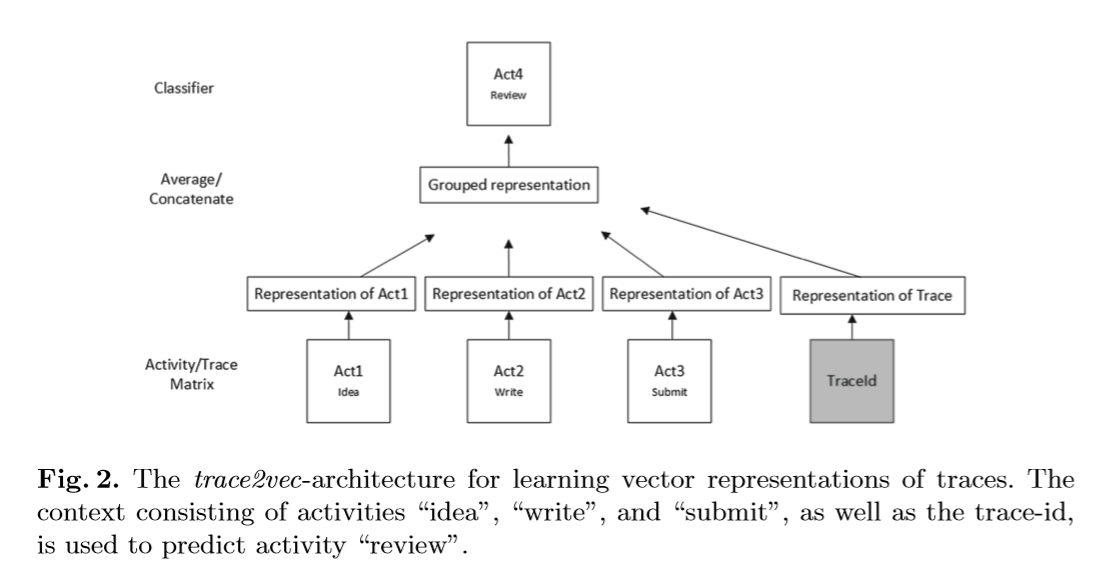
Nonostante questo, è stato dimostrato che architetture di incorporamento della rete neurale che trascurano l'ordine, come gli autoencoder, forniscono risultati solidi utilizzando l'iperparametrizzazione predefinita.

**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).



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.

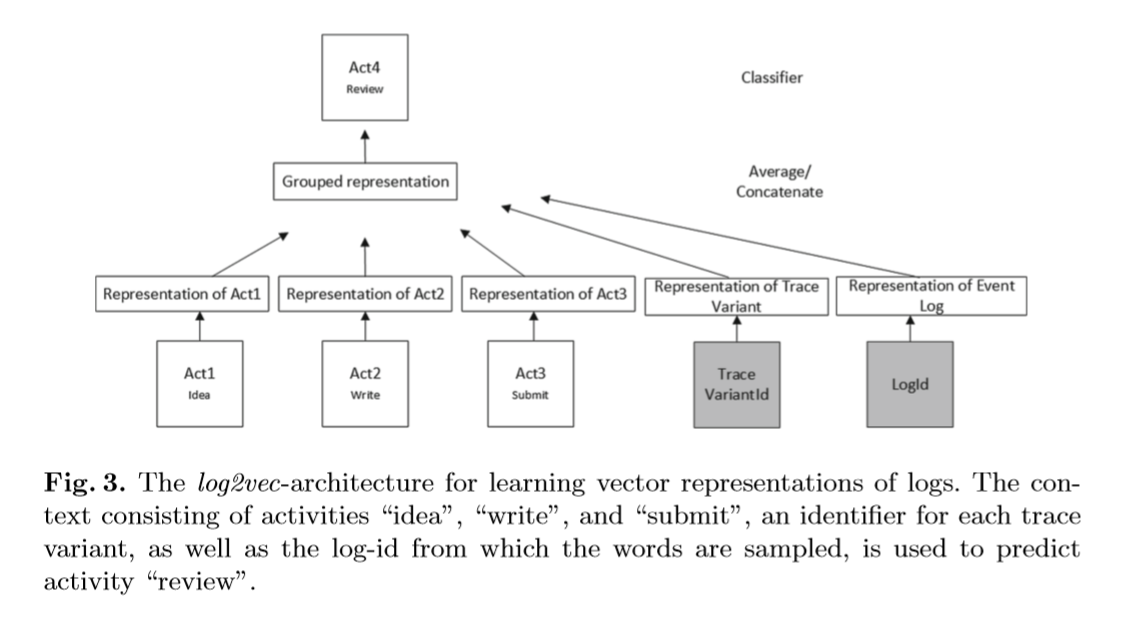
Si noti che, oltre ad utilizzare queste architetture basate su doc2vec, sarebbe anche possibile derivare rappresentazioni a livello di traccia facendo uso di architetture di aggregatori. Una tale architettura aggregatrice aggregato le rappresentazioni di attività.

**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.



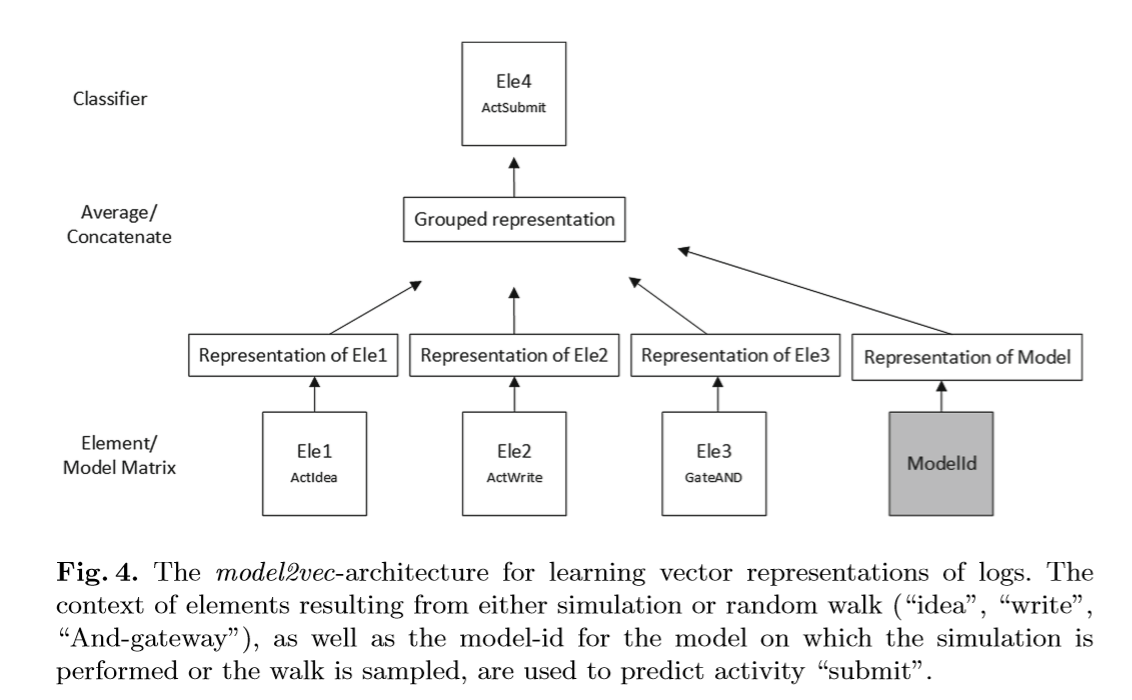
**Model2vec**

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).



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

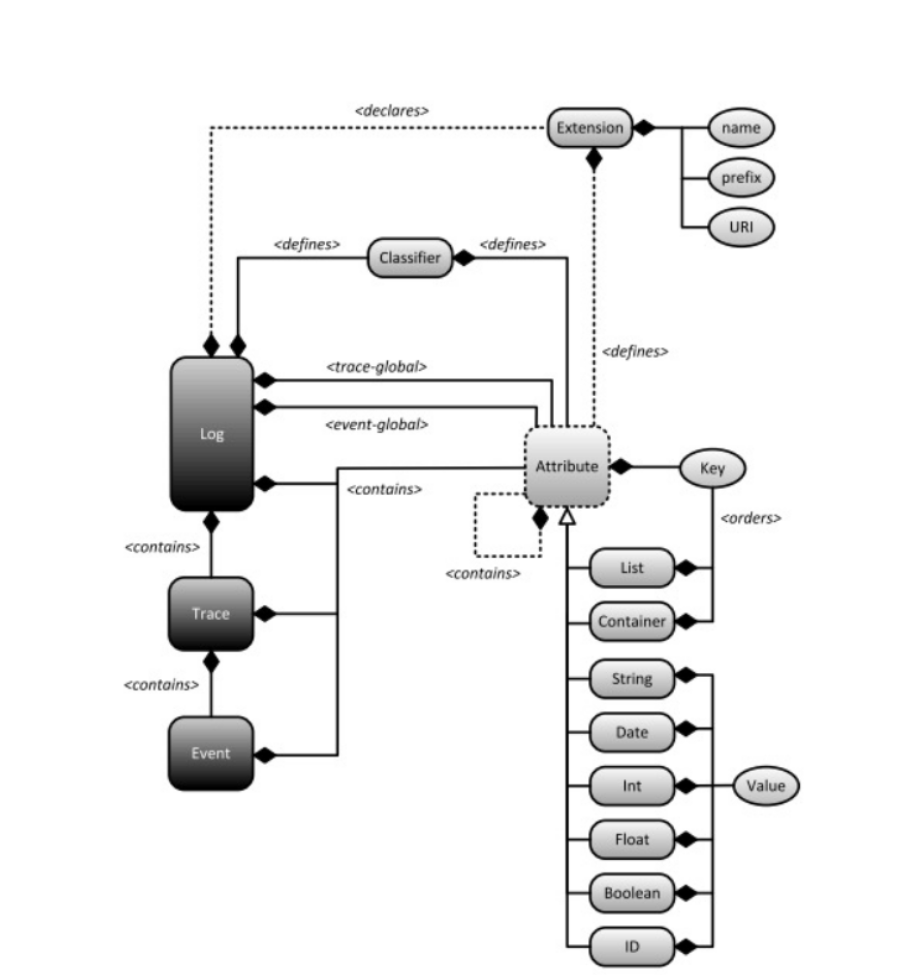
Il confronto tra modelli basati su model2vec potrebbe sfruttare la dimensione strutturale e grafica dei modelli stessi, con approcci esistenti fortemente incentrati sulla somiglianza comportamentale. Nel caso in cui questa dimensione strutturale / grafica dovesse essere de-enfatizzata, prevediamo che 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.

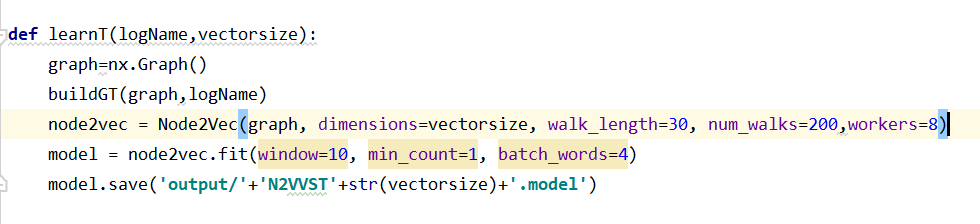
**Implementazione**

Verranno illustrate le parti salienti del codice prodotto.

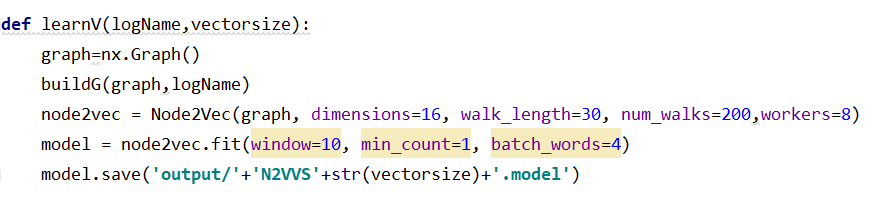
Le prime due classi, sviluppate in Python, che si occupano di costruire i modelli in esame, sono Node2Vec e Graph2Vec.

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 di 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.

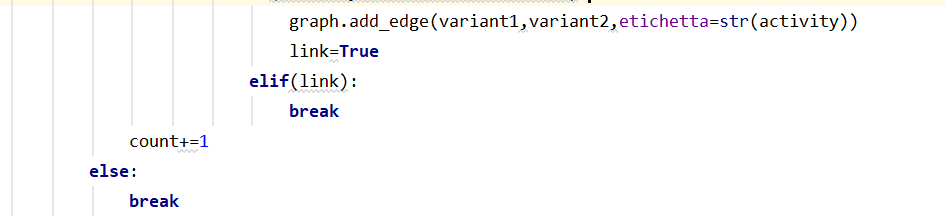


Il secondo modello prodotto segue la falsa riga del primo. I nodi, del grafo orientato, conterranno l’id variant della traccia in esame, i collegamenti tra i nodi sono sviluppati in base al valore dell’attività svolta, con attributi relativi alla risorsa utilizzata e al timestamp relativo all’attività, 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.



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.





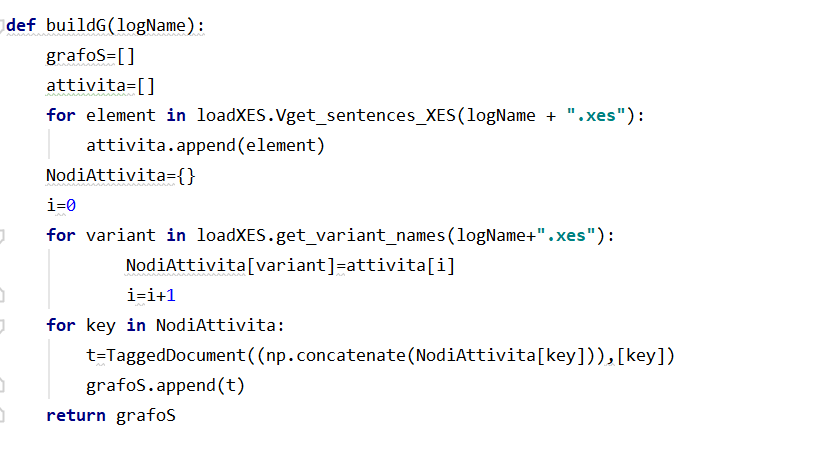
I modelli prodotti saranno “allenati”, sulla base dei grafi ottenuti seguendo i ragionamenti precedentemente elencati, utilizzando il metodo fit di Node2vec e il prodotto ottenuto.

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 costituiere 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.



“L’allenamento” del grafo è fatto sfruttando il metodo train, implementato nella classe Doc2Vec.



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.

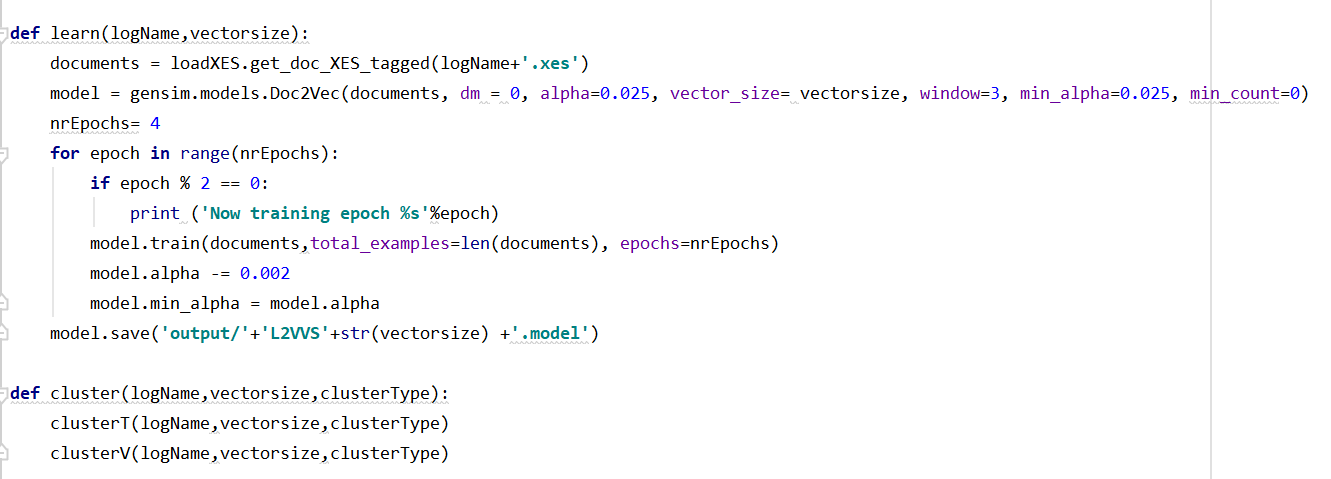
Per il cluster sui modelli ottenuti, si utilizzano gli algoritmi KMeans e Hierward. Si mostra, a titolo di esempio, il cluster relativo al variant-id, molto simile a quello prodotto dal cluster sulle tracce. Per le due implementazioni complete, si rimanda al codice.



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.



I test effettuati, insieme ad altri metodi utili, sono implementati nella classe Utility.