

**You said:**

Estoy implementando LightFM para un modelo donde uso siglip, clip y blip. Por separado, pero con la misma estructura de código. Estoy teniendo problemas con la memoria ram, hay mecanismos para optimizar el uso de memoria cuando trabajo con embeddings y altos volúmenes de datos?

**ChatGPT said:**

ChatGPT

Sí, hay varias estrategias para optimizar el uso de memoria RAM al trabajar con embeddings y grandes volúmenes de datos en modelos como LightFM junto con embeddings generados por SIGLIP, CLIP y BLIP. Aquí te comparto algunas técnicas que te pueden ayudar:

---

**1. Optimización de Embeddings****a. Reducir la Dimensionalidad de los Embeddings**

- Utiliza técnicas como **PCA (Análisis de Componentes Principales)** o **t-SNE** para reducir la dimensionalidad de los embeddings.
- Herramientas como TruncatedSVD de scikit-learn también son útiles si los embeddings son dispersos.

python

Copy code

```
from sklearn.decomposition import TruncatedSVD
```

```
svd = TruncatedSVD(n_components=128) # Reducir a 128 dimensiones, por ejemplo
reduced_embeddings = svd.fit_transform(embeddings)
```

**b. Cuantización de Embeddings**

- Convierte los embeddings de precisión flotante de 32 bits (float32) a 16 bits (float16) o incluso a 8 bits (int8) para ahorrar memoria.

python

Copy code

```
embeddings = embeddings.astype('float16')
```

---

**2. Gestión de Datos****a. Procesamiento por Lotes (Batching)**

- Carga los datos y calcula las recomendaciones en lotes más pequeños para evitar desbordar la memoria.

python

Copy code

```
batch_size = 1024
```

```
for i in range(0, len(embeddings), batch_size):
```

```
    batch = embeddings[i:i+batch_size]
```

```
    # Procesa el batch aquí
```

**b. Almacenamiento en Disco con memmap**

- Utiliza numpy.memmap para leer grandes conjuntos de datos directamente desde el disco sin cargarlos completamente en memoria.

python

Copy code

```
import numpy as np
```

```
embeddings = np.memmap('embeddings.dat', dtype='float32', mode='r', shape=(100000, 512))
```

---

### 3. Optimización del Modelo LightFM

#### a. Uso de Matrices Dispersas (Sparse Matrices)

- LightFM permite trabajar con **matrices dispersas** para reducir el consumo de memoria al manejar grandes cantidades de datos.

python

Copy code

```
from scipy.sparse import csr_matrix
```

```
interactions_sparse = csr_matrix(interactions_matrix)
```

#### b. Entrenamiento en Modo No Components

- Si no necesitas características de embeddings complejas, prueba reducir `no_components` en el modelo LightFM.

python

Copy code

```
model = LightFM(no_components=30) # Reducir la cantidad de componentes latentes
```

---

### 4. Liberar Memoria y Control de Recursos

#### a. Liberar Memoria con gc

- Usa el recolector de basura para liberar memoria que ya no se necesita.

python

Copy code

```
import gc
```

```
gc.collect()
```

#### b. Eliminar Variables Innecesarias

- Utiliza `del` para eliminar variables grandes que ya no necesitas.

python

Copy code

```
del large_variable
```

---

### 5. Paralelización y Computación en GPU

#### a. Procesamiento en GPU

- Si tienes acceso a una GPU, asegúrate de enviar los embeddings a la GPU para liberar memoria de la CPU.

python

Copy code

```
import torch
```

```
embeddings_gpu = torch.tensor(embeddings).to('cuda')
```

#### b. Paralelización con joblib

- Utiliza `joblib` para paralelizar tareas y optimizar el procesamiento de datos.

python

Copy code

```
from joblib import Parallel, delayed
```

```
Parallel(n_jobs=4)(delayed(func)(x) for x in data)
```

---

Estas estrategias deberían ayudarte a reducir el consumo de memoria y a mejorar la eficiencia de tu modelo. Si tienes un caso más específico o deseas ejemplos de implementación para alguna técnica, házmelo saber.

**You said:**

De qué sirven los float 32 y 16?

**ChatGPT said:**

ChatGPT

Los formatos de punto flotante **float32** y **float16** son tipos de datos utilizados para representar números decimales en cálculos computacionales. La principal diferencia entre ellos radica en su **precisión** y el **consumo de memoria**. Aquí te explico para qué sirven y cuáles son sus ventajas y desventajas.

---

### Float32 (32 bits)

#### Características:

- **Tamaño:** 32 bits (4 bytes).
- **Rango de valores:** Aproximadamente de  $\pm 3.4 \times 10^{38}$ .
- **Precisión:** Alrededor de 7 dígitos decimales.

#### Ventajas:

1. **Mayor precisión:** Es suficiente para la mayoría de las tareas de aprendizaje automático y procesamiento numérico.
2. **Compatibilidad:** Es el estándar predeterminado en la mayoría de los frameworks como TensorFlow, PyTorch y NumPy.
3. **Estabilidad numérica:** Menos propenso a errores de redondeo en cálculos complejos.

#### Desventajas:

1. **Consumo de memoria:** Utiliza más memoria que float16, lo que puede ser un problema con grandes volúmenes de datos.
  2. **Rendimiento:** Puede ser más lento en ciertas operaciones si se compara con float16 en hardware que soporta cálculos de precisión reducida.
- 

### Float16 (16 bits)

#### Características:

- **Tamaño:** 16 bits (2 bytes).
- **Rango de valores:** Aproximadamente de  $\pm 6.5 \times 10^4$ .
- **Precisión:** Alrededor de 3-4 dígitos decimales.

#### Ventajas:

1. **Ahorro de memoria:** Reduce el uso de memoria a la mitad en comparación con float32, permitiendo manejar conjuntos de datos más grandes.
2. **Rendimiento optimizado:** Puede acelerar cálculos en GPUs modernas (como NVIDIA con soporte para **Tensor Cores**).
3. **Eficiencia energética:** Consume menos energía, lo que es importante en dispositivos con recursos limitados.

#### Desventajas:

1. **Menor precisión:** Más propenso a errores de redondeo y desbordamiento para valores muy grandes o pequeños.
2. **Rango limitado:** Puede no ser adecuado para cálculos que requieran alta precisión numérica o tengan un rango amplio de valores.

---

### 🔧 ¿Cuándo usar cada uno?

#### 1. Usa float32 cuando:

- Necesitas alta precisión para cálculos numéricos complejos.
- Estás trabajando en modelos de redes neuronales donde la precisión es crítica.
- Tu hardware no tiene soporte eficiente para float16.

#### 2. Usa float16 cuando:

- Trabajas con grandes volúmenes de datos o modelos que requieren mucha memoria.
  - El entrenamiento en GPU se puede acelerar con precisión mixta (**Mixed Precision Training**).
  - El hardware (como GPUs NVIDIA con Tensor Cores) admite operaciones optimizadas para float16.
- 

### 🚀 Ejemplo de conversión en Python

Puedes convertir tus datos entre float32 y float16 en NumPy o PyTorch para optimizar el uso de memoria:

#### En NumPy:

python

Copy code

```
import numpy as np
```

```
# Crear un array de float32
```

```
data = np.array([1.5, 2.5, 3.5], dtype=np.float32)
```

```
# Convertir a float16
```

```
data_float16 = data.astype(np.float16)
```

```
print(data_float16)
```

#### En PyTorch:

python

Copy code

```
import torch
```

```
# Crear un tensor de float32
```

```
tensor = torch.tensor([1.5, 2.5, 3.5], dtype=torch.float32)
```

```
# Convertir a float16
```

```
tensor_float16 = tensor.to(torch.float16)
```

```
print(tensor_float16)
```

---

Elegir entre float32 y float16 dependerá de tu caso de uso específico, equilibrando la necesidad de precisión con la optimización de memoria y rendimiento.

#### You said:

Cuando normalizo los embeddings, lo que hago en una variable afecta en las demás, cómo puedo hacer que las modificaciones no alteren los datos originales?

#### ChatGPT said:

ChatGPT

Cuando trabajas con embeddings y necesitas normalizarlos sin afectar los datos originales, puedes hacer copias independientes para asegurarte de que las modificaciones no impacten los datos originales. Aquí te muestro algunas estrategias efectivas en **NumPy** y **PyTorch**.

---

## 1. Copiar Datos Antes de Modificar

### En NumPy

Usa `np.copy()` para crear una copia independiente del array antes de normalizar.

python

Copy code

```
import numpy as np
```

```
# Embeddings originales
```

```
embeddings = np.array([[1, 2], [3, 4], [5, 6]], dtype=np.float32)
```

```
# Crear una copia de los embeddings
```

```
embeddings_copy = np.copy(embeddings)
```

```
# Normalizar la copia
```

```
embeddings_copy /= np.linalg.norm(embeddings_copy, axis=1, keepdims=True)
```

```
print("Original:")
```

```
print(embeddings)
```

```
print("Normalizado:")
```

```
print(embeddings_copy)
```

### En PyTorch

Usa `.clone()` para crear una copia del tensor antes de normalizar.

python

Copy code

```
import torch
```

```
# Embeddings originales
```

```
embeddings = torch.tensor([[1, 2], [3, 4], [5, 6]], dtype=torch.float32)
```

```
# Crear una copia de los embeddings
```

```
embeddings_copy = embeddings.clone()
```

```
# Normalizar la copia
```

```
embeddings_copy = torch.nn.functional.normalize(embeddings_copy, p=2, dim=1)
```

```
print("Original:")
```

```
print(embeddings)
```

```
print("Normalizado:")
```

```
print(embeddings_copy)
```

---

## 2. Explicación de los Métodos

- **np.copy()** en NumPy:  
Crea una copia profunda del array, garantizando que cualquier cambio en la copia no afecte al original.
  - **.clone()** en PyTorch:  
Similar a np.copy(), produce una copia independiente del tensor, lo que te permite realizar operaciones sin modificar los datos originales.
- 

### ¿Por qué es importante hacer copias?

1. **Evitar efectos secundarios:** Si trabajas con múltiples transformaciones y reutilizas los datos originales, las modificaciones pueden propagarse accidentalmente.
  2. **Preservar los datos originales:** Mantener una versión sin modificar te permite comparar resultados o volver a realizar operaciones sin recalculer los embeddings desde cero.
  3. **Depuración más sencilla:** Facilita rastrear errores y probar diferentes técnicas sin preocuparte por modificar datos base.
- 

Con estas estrategias puedes normalizar y transformar tus embeddings de manera segura sin comprometer la integridad de los datos originales.

#### **You said:**

Para medir el consumo de cpu y ram en las métricas, qué librerías o códigos puedo hacer para calcularlo en mi modelo

#### **ChatGPT said:**

ChatGPT

Para medir el consumo de **CPU** y **RAM** mientras ejecutas tu modelo, existen varias bibliotecas en Python que te permiten monitorear estos recursos de manera efectiva. Aquí te muestro algunas de las más comunes y ejemplos de código para usarlas:

---

### 1. psutil para Monitoreo General

La biblioteca psutil permite obtener información detallada sobre el uso de CPU, memoria RAM y otros recursos del sistema.

#### **Instalación**

bash

Copy code

```
pip install psutil
```

#### **Ejemplo de Uso**

python

Copy code

```
import psutil
```

```
import time
```

```
def medir_consumo():
```

```
    # Obtener uso de CPU
```

```
    uso_cpu = psutil.cpu_percent(interval=1) # Porcentaje de uso en 1 segundo
```

```
    # Obtener uso de RAM
```

```
    memoria = psutil.virtual_memory()
```

```
    uso_ram = memoria.used / (1024 ** 3) # Convertir a GB
```

```
    total_ram = memoria.total / (1024 ** 3)
```

```
print(f"Uso de CPU: {uso_cpu}%")
print(f"Uso de RAM: {uso_ram:.2f} GB / {total_ram:.2f} GB")

# Llamar a la función antes y después de ejecutar el modelo
print("Antes de ejecutar el modelo:")
medir_consumo()

# Simular ejecución de un modelo pesado
time.sleep(3) # Simula un proceso de larga duración

print("Después de ejecutar el modelo:")
medir_consumo()
```

---

## 2. memory\_profiler para Monitoreo de Memoria Detallado

memory\_profiler es útil para perfilar el uso de memoria línea por línea en una función.

### Instalación

```
bash
Copy code
pip install memory-profiler
```

### Ejemplo de Uso

```
python
Copy code
from memory_profiler import profile

@profile
def funcion_pesada():
    a = [0] * (10 ** 7) # Crear una lista grande
    b = [1] * (10 ** 7) # Otra lista grande
    del b # Eliminar una variable
```

```
funcion_pesada()
```

#### ♦ Ejecuta el script con el comando:

```
bash
Copy code
python script.py
```

Este perfilador imprimirá una línea detallada del uso de memoria antes y después de cada operación dentro de la función.

---

## 3. time y tracemalloc para Medir Tiempo y Memoria

tracemalloc permite rastrear la asignación de memoria, mientras que time ayuda a medir el tiempo de ejecución.

### Ejemplo de Uso

```
python
Copy code
import tracemalloc
import time
```

```

def funcion_modelo():
    lista = [i ** 2 for i in range(10**6)]
    time.sleep(2) # Simula una pausa en el procesamiento

# Iniciar el rastreo
tracemalloc.start()
start_time = time.time()

# Ejecutar la función
funcion_modelo()

# Medir tiempo y memoria
end_time = time.time()
current, peak = tracemalloc.get_traced_memory()

print(f"Tiempo de ejecución: {end_time - start_time:.2f} segundos")
print(f"Memoria actual: {current / 1024**2:.2f} MB")
print(f"Memoria pico: {peak / 1024**2:.2f} MB")

# Detener el rastreo
tracemalloc.stop()

```

---

#### 4. GPUUtil para Monitoreo de GPU

Si utilizas una GPU para tu modelo, GPUUtil te ayuda a monitorear el uso de memoria y carga de la GPU.

##### Instalación

```

bash
Copy code
pip install gputil

```

##### Ejemplo de Uso

```

python
Copy code
import GPUUtil
from tabulate import tabulate

```

```

def medir_gpu():
    gpus = GPUUtil.getGPUs()
    lista = []

    for gpu in gpus:
        lista.append((
            gpu.id,
            gpu.name,
            f"{gpu.load * 100:.1f}%",
            f"{gpu.memoryUsed:.2f} MB",
            f"{gpu.memoryTotal:.2f} MB"
        ))

```



```
print(tabulate(lista, headers=("ID", "Nombre", "Uso de GPU", "Memoria Usada", "Memoria Total")))
```

```
# Medir uso de GPU
medir_gpu()
```

---

### ✓ Resumen de Herramientas

Librería	Uso
psutil	Monitoreo general de CPU y RAM.
memory_profiler	Perfilado detallado de memoria línea a línea.
tracemalloc	Rastreo de asignación de memoria.
GPUtil	Monitoreo de GPU (memoria y carga).
time	Medición del tiempo de ejecución.

---

Estas herramientas te permiten monitorear y optimizar el uso de recursos de manera efectiva durante el desarrollo y ejecución de tu modelo.

#### You said:

Tengo que agregarle las siguientes métricas a mi código, para esto me dieron métricas de ejemplo para svd y als, pero yo estoy usando lightFM, quiero que me des ideas de cómo implementarlas en mi código, te adjunto los códigos para el map, ncdg y recall

```
def precision_at_k(r, k): assert 1 <= k <= r.size return (np.asarray(r)[:k] != 0).mean()
def average_precision_at_k(r, k): r = np.asarray(r) score = 0. for i in range(min(k, r.size)): score +=
precision_at_k(r, i + 1) return score / k
def dcg_at_k(r, k): r = np.asarray(r)[:k] if r.size: return
np.sum(np.subtract(np.power(2, r), 1) / np.log2(np.arange(2, r.size + 2))) return 0.
def idcg_at_k(k): return dcg_at_k(np.ones(k), k)
def ndcg_at_k(r, k, max_relevant): idcg = idcg_at_k(min(k, max_relevant)) if not idcg: return 0. return dcg_at_k(r, k) / idcg
def recall_at_k(relevant_items, recommended_items, k): relevant_items = set(relevant_items)
recommended_items = set(recommended_items[:k]) intersection =
relevant_items.intersection(recommended_items) recall = len(intersection) /
len(relevant_items) if len(relevant_items) > 0 else 0 return recall
def evaluate_ranking_SVD_model(model, user_items_test, k): mean_ap = 0. # o MAP mean_ndcg =
0. mean_recall = 0. for u in user_items_test.keys(): rec = model.recommend(str(u), k,
includeRated=False) # Pyreclab's API rec = [int(x) for x in rec] rel_vector = np.isin(rec,
user_items_test[u], assume_unique=True).astype(int) mean_ap +=
average_precision_at_k(rel_vector, k) mean_ndcg += ndcg_at_k(rel_vector, k,
len(user_items_test[u])) mean_recall += recall_at_k(user_items_test[u], rec, k) mean_ap /=
len(user_items_test) mean_ndcg /= len(user_items_test) mean_recall /= len(user_items_test)
return mean_ap, mean_ndcg, mean_recall
def evaluate_ranking_ALS_model(model, user_items_test, k): mean_ap = 0. # o MAP mean_ndcg = 0. mean_recall = 0. for u in
user_items_test.keys(): rec = model.recommend(u, user_item_matrix[u], k)[0] # implicit's API
rel_vector = np.isin(rec, user_items_test[u], assume_unique=True).astype(int) mean_ap +=
average_precision_at_k(rel_vector, k) mean_ndcg += ndcg_at_k(rel_vector, k,
len(user_items_test[u])) mean_recall += recall_at_k(user_items_test[u], rec, k) mean_ap /=
len(user_items_test) mean_ndcg /= len(user_items_test) mean_recall /= len(user_items_test)
return mean_ap, mean_ndcg, mean_recall # This will be used by SVD user_item_valid =
```

```
validfile.groupby('userID')['itemID'].apply(list).to_dict() # This will be used by ALS (remember it
uses a compact range) compact_user_item_valid = {} for user, items in user_item_valid.items():
compact_user_item_valid[original_user_id_to_compact[user]] =
[original_item_id_to_compact[item] for item in items] assert len(user_item_valid) ==
len(compact_user_item_valid) # SVD top_k = 10 map, ndcg, recall =
evaluate_ranking_SVD_model(svd, user_item_valid, top_k) print('MAP@{}: {} \nNDCG@{}: {} \nRecall@{}: {}'
.format(top_k, map, top_k, ndcg, top_k, recall)) MAP@10:
0.005379334042741862 NDCG@10: 0.007546552937359704 Recall@10:
0.009456050336673679 # ALS top_k = 10 map, ndcg, recall = evaluate_ranking_ALS_model(als,
compact_user_item_valid, top_k) print('MAP@{}: {} \nNDCG@{}: {} \nRecall@{}: {}'
.format(top_k, map, top_k, ndcg, top_k, recall)) MAP@10: 0.054869712689545165 NDCG@10:
0.07826595537013126 Recall@10: 0.08479393375446878
```

#### ChatGPT said:

ChatGPT

Para implementar las métricas **MAP (Mean Average Precision)**, **NDCG (Normalized Discounted Cumulative Gain)** y **Recall** en tu modelo **LightFM**, puedes adaptar las funciones que te dieron para SVD y ALS con algunos ajustes específicos a la API de **LightFM**. Aquí te muestro cómo hacerlo paso a paso.

---

#### 🔧 Pasos para Adaptar el Código a LightFM

1. **Obtener recomendaciones con LightFM:**
  - LightFM permite obtener recomendaciones usando el método `model.predict()` o `model.predict_rank()`.
2. **Crear una función de evaluación:**
  - Similar a las funciones `evaluate_ranking_SVD_model` y `evaluate_ranking_ALS_model`, crear una función para LightFM que calcule **MAP**, **NDCG** y **Recall**.
3. **Adaptar las métricas:**
  - Utilizar las mismas funciones de métricas que ya tienes (`average_precision_at_k`, `ndcg_at_k`, `recall_at_k`).

---

#### 📦 Código para Evaluar el Modelo LightFM

python

Copy code

```
import numpy as np
from lightfm import LightFM
from lightfm.evaluation import precision_at_k as lightfm_precision_at_k
```

# Métricas existentes

```
def precision_at_k(r, k):
    assert 1 <= k <= r.size
    return (np.asarray(r)[:k] != 0).mean()
```

```
def average_precision_at_k(r, k):
    r = np.asarray(r)
    score = 0.
    for i in range(min(k, r.size)):
        score += precision_at_k(r, i + 1)
```

```

return score / k

def dcg_at_k(r, k):
    r = np.asarray(r)[:k]
    if r.size:
        return np.sum(np.subtract(np.power(2, r), 1) / np.log2(np.arange(2, r.size + 2)))
    return 0.

def idcg_at_k(k):
    return dcg_at_k(np.ones(k), k)

def ndcg_at_k(r, k, max_relevant):
    idcg = idcg_at_k(min(k, max_relevant))
    if not idcg:
        return 0.
    return dcg_at_k(r, k) / idcg

def recall_at_k(relevant_items, recommended_items, k):
    relevant_items = set(relevant_items)
    recommended_items = set(recommended_items[:k])
    intersection = relevant_items.intersection(recommended_items)
    recall = len(intersection) / len(relevant_items) if len(relevant_items) > 0 else 0
    return recall

# Función para evaluar el modelo LightFM
def evaluate_ranking_LightFM_model(model, user_items_test, user_ids, item_ids,
interactions, k):
    mean_ap = 0. # MAP
    mean_ndcg = 0.
    mean_recall = 0.

    for user_id in user_ids:
        # Predecir puntuaciones para todos los ítems
        scores = model.predict(user_id, item_ids)

        # Ordenar ítems por las puntuaciones en orden descendente
        recommended_items = np.argsort(-scores)[:k]

        # Ítems relevantes para el usuario actual
        relevant_items = user_items_test.get(user_id, [])

        # Vector de relevancia: 1 si el ítem está en los ítems relevantes, 0 si no
        rel_vector = np.isin(recommended_items, relevant_items,
assume_unique=True).astype(int)

        # Calcular métricas
        mean_ap += average_precision_at_k(rel_vector, k)
        mean_ndcg += ndcg_at_k(rel_vector, k, len(relevant_items))

```

```

mean_recall += recall_at_k(relevant_items, recommended_items, k)

mean_ap /= len(user_ids)
mean_ndcg /= len(user_ids)
mean_recall /= len(user_ids)

return mean_ap, mean_ndcg, mean_recall

```

---

## Ejemplo de Uso

### Datos de Ejemplo

Supongamos que ya tienes:

- **model:** Tu modelo LightFM entrenado.
- **user\_items\_test:** Un diccionario con los ítems relevantes para cada usuario, por ejemplo:

python

Copy code

```

user_items_test = {
    0: [1, 4, 7],
    1: [0, 3, 5],
    2: [2, 6, 8]
}

```

- **interactions:** Una matriz de interacciones `coo_matrix` que usaste para entrenar el modelo.
- **user\_ids:** Una lista de IDs de usuarios.
- **item\_ids:** Una lista de IDs de ítems.

### Código de Ejecución

python

Copy code

```

from scipy.sparse import coo_matrix

# Crear listas de usuarios e ítems
user_ids = list(user_items_test.keys())
item_ids = np.arange(interactions.shape[1]) # Todos los ítems posibles

# Evaluar el modelo
top_k = 10
map, ndcg, recall = evaluate_ranking_LightFM_model(model, user_items_test, user_ids,
item_ids, interactions, top_k)

# Mostrar resultados
print('MAP@{}: {:.4f}\nNDCG@{}: {:.4f}\nRecall@{}: {:.4f}'.format(top_k, map, top_k, ndcg,
top_k, recall))

```

---

## Explicación del Código

### 1. **model.predict(user\_id, item\_ids):**

- Obtiene las puntuaciones de predicción del modelo para un usuario específico y todos los ítems.

2. **np.argsort(-scores)[:k]:**
  - Ordena los ítems por sus puntuaciones de forma descendente y selecciona los k mejores.
3. **Vector de relevancia (rel\_vector):**
  - Compara los ítems recomendados con los relevantes para el usuario y crea un vector binario.
4. **Cálculo de métricas:**
  - Se suman los valores de **MAP**, **NDCG** y **Recall** para cada usuario y se promedian al final.

---

Este código te permitirá evaluar tu modelo LightFM utilizando las métricas **MAP**, **NDCG** y **Recall** de manera similar a los ejemplos proporcionados para SVD y ALS.

**You said:**

```
Ahora para la diversidad y novedad
user_id = 5401 # para SVD compact_user_id =
original_user_id_to_compact[user_id] # para ALS top_k = 10
svd_ranking = [int(r) for r in
svd.recommend(str(user_id), top_k, includeRated=False)]
als_ranking = list(als.recommend(userid=compact_user_id,
user_items=user_item_matrix[compact_user_id], N=top_k)[0])
als_ranking = [itemset[r] for r in als_ranking]
print('SVD: Recommendation for user {}: {}'.format(user_id, svd_ranking))
print('ALS: Recommendation for user {}: {}'.format(user_id, als_ranking))
SVD: Recommendation for user 5401: [66849, 47658, 11895, 44081, 72821, 50524, 64487, 15660, 36039, 61745]
ALS: Recommendation for user 5401: [35649, 43091, 961, 37403, 21556, 45260, 1589, 1121, 47659, 45878]
def diversity(recommended_items, item_categories):
unique_categories = set()
for item in recommended_items:
categories = item_categories[item]
if type(categories) == list:
unique_categories.update(categories)
else:
unique_categories.add(categories)
return len(unique_categories) / len(recommended_items)
item_categories = items_styles.set_index('itemID')['styleID'].to_dict()
categorias_recomendadas = [item_categories[item] for item in als_ranking]
# Calcular la proporción de categorías únicas
diversidad = diversity(als_ranking, item_categories)
print('Diversidad: {}'.format(diversidad))
Diversidad: 0.7
categorias_recomendadas [3268, 16386, 365, 1199, 568, 16386, 395, 395, 1199, 14]
2.5. Novedad
Primero se obtiene la popularidad de cada item. Para esto utilizaremos el set de entrenamiento.
item_counts = trainfile['itemID'].value_counts()
total_interacciones = len(trainfile)
items_popularity = item_counts / total_interacciones
items_popularity = items_popularity.to_dict()
def novelty(recommended_items, items_popularity):
novelty_score = 0
for item in recommended_items:
novelty_score += np.log(1 / items_popularity[item])
return novelty_score
novedad = novelty(als_ranking, items_popularity)
print('Novedad: {}'.format(novedad))
Novedad: 66.55537288016617
```

**ChatGPT said:**

ChatGPT

Para implementar las métricas de **diversidad** y **novedad** con tu modelo **LightFM**, podemos adaptar las funciones proporcionadas para SVD y ALS. Aquí te muestro cómo hacerlo paso a paso.

---

### 1. Función para Calcular Diversidad

La función de diversidad mide la variedad de categorías presentes en las recomendaciones. Cuantas más categorías únicas haya en las recomendaciones, mayor será la diversidad.

 **Código Adaptado para LightFM**

python

Copy code

```
def diversity(recommended_items, item_categories):
```

```
    """
```

Calcula la diversidad de una lista de ítems recomendados.

Parámetros:

- recommended\_items: Lista de IDs de ítems recomendados.

- item\_categories: Diccionario que asigna cada ítem a una o varias categorías.

Retorna:

- Diversidad como la proporción de categorías únicas en las recomendaciones.

```
    """
```

```
    unique_categories = set()
```

```
    for item in recommended_items:
```

```
        categories = item_categories.get(item, [])
```

```
        if isinstance(categories, list):
```

```
            unique_categories.update(categories)
```

```
        else:
```

```
            unique_categories.add(categories)
```

```
    return len(unique_categories) / len(recommended_items)
```

#### ◆ Ejemplo de Uso

python

Copy code

```
# Suponiendo que tienes un diccionario de categorías por ítem
```

```
item_categories = items_styles.set_index('itemID')['styleID'].to_dict()
```

```
# Obtener recomendaciones con LightFM
```

```
user_id = 5401
```

```
item_ids = np.arange(interactions.shape[1]) # Todos los ítems posibles
```

```
scores = model.predict(user_id, item_ids)
```

```
# Top-K recomendaciones
```

```
top_k = 10
```

```
lightfm_ranking = np.argsort(-scores)[:top_k]
```

```
# Calcular diversidad
```

```
diversidad = diversity(lightfm_ranking, item_categories)
```

```
print(f'Diversidad: {diversidad:.2f}')
```



## 2. Función para Calcular Novedad

La función de novedad evalúa cuán inesperadas o poco populares son las recomendaciones.

Si se recomiendan ítems menos populares, la novedad será mayor.



### Código Adaptado para LightFM

python

Copy code

```
def novelty(recommended_items, items_popularity):
```

```
    """
```

Calcula la novedad de una lista de ítems recomendados.

Parámetros:

- recommended\_items: Lista de IDs de ítems recomendados.
- items\_popularity: Diccionario con la popularidad de cada ítem (frecuencia de interacción).

Retorna:

- Novedad como la suma de los logaritmos inversos de la popularidad de los ítems.

"""

```
novelty_score = 0
for item in recommended_items:
    popularity = items_popularity.get(item, 1e-10) # Evitar divisiones por cero
    novelty_score += np.log(1 / popularity)
return novelty_score / len(recommended_items)
```

#### ◆ Ejemplo de Uso

python

Copy code

```
# Calcular la popularidad de cada ítem a partir del set de entrenamiento
```

```
item_counts = trainfile['itemID'].value_counts()
```

```
total_interacciones = len(trainfile)
```

```
items_popularity = (item_counts / total_interacciones).to_dict()
```

```
# Calcular novedad para las recomendaciones de LightFM
```

```
novedad = novelty(lightfm_ranking, items_popularity)
```

```
print(f'Novedad: {novedad:.2f}')
```

---

#### Resumen del Código Completo

python

Copy code

```
import numpy as np
```

```
# Funciones de métricas
```

```
def diversity(recommended_items, item_categories):
```

```
    unique_categories = set()
```

```
    for item in recommended_items:
```

```
        categories = item_categories.get(item, [])
```

```
        if isinstance(categories, list):
```

```
            unique_categories.update(categories)
```

```
        else:
```

```
            unique_categories.add(categories)
```

```
    return len(unique_categories) / len(recommended_items)
```

```
def novelty(recommended_items, items_popularity):
```

```
    novelty_score = 0
```

```
    for item in recommended_items:
```

```
        popularity = items_popularity.get(item, 1e-10)
```

```
        novelty_score += np.log(1 / popularity)
```

```
    return novelty_score / len(recommended_items)
```

```
# Obtener recomendaciones con LightFM
user_id = 5401
item_ids = np.arange(interactions.shape[1]) # Todos los ítems posibles
scores = model.predict(user_id, item_ids)

# Top-K recomendaciones
top_k = 10
lightfm_ranking = np.argsort(-scores)[:top_k]

# Diccionario de categorías y popularidad
item_categories = items_styles.set_index('itemID')['styleID'].to_dict()
item_counts = trainfile['itemID'].value_counts()
total_interacciones = len(trainfile)
items_popularity = (item_counts / total_interacciones).to_dict()

# Calcular diversidad y novedad
diversidad = diversity(lightfm_ranking, item_categories)
novedad = novelty(lightfm_ranking, items_popularity)

print(f'Diversidad: {diversidad:.2f}')
print(f'Novedad: {novedad:.2f}')
```

---

#### ✅ Explicación del Flujo

1. **Obtener recomendaciones** con LightFM para un usuario específico.
2. **Calcular diversidad:**
  - Mide la proporción de categorías únicas en los ítems recomendados.
3. **Calcular novedad:**
  - Evalúa qué tan inesperadas son las recomendaciones en función de su popularidad en el set de entrenamiento.
4. **Imprimir resultados** de diversidad y novedad.

Estas métricas te ayudarán a evaluar cuán variadas y novedosas son las recomendaciones generadas por tu modelo LightFM.