1. Ranking Techniques: Listwise and Pairwise

Listwise Ranking

In listwise ranking, the model considers the entire list of items to be ranked rather than individual pairs or items. It's typically used in learning-to-rank algorithms to optimize the order of a list, considering overall relevance.

Example Objective: NDCG (Normalized Discounted Cumulative Gain), which measures the quality of a ranked list based on relevance scores.

Formula:

- For a list of items with predicted relevance scores yyy and true relevance y′y'y′: DCGp=∑i=1p2yi′−1log2(i+1)\text{DCG}_p = \sum_{i=1}^{p} \frac{2^{y'_i} - 1}{\log_2(i+1)}DCGp=i=1∑plog2(i+1)2yi′−1 where ppp is the position in the ranked list.
- NDCG normalizes this against the ideal DCG, where items are perfectly ordered by relevance.

Tools and Libraries:

- **TensorFlow Ranking** and **XGBoost** for listwise ranking models.

Pairwise Ranking

In pairwise ranking, the model learns relative preferences between two items. It's especially useful for tasks where items need to be ranked by comparison rather than absolute scores.

Example Objective: Pairwise loss (e.g., Hinge Loss), where the model penalizes incorrect orderings in item pairs.

Formula:

- Given two items iii and jjj with scores sis_isi and sjs_jsj: L=max(0,1−(si−sj)⋅sign(yi−yj))L = \max(0, 1 - (s_i - s_j) \cdot \text{sign}(y_i - y_j))L=max(0,1−(si−sj)⋅sign(yi−yj)) where yiy_iyi and yjy_jyj are true labels indicating preference.

Tools and Libraries:

- **LightGBM** and **RankNet** in PyTorch for pairwise ranking.

---

2. Code Examples

Pairwise Ranking in PyTorch

Here's a simplified example using PyTorch to train a pairwise ranker. We'll use a neural network to learn preferences between pairs.

python
Copy code

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple neural network for ranking
class PairwiseRanker(nn.Module):
    def __init__(self):
        super(PairwiseRanker, self).__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# Pairwise ranking loss (hinge loss)
def pairwise_loss(pos_scores, neg_scores, margin=1.0):
    return torch.mean(torch.clamp(margin - (pos_scores -
neg_scores), min=0))

# Sample data (positive and negative pairs)
model = PairwiseRanker()
optimizer = optim.Adam(model.parameters(), lr=0.01)
for epoch in range(10):
    optimizer.zero_grad()
    pos_input, neg_input = torch.rand((5, 10)), torch.rand((5, 10))
# Positive and negative samples
    pos_scores = model(pos_input)
    neg_scores = model(neg_input)
    loss = pairwise_loss(pos_scores, neg_scores)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

Listwise Ranking with TensorFlow Ranking

TensorFlow Ranking provides prebuilt utilities for listwise ranking. Here's an example to train a listwise ranker with a sample dataset:

python
Copy code

```
import tensorflow as tf
import tensorflow_ranking as tfr

# Create model and compile with a listwise loss
feature_columns = [tf.feature_column.numeric_column("features",
shape=(10,))]
model = tf.keras.Sequential([
    tf.keras.layers.DenseFeatures(feature_columns),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer="adam",
loss=tfr.keras.losses.get("list_mle_loss"))

# Sample dataset for training
train_data = {
    "features": tf.random.normal([32, 10]),
    "labels": tf.random.uniform([32, 1], maxval=2, dtype=tf.int32)
}

# Train model
model.fit(train_data, epochs=5)
```

---

3. Tools and Code for Each Block

Personalization

- **Tool**: TensorFlow, PyTorch, and Redis.

Code for Embedding-Based Recommendations in TensorFlow:
python
Copy code

```
import tensorflow as tf
user_embedding = tf.keras.layers.Embedding(input_dim=1000,
output_dim=64)
```

```
item_embedding = tf.keras.layers.Embedding(input_dim=1000,
output_dim=64)

# Calculate similarity between user and item embeddings
def recommend(user_id, item_ids):
    user_emb = user_embedding(user_id)
    item_embs = item_embedding(item_ids)
    scores = tf.linalg.matmul(user_emb, item_embs, transpose_b=True)
    return tf.argsort(scores, direction='DESCENDING')
```

-

Recommendations

- **Tool**: LightGBM, Matrix Factorization libraries (e.g., Implicit).

Code for Matrix Factorization Recommendations using LightGBM:
python
Copy code

```
import lightgbm as lgb
train_data = lgb.Dataset(X_train, label=y_train)
params = {'objective': 'lambdarank', 'metric': 'ndcg'}
model = lgb.train(params, train_data, num_boost_round=100)
```

-

Ranking

- **Tool**: TensorFlow Ranking, PyTorch, or scikit-learn for traditional models.

Code for NDCG Calculation in Python:
python
Copy code

```
import numpy as np

def ndcg_score(y_true, y_pred, k=10):
    ideal = sorted(y_true, reverse=True)[:k]
    dcg = sum((2**y - 1) / np.log2(idx + 2) for idx, y in
enumerate(y_pred[:k]))
    idcg = sum((2**y - 1) / np.log2(idx + 2) for idx, y in
enumerate(ideal))
    return dcg / idcg if idcg > 0 else 0.0
```

-

Search

- **Tool**: Elasticsearch and FAISS for semantic search.

Code for Elasticsearch Full-Text Search:
python
Copy code

```
from elasticsearch import Elasticsearch

es = Elasticsearch()
query = {
    "query": {
        "bool": {
            "must": {
                "multi_match": {
                    "query": "machine learning",
                    "fields": ["title", "content"]
                }
            },
            "should": [{"term": {"user_id": "12345"}}]  #
Personalized boost
        }
    }
}
results = es.search(index="posts", body=query)
```

-

Content Discovery

- **Tool**: Neo4j for graph-based recommendations.

Code for Graph-Based Content Discovery in Neo4j:
python
Copy code

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687",
auth=("neo4j", "password"))
with driver.session() as session:
    query = """
    MATCH (u:User)-[:FOLLOWS]->(f:User)-[:POSTED]->(p:Post)
    WHERE u.user_id = $user_id
    RETURN p ORDER BY p.popularity DESC LIMIT 10
    """
    posts = session.run(query, user_id="12345")
```

-

---

4. Quantitative Analysis (Advanced)

With listwise and pairwise ranking, we balance accuracy and computational cost.

Listwise Ranking:

- **Complexity**: Higher computational cost but better accuracy. Ideal for static or high-value recommendations.
- **Latency**: Requires batching or preprocessing for large-scale usage.

Pairwise Ranking:

- **Complexity**: Pairwise is faster due to pair comparisons rather than entire list sorting. Ideal for dynamic, real-time rankings.
- **Latency**: More responsive in real-time settings.

Quantitative Metrics:

- **Memory Usage**: Embedding and user interaction matrices can grow large; compressing embeddings and sharding by users helps.
- **Latency**: Aim for under 100ms for both recommendation and search latency to ensure smooth UX.
- **Scalability**: Neo4j scales well with user connections, and Elasticsearch can handle large-scale text-based queries.

TWO TOWER RECOMMENDATION

The Two-Tower (or Dual-Tower) recommendation model is a popular architecture used in recommendation systems, especially when you need to match users with content (e.g., posts, movies) based on embeddings. This model is commonly applied in scenarios requiring personalized recommendations, such as YouTube's video recommendations or Twitter's post recommendations.

Two-Tower Architecture Overview

The Two-Tower model has two separate neural network towers: one for the user embeddings and another for the item embeddings. These towers are trained in parallel, and each outputs an embedding vector. The similarity between these two embeddings determines the recommendation score for a specific user-item pair. This architecture is efficient for large-scale recommendations as it pre-computes embeddings separately, enabling fast retrieval.

Key Benefits

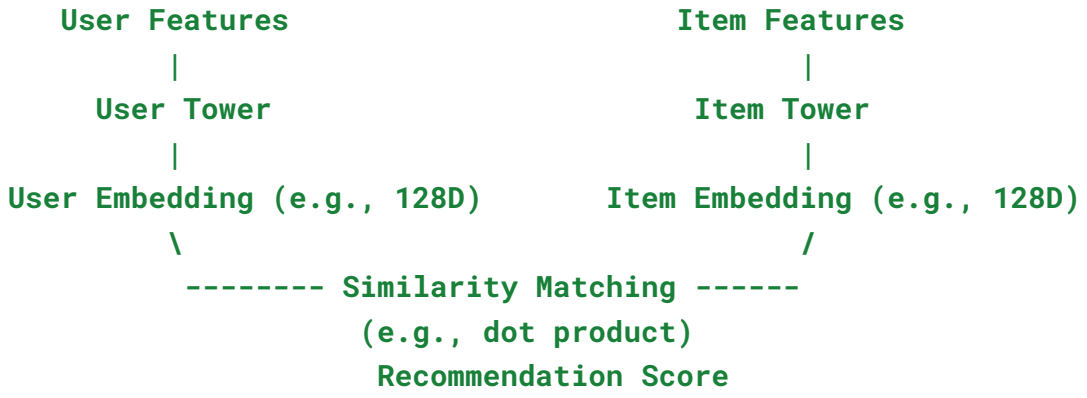
- **Efficiency: User and item embeddings are computed separately, enabling caching and quick retrieval.**
- **Scalability: Allows pre-computation of embeddings for fast, real-time recommendations.**
- **Modularity: Each tower can be optimized independently for better personalization.**

1. Architecture and Workflow

1. **User Tower: Inputs user-specific features (e.g., user ID, user behavior, interests) and outputs a user embedding vector.**
2. **Item Tower: Inputs item-specific features (e.g., content type, tags, popularity) and outputs an item embedding vector.**
3. **Similarity Matching: The recommendation score is the similarity (often cosine similarity or dot product) between the user and item embeddings.**

Diagram

plaintext
Copy code



2. Tools

- **TensorFlow / PyTorch: Build and train the Two-Tower model.**
- **Annoy or FAISS: Efficient approximate nearest neighbors for embedding similarity search.**
- **Redis: Store precomputed embeddings for fast lookup.**
- **Apache Kafka: For streaming user-item interactions to update embeddings in near real-time.**

3. Code Example in TensorFlow

Here's a simplified implementation of a Two-Tower recommendation model in TensorFlow:

python
Copy code

```
import tensorflow as tf

# User Tower: Inputs are user features (e.g., age, gender, past
interactions)
user_input = tf.keras.layers.Input(shape=(10,),
name='user_features')
user_tower = tf.keras.layers.Dense(64,
activation='relu')(user_input)
user_embedding = tf.keras.layers.Dense(128)(user_tower)

# Item Tower: Inputs are item features (e.g., genre, popularity)
item_input = tf.keras.layers.Input(shape=(10,),
name='item_features')
item_tower = tf.keras.layers.Dense(64,
activation='relu')(item_input)
item_embedding = tf.keras.layers.Dense(128)(item_tower)

# Similarity matching (dot product for simplicity)
similarity = tf.keras.layers.Dot(axes=1)([user_embedding,
item_embedding])

# Model
model = tf.keras.Model(inputs=[user_input, item_input],
outputs=similarity)
model.compile(optimizer='adam', loss='binary_crossentropy')

# Training on user-item pairs
user_features = tf.random.normal((1000, 10))  # Sample user features
item_features = tf.random.normal((1000, 10))  # Sample item features
labels = tf.random.uniform((1000, 1), maxval=2, dtype=tf.int32)  #
Interaction labels
model.fit([user_features, item_features], labels, epochs=5)
```

4. Quantitative Analysis

Key Metrics

- **Latency: Aim for sub-10ms latency by caching user and item embeddings. Dot product similarity search with FAISS/Annoy enables near-instant recommendations.**
- **Throughput: For real-time systems, aim to handle thousands of requests per second, especially during peak times.**

- **Storage Requirements:**
    - **User and Item Embeddings: With 1M users and 1M items, using 128-dimensional embeddings would require ~1 GB of storage (assuming float32 encoding).**
    - **Redis Cache: Storing embeddings in Redis improves retrieval times and reduces main database queries.**

Trade-Offs:

- **Latency vs. Accuracy: FAISS or Annoy offers approximate nearest neighbor search, trading slight accuracy for speed.**
- **Real-Time Updates vs. Batch Processing: Updating embeddings in real time for dynamic personalization is more accurate but requires higher compute. Alternatively, batch updates can reduce load.**

Detailed Workflow for Two-Tower Recommendation System

1. **Data Collection and Processing:**
    - **Use Apache Kafka to collect user interactions (e.g., clicks, views, likes) and feed them to the Two-Tower model.**
    - **Preprocess features for the user and item towers.**
2. **Training Phase:**
    - **Train the user and item towers in parallel, with embeddings optimized for similarity matching.**
    - **Optimize based on relevance metrics like AUC (Area Under Curve) or hit rate.**
3. **Embedding Storage and Retrieval:**
    - **After training, precompute and cache embeddings for all items in FAISS or Redis, allowing efficient similarity search.**
    - **Use Redis for low-latency access to user embeddings, updating them periodically as new interactions are ingested.**
4. **Real-Time Recommendations:**
    - **For each recommendation request, compute or retrieve the user embedding, and match it with the most similar item embeddings.**
    - **Use the similarity score to rank items and generate the top recommendations.**

Advanced Optimization Considerations

- **FAISS Indexing: Build an index for the item embeddings and perform approximate nearest neighbor search to find top-N recommendations.**
- **Hybrid Two-Tower Setup: Use a hybrid of two-tower and traditional collaborative filtering for cold start items (new or rarely interacted items).**
- **Real-Time Embedding Update: Retrain embeddings in near real-time for active users, dynamically adjusting based on recent user behavior.**

Here's an example of how to integrate FAISS for similarity search:

python
Copy code

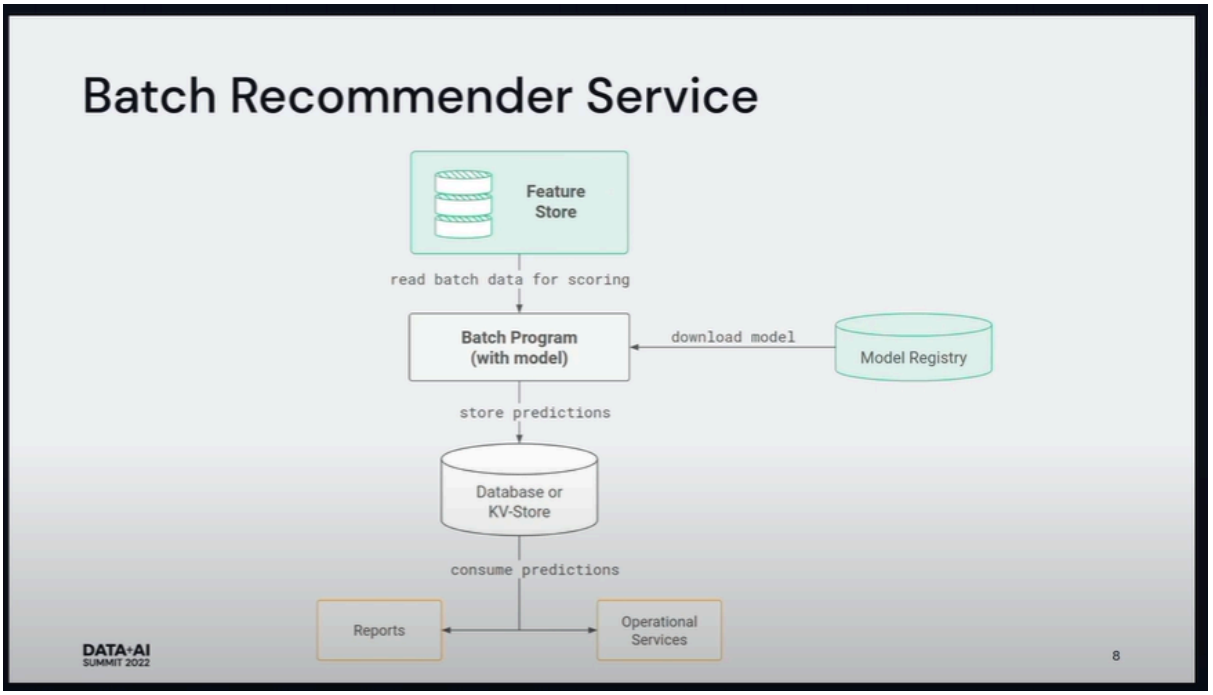
```
import faiss
import numpy as np

# Assume embeddings is a numpy array of item embeddings
d = 128  # Dimension of embeddings
index = faiss.IndexFlatL2(d)
index.add(embeddings)  # Add item embeddings to FAISS index

# Search for the top 10 nearest neighbors to a user embedding
user_embedding = np.random.rand(1, d).astype('float32')
D, I = index.search(user_embedding, 10)  # Distances and indices of
the top 10 items
```

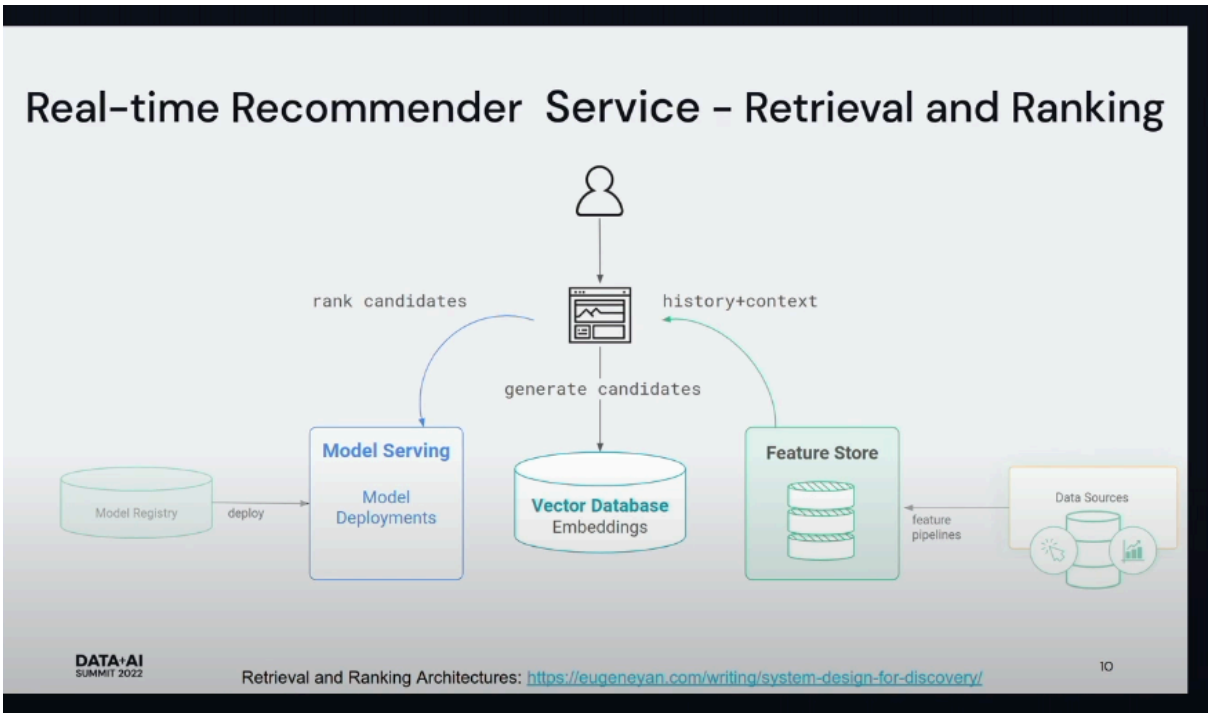
Final Takeaways

The Two-Tower model is highly efficient for large-scale recommendation systems due to its parallel embedding generation and matching mechanism. By integrating FAISS, Redis, and Kafka, this system can achieve low latency and high scalability, making it suitable for real-time, personalized recommendations.

This setup is flexible and can be scaled further by adding more embeddings or adjusting the architecture for hybrid recommendation scenarios. Let me know if you'd like additional details on any specific area or a deeper exploration of similarity search with FAISS and Redis!

DIAGRAM


Batch Recommender Service
Feature Store
read batch data for scoring
Batch Program (with model)
download model
Model Registry
store predictions
Database or KV-Store
consume predictions
Reports
Operational Services
DATA+AI SUMMIT 2022
8


Real-time Recommender Service - Retrieval and Ranking
rank candidates
history+context
generate candidates
Model Serving
Model Deployments
Model Registry
deploy
Vector Database
Embeddings
Feature Store
feature pipelines
Data Sources
DATA+AI SUMMIT 2022
Retrieval and Ranking Architectures: https://eugeneyan.com/writing/system-design-for-discovery/
10