```
!pip install -q cornac
import numpy as np
import pandas as pd
import cornac
from cornac.models import BPR
from cornac.data import Dataset
import warnings
import os
import torch
warnings.filterwarnings('ignore')
print("Setup complete. Cornac version:", cornac.__version__)
```

```
Setup complete. Cornac version: 2.3.5
```

```
def load_data_robust(input_file):
    print(f"Loading data from {input_file}...")
    user_items = []
    if not os.path.exists(input_file):
        print(f"ERROR: {input_file} not found. Please upload it to
Colab!")
        return []
    with open(input_file, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) >= 2:
                user_id = parts[0]
                items = parts[1:]
                for item_id in items:
                     user_items.append((user_id, item_id, 1.0))

    print(f"Loaded {len(user_items)} interactions")
    return user_items

input_file = 'train.txt'
user_items = load_data_robust(input_file)

train_dataset = Dataset.from_uir(user_items, seed=123)

print("--- Data Statistics ---")
print(f"Unique Users: {train_dataset.num_users}")
print(f"Unique Items: {train_dataset.num_items}")
print(f"Sparsity: {100 * (1 - len(user_items) /
(train_dataset.num_users * train_dataset.num_items)):.4f}%")
```

```
Loading data from train.txt...
Loaded 810128 interactions
--- Data Statistics ---
Unique Users: 29858
Unique Items: 40981
Sparsity: 99.9338%
```

```python
def verify_data_stats(input_file):
    print("--- 1. Checking Data Format (First 3 Lines) ---")
    with open(input_file, 'r') as f:
        for i, line in enumerate(f):
            if i < 3:
                print(f"Line {i}: {line.strip()[:50]}...")
                parts = line.strip().split()
                if len(parts) > 2:
                    print(f"User {parts[0]} has {len(parts)-1}
items.")
            else:
                break

    print("\n--- 2. Calculating Sparsity ---")
    user_items = []
    with open(input_file, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) >= 2:
                user_id = parts[0]
                items = parts[1:]
                for item_id in items:
                     user_items.append((user_id, item_id))

    unique_users = len(set(u for u, i in user_items))
    unique_items = len(set(i for u, i in user_items))
    total_interactions = len(user_items)

    total_cells = unique_users * unique_items
    density = total_interactions / total_cells
    sparsity = 1.0 - density

    print(f"Total Users: {unique_users}")
    print(f"Total Items: {unique_items}")
    print(f"Total Interactions: {total_interactions}")
    print(f"Matrix Size: {total_cells:,}")
    print(f"Sparsity: {sparsity * 100:.4f}%")

verify_data_stats('/content/train.txt')
```

```
--- 1. Checking Data Format (First 3 Lines) ---
Line 0: 0 13264 3556 8355 3557 17801 18458 18068 2978 2307...
User 0 has 9 items.
Line 1: 1 39068 39525 37826 38174 37402 24989 23304 27250 ...
User 1 has 9 items.
Line 2: 2 6990 19114 8874 23176 32264 31804 19666 31939...
User 2 has 8 items.

--- 2. Calculating Sparsity ---
Total Users: 29858
```

```
Total Items: 40981
Total Interactions: 810128
Matrix Size: 1,223,610,698
Sparsity: 99.9338%
```

```
print("\n--- Statistics ---")
print(f"Unique Users: {train_dataset.num_users}")
print(f"Unique Items: {train_dataset.num_items}")
print(f"Total Interactions: {len(user_items)}")
```

```
--- Statistics ---
Unique Users: 29858
Unique Items: 40981
Total Interactions: 810128
```

```
print("Starting Hyperparameter Tuning...")
param_grid = {
    'k': [50, 100, 150],
    'lr': [0.005, 0.01],
    'lambda': [0.001, 0.01]
}

best_score = -1
best_params = {'k': 100, 'lr': 0.01, 'lambda': 0.01}

eval_method = RatioSplit(
    data=user_items,
    test_size=0.1,
    rating_threshold=0.0,
    seed=123,
    exclude_unknowns=False,
    verbose=False
)
for k in param_grid['k']:
    for lr in param_grid['lr']:
        for lam in param_grid['lambda']:
            print(f"Testing k={k}, lr={lr}, lambda={lam} ... ",
end="")

            try:
                model = BPR(
                    k=k,
                    max_iter=100,
                    learning_rate=lr,
                    lambda_reg=lam,
                    verbose=False,
                    seed=123
                )
```

```
                exp = Experiment(
                    eval_method=eval_method,
                    models=[model],
                    metrics=[NDCG(k=20)],
                    verbose=False
                )
                exp.run()

                score = exp.result[0].metric_avg_results['NDCG@20']
                print(f"NDCG: {score:.4f}")

                if score > best_score:
                    best_score = score
                    best_params = {'k': k, 'lr': lr, 'lambda': lam}

            except Exception as e:
                print(f"Failed: {e}")
                continue

print(f"\n Best Parameters Found: {best_params} (NDCG:
{best_score:.4f})")
```

```
Starting Hyperparameter Tuning...
Testing k=50, lr=0.005, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0178 |   24.2215 |  68.0860

NDCG: 0.0178
Testing k=50, lr=0.005, lambda=0.01 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0195 |   21.8674 |  67.0224

NDCG: 0.0195
Testing k=50, lr=0.01, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0475 |   21.9359 |  65.8606

NDCG: 0.0475
Testing k=50, lr=0.01, lambda=0.01 ...
TEST:
...
```

```
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0505 |   22.3190 |  67.3333

NDCG: 0.0505
Testing k=100, lr=0.005, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0168 |   30.5799 |  58.9040

NDCG: 0.0168
Testing k=100, lr=0.005, lambda=0.01 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0173 |   31.5291 |  60.9132

NDCG: 0.0173
Testing k=100, lr=0.01, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0465 |   29.7425 |  59.9953

NDCG: 0.0465
Testing k=100, lr=0.01, lambda=0.01 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0493 |   31.4548 |  61.1394

NDCG: 0.0493
Testing k=150, lr=0.005, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0162 |   44.6626 |  90.4289

NDCG: 0.0162
Testing k=150, lr=0.005, lambda=0.01 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
```

```
BPR |  0.0173 |   45.4151 |  89.0198

NDCG: 0.0173
Testing k=150, lr=0.01, lambda=0.001 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0449 |   46.6235 |  89.8908

NDCG: 0.0449
Testing k=150, lr=0.01, lambda=0.01 ...
TEST:
...
    | NDCG@20 | Train (s) | Test (s)
--- + ------- + --------- + --------
BPR |  0.0483 |   47.2079 |  86.2953

NDCG: 0.0483

✓ Best Parameters Found: {'k': 50, 'lr': 0.01, 'lambda': 0.01} (NDCG:
0.0505)

print("Starting BPR Training")
models = []
configs = [
    {'k': 50, 'lr': 0.01,'lambda': 0.01 ,'seed': 42},
    {'k': 300, 'lr': 0.005,'lambda': 0.001 ,'seed': 123},
    {'k': 500, 'lr': 0.001,'lambda': 0.001 ,'seed': 999}
]

for config in configs:
    print(f"\nTraining Supercharged BPR (k={config['k']},
lr={config['lr']})...")

    bpr = BPR(
        k=config['k'],
        max_iter=2000,
        learning_rate=config['lr'],
        lambda_reg=config['lambda'],
        verbose=True,
        seed=config['seed']
    )

    bpr.fit(train_dataset)
    models.append(bpr)

print(f"\nMassive Ensemble Training Complete.")
```

```
Starting BPR Training

Training Supercharged BPR (k=50, lr=0.01)...

{"model_id":"7889eaa4a50c40cbb250993019b2a460","version_major":2,"vers
ion_minor":0}

Optimization finished!

Training Supercharged BPR (k=300, lr=0.005)...

{"model_id":"a6fd0e7769744748a0edb49f525a9702","version_major":2,"vers
ion_minor":0}

Optimization finished!

Training Supercharged BPR (k=500, lr=0.001)...

{"model_id":"9530a30dc7b54b2bb7dd4c4e3933dcf0","version_major":2,"vers
ion_minor":0}

Optimization finished!

Massive Ensemble Training Complete.

def calculate_metrics(ground_truth_file, submission_file, k=20):
    print(f"Evaluating '{submission_file}' against
'{ground_truth_file}'...")
    def load_dict(path):
        data = {}
        if not os.path.exists(path): return {}
        with open(path, 'r') as f:
            for line in f:
                parts = line.strip().split()
                if len(parts) >= 2:
                    data[parts[0]] = parts[1:]
        return data
    gt_dict = {u: set(items) for u, items in
load_dict(ground_truth_file).items()}
    pred_dict = load_dict(submission_file)
    if not gt_dict or not pred_dict:
        print("Error: Empty data files.")
        return
    ndcg_scores = []
    recall_scores = []
    for user, true_items in gt_dict.items():
        if user not in pred_dict:
            ndcg_scores.append(0); recall_scores.append(0)
            continue
        preds = pred_dict[user][:k]
        hits = [1 if item in true_items else 0 for item in preds]
```

```
        recall_scores.append(sum(hits) / len(true_items) if true_items
else 0)
        dcg = sum((2**h - 1) / np.log2(i + 2) for i, h in
enumerate(hits))
        ideal_len = min(len(true_items), k)
        idcg = sum((2**1 - 1) / np.log2(i + 2) for i in
range(ideal_len))
        ndcg_scores.append(dcg / idcg if idcg > 0 else 0)

    print(f"Sanity Check Results")
    print(f"NDCG@{k}:   {np.mean(ndcg_scores):.5f}")
    print(f"Recall@{k}: {np.mean(recall_scores):.5f}")

def generate_ensemble_predictions_gpu_eval(models, dataset, top_k=20,
batch_size=1000, mask_seen=True):
    print(f"Generating Hybrid RRF predictions (Masking Seen Items:
{mask_seen})...")
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    print(f"Using device: {device}")
    num_users = dataset.num_users
    num_items = dataset.num_items
    k_rrf = 60

    print("Calculating Global Popularity Ranks...")
    item_freqs = np.array(dataset.csr_matrix.sum(axis=0)).flatten()
    pop_indices = np.argsort(item_freqs)[::-1][:200]
    pop_rrf_vector = torch.zeros(num_items, device=device)
    pop_ranks = torch.arange(200, device=device).float()
    pop_weights = 1.0 / (pop_ranks + k_rrf)
    pop_indices_tensor = torch.tensor(pop_indices.copy(),
device=device)
    pop_rrf_vector.scatter_add_(0, pop_indices_tensor, pop_weights)
    print("Popularity vector ready.")

    gpu_models = []
    for model in models:
        gpu_model = {
            'U': torch.tensor(model.u_factors, dtype=torch.float32,
device=device),
            'V': torch.tensor(model.i_factors, dtype=torch.float32,
device=device),
            'B': torch.tensor(model.i_biases, dtype=torch.float32,
device=device)
        }
        gpu_models.append(gpu_model)

    user_preds = {}
    for start_idx in range(0, num_users, batch_size):
        end_idx = min(start_idx + batch_size, num_users)
```

```
        current_batch_size = end_idx - start_idx

        if start_idx % 2000 == 0:
            print(f"Processed {start_idx}/{num_users} users")

        batch_rrf_scores = pop_rrf_vector.expand(current_batch_size, -
1).clone()

        for model_data in gpu_models:
            U_batch = model_data['U'][start_idx:end_idx]
            V = model_data['V']
            B = model_data['B']
            batch_scores = torch.matmul(U_batch, V.t()) + B
            if mask_seen:
                for i in range(current_batch_size):
                    global_user_idx = start_idx + i
                    start_ptr =
dataset.csr_matrix.indptr[global_user_idx]
                    end_ptr =
dataset.csr_matrix.indptr[global_user_idx + 1]
                    seen_indices =
dataset.csr_matrix.indices[start_ptr:end_ptr]
                    batch_scores[i, seen_indices] = -float('inf')
            _, top_200_indices = torch.topk(batch_scores, k=200,
dim=1)
            ranks = torch.arange(200, device=device).float()
            rrf_weights = 1.0 / (ranks + k_rrf)
            batch_rrf_weights = rrf_weights.expand(current_batch_size,
-1)
            batch_rrf_scores.scatter_add_(1, top_200_indices,
batch_rrf_weights)
        if mask_seen:
            for i in range(current_batch_size):
                global_user_idx = start_idx + i
                start_ptr = dataset.csr_matrix.indptr[global_user_idx]
                end_ptr = dataset.csr_matrix.indptr[global_user_idx +
1]
                seen_indices =
dataset.csr_matrix.indices[start_ptr:end_ptr]
                batch_rrf_scores[i, seen_indices] = -float('inf')
        _, final_top_k_indices = torch.topk(batch_rrf_scores, k=top_k,
dim=1)
        final_indices_cpu = final_top_k_indices.cpu().numpy()
        for i in range(current_batch_size):
            user_id = dataset.user_ids[start_idx + i]
            top_items = [dataset.item_ids[idx] for idx in
final_indices_cpu[i]]
            user_preds[user_id] = top_items

    return user_preds
```

```
recs_eval = generate_ensemble_predictions_gpu_eval(
    models,
    train_dataset,
    top_k=20,
    mask_seen=False
)
```

```
Generating Hybrid RRF predictions (Masking Seen Items: False)...
Using device: cpu
Calculating Global Popularity Ranks...
Popularity vector ready.
Processed 0/29858 users
Processed 2000/29858 users
Processed 4000/29858 users
Processed 6000/29858 users
Processed 8000/29858 users
Processed 10000/29858 users
Processed 12000/29858 users
Processed 14000/29858 users
Processed 16000/29858 users
Processed 18000/29858 users
Processed 20000/29858 users
Processed 22000/29858 users
Processed 24000/29858 users
Processed 26000/29858 users
Processed 28000/29858 users
```

```
check_file = 'train_sanity_check.txt'
with open(check_file, 'w') as f:
    for user_id, items in recs_eval.items():
        line = f"{user_id} {' '.join(items)}\n"
        f.write(line)

calculate_metrics(ground_truth_file='train.txt',
submission_file=check_file, k=20)
```

```
Evaluating 'train_sanity_check.txt' against 'train.txt'...
Sanity Check Results
NDCG@20:   0.22173
Recall@20: 0.15987
```

```
import numpy as np
import os

def calculate_metrics(ground_truth_file, submission_file, k=20):
    print(f"Evaluating '{submission_file}' against
'{ground_truth_file}'...")

    # 1. Helper to load data
    def load_dict(path):
```

```
        data = {}
        if not os.path.exists(path): return {}
        with open(path, 'r') as f:
            for line in f:
                parts = line.strip().split()
                if len(parts) >= 2:
                    data[parts[0]] = parts[1:]
        return data

    gt_dict = {u: set(items) for u, items in
load_dict(ground_truth_file).items()}
    pred_dict = load_dict(submission_file)

    if not gt_dict or not pred_dict:
        print("Error: Empty data files.")
        return

    ndcg_scores = []
    recall_scores = []
    map_scores = []

    for user, true_items in gt_dict.items():
        if user not in pred_dict:
            ndcg_scores.append(0)
            recall_scores.append(0)
            map_scores.append(0)
            continue

        preds = pred_dict[user][:k]
        hits = [1 if item in true_items else 0 for item in preds]
        recall_scores.append(sum(hits) / len(true_items) if true_items
else 0)
        dcg = sum((2**h - 1) / np.log2(i + 2) for i, h in
enumerate(hits))
        ideal_len = min(len(true_items), k)
        idcg = sum((2**1 - 1) / np.log2(i + 2) for i in
range(ideal_len))
        ndcg_scores.append(dcg / idcg if idcg > 0 else 0)
        score_sum = 0.0
        num_hits = 0.0
        for i, h in enumerate(hits):
            if h == 1:
                num_hits += 1
                precision_at_i = num_hits / (i + 1)
                score_sum += precision_at_i
        denominator = min(len(true_items), k)
        ap = score_sum / denominator if denominator > 0 else 0
        map_scores.append(ap)
    print("-" * 40)
    print(f"EVALUATION RESULTS (k={k})")
```

```
    print(f"NDCG:   {np.mean(ndcg_scores):.5f}")
    print(f"Recall: {np.mean(recall_scores):.5f}")
    print(f"MAP:    {np.mean(map_scores):.5f}")
    print("-" * 40)
check_file = 'train_sanity_check.txt'
with open(check_file, 'w') as f:
    for user_id, items in recs_eval.items():
        line = f"{user_id} {' '.join(items)}\n"
        f.write(line)
calculate_metrics(ground_truth_file='train.txt',
submission_file=check_file, k=20)
```

```
Evaluating 'train_sanity_check.txt' against 'train.txt'...
----------------------------------------
EVALUATION RESULTS (k=20)
NDCG:   0.22173
Recall: 0.15987
MAP:    0.09809
----------------------------------------
```