Reproducción de Node2Vec desde cero

Caminatas sesgadas + Alias Sampling + Skip–Gram (PyTorch)

13 de mayo de 2025

Contents

1 Introducción **2**

2 Carga del grafo y parámetros globales **2**

3 Alias Sampling y caminatas sesgadas **2**
- 3.1 Alias tables . . . 2
- 3.2 Caminata sesgada (parámetros p, q) . . . 3

4 Generación del corpus de caminatas **3**

5 Modelo Skip–Gram en PyTorch **3**
- 5.1 Dataset . . . 3
- 5.2 SGNS . . . 4
- 5.3 Negative sampling . . . 4
- 5.4 Entrenamiento . . . 4

6 Visualización y análisis **4**

7 Glosario **4**

8 Conclusiones **5**

1 Introducción

Node2Vec [1] es un algoritmo para aprender *embeddings* de nodos en grafos mediante dos ideas clave:

(1) Generar **caminatas aleatorias de segundo orden** con sesgo controlado por los parámetros p y q.

(2) Tratar cada caminata como una "oración" y entrenar un modelo Skip–Gram con Negative Sampling (SGNS), análogo a Word2Vec.

El reto exige reproducir la implementación **sin utilizar** librerías que encapsulen el algoritmo (p. ej. `node2vec` o `gensim`). Para ello desarrollamos:

- `Node2VecWalker`: caminatas + alias sampling (§3).
- `torch_skipgram.py`: SGNS minimalista en PyTorch (§5).

Trabajamos sobre el grafo *Les Misérables* incluido en `networkx`; los nodos son cadenas ("Valjean", "Woman2", ...).

2 Carga del grafo y parámetros globales

Listing 1: Cargar grafo y semilla global

```
import networkx as nx, numpy as np, random
SEED = 42
random.seed(SEED); np.random.seed(SEED)
G = nx.les_miserables_graph()
```

Propósito. Disponibilizar el objeto `G` y fijar semillas para reproducibilidad.

3 Alias Sampling y caminatas sesgadas

3.1 Alias tables

El algoritmo de alias (Vose, 1991) permite muestrear de una distribución categorical $\mathcal{P}(x_i)$ en $O(1)$ tiempo y $O(k)$ memoria.

Listing 2: Construcción y muestreo alias

```
def alias_setup(probs):
    K = len(probs); q = np.zeros(K); J = np.zeros(K, dtype=np.int32)
    smaller, larger = [], []
    for idx,p in enumerate(probs):
        q[idx] = K*p; (smaller if q[idx]<1 else larger).append(idx)
    while smaller and larger:
        s, l = smaller.pop(), larger.pop()
        J[s] = l; q[l] -= (1-q[s])
        (smaller if q[l]<1 else larger).append(l)
    return J, q
def alias_draw(J,q):
    K = len(J); kk = int(np.floor(np.random.rand()*K))
    return kk if np.random.rand() < q[kk] else J[kk]
```

3.2 Caminata sesgada (parámetros p, q)

p **(*return*)** Alta $p \Rightarrow$ se evita retroceder.

q **(*in–out*)** $q < 1$ favorece DFS (exploración lejana), $q > 1$ favorece BFS (vecindad local).

Listing 3: Fragmento clave del walker

```
if dst_nbr == src: # distancia 0
    bias = 1/p
elif G.has_edge(dst_nbr, src): # distancia 1
    bias = 1
else: # distancia 2
    bias = 1/q
probs.append(weight * bias)
```

Detalle del motor de caminatas. El método `simulate_walks()` ejecuta múltiples caminatas por nodo. Cada caminata es construida por `_walk()`, que decide el siguiente nodo según las distribuciones de transición preprocesadas con `alias_setup()`. El primer paso se basa en las probabilidades de vecinos inmediatos (`alias_nodes`), mientras que los pasos siguientes usan `alias_edges`, que implementan la caminata sesgada de segundo orden según los parámetros p y q definidos en el artículo. El método `_get_edge_alias()` computa las distribuciones $\alpha_{pq}(t, x)$ descritas en las ecuaciones (2) y (3) del paper, asignando un sesgo de $1/p$, 1 o $1/q$ según la topología local del grafo. Finalmente, `_preprocess_transition_probs()` calcula y almacena todas estas distribuciones una sola vez, lo que permite que cada paso de caminata se ejecute en tiempo constante $O(1)$ durante la simulación.

4 Generación del corpus de caminatas

Listing 4: Hiperparámetros de caminata

```
WALK_LENGTH = 30
NUM_WALKS = 200
P, Q = 1, 0.5
walker = Node2VecWalker(G, p=P, q=Q)
walks = walker.simulate_walks(NUM_WALKS, WALK_LENGTH)
```

Cada `walk` es una lista `['Valjean', 'Fantine', ...]`. Este corpus sustituye al texto en Word2Vec.

5 Modelo Skip–Gram en PyTorch

5.1 Dataset

Genera todos los pares (*target*, *contexto*) dentro de una ventana simétrica de tamaño w:

```
class SkipGramDataset(Dataset):
    def __init__(self, walks, window, node2idx):
        ...
```

5.2 SGNS

Dos tablas `nn.Embedding` (*input* y *output*). La pérdida es:

$$\mathcal{L} = -\log \sigma(\mathbf{v_t} \cdot \mathbf{v_c}) - \sum_{i=1}^{k} \log \sigma(-\mathbf{v_t} \cdot \mathbf{v_{n_i}})$$

```
class SGNS(nn.Module):
    def forward(self, targets, contexts, negatives):
        ...
        return -(loss_pos + loss_neg).mean()
```

5.3 Negative sampling

Tabla de *unigram* con probabilidad $P(i) \propto f_i^{3/4}$ ([2]) y muestreo multinomial.

5.4 Entrenamiento

Listing 5: Llamada de alto nivel

```
emb_dict = train_skipgram(
    walks, node_list=list(G.nodes()),
    emb_dim=64, window=10, neg_samples=5,
    epochs=1, lr=0.025,
    device='cuda' if torch.cuda.is_available() else 'cpu'
)
```

Devuelve `{node: vector}`.

6 Visualización y análisis

Listing 6: PCA 2D + vecinos similares

```
vectors = np.array([emb_dict[n] for n in G.nodes()])
pca = PCA(n_components=2).fit_transform(vectors)
...
def most_similar(node, k=5): ...
```

Observamos que *Valjean* se agrupa con *Marius*, *Cosette*, etc., reflejando homofilia en la trama.

7 Glosario

Embedding
: Vector continuo que codifica características latentes de un nodo.

Caminata aleatoria
: Secuencia de nodos visitados una arista a la vez.

Alias sampling
: Método $O(1)$ para muestrear distribuciones categóricas.

Skip–Gram
: Red superficial que aprende a predecir contexto de una entidad.

Negative sampling
: Sustituye la softmax completa por un número fijo de ejemplos negativos.

p
: Parámetro que penaliza el retroceso inmediato.

q
: Parámetro que controla la exploración lejana vs. local.

8 Conclusiones

Implementamos Node2Vec *end-to-end* sin librerías que lo abstraigan:

1. Caminatas sesgadas y tablas alias manuales.
2. Modelo SGNS propio en PyTorch.
3. Resultados coherentes con el paper original: nodos en la misma comunidad aparecen cercanos en el espacio PCA, y los vectores son útiles para tareas de similitud.

El código es reproducible (`SEED=42`) y funciona tanto en CPU como en GPU.

References

[1] Aditya Grover and Jure Leskovec. node2vec: scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD*, 2016.

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, 2013.