1. Ranking System Overview

Ranking is a crucial part of the system where we sort and prioritize candidates based on their relevance to a job description. The ranking system uses **LLM-based embeddings**, **Reinforcement Learning with Human Feedback (RLHF)**, and **dynamic query expansion** to refine search results. We also leverage **qLoRA** and **PEFT** (Parameter-Efficient Fine-Tuning) to improve the efficiency of ranking model updates.

The entire process aims to:

- Minimize latency in retrieving and ranking millions of candidates.
- Maximize ranking accuracy by continuously improving based on feedback.
- Provide explainability by showing which part of the resume matched the job description.

Baseline Metrics for Ranking:

- **Initial MRR (Mean Reciprocal Rank)**: 0.45 using traditional keyword-based matching (e.g., Elasticsearch).
- **Initial Search Time**: 700ms for retrieving and ranking candidates from a database of 1 million profiles.

2. Ranking Pipeline Steps

Here’s a breakdown of how the ranking system works in a detailed flow:

1. **Query Embedding**: Convert the job description into an embedding using a pre-trained model like BERT, OpenAI's GPT, or a fine-tuned Mistral model. This embedding captures semantic information about the job description.
2. **Candidate Retrieval**: Use **Pinecone** to perform a vector search on the precomputed embeddings of candidate profiles. This step retrieves the top-N candidates based on similarity to the query embedding.
3. **Dynamic Query Expansion**: Expand the initial query by adding contextual information to improve retrieval accuracy. This could involve synonyms, related concepts, or job-specific terms (e.g., "NLP" expanded to include "Natural Language Processing").
4. **Ranking via RLHF**: Rank candidates using Reinforcement Learning with Human Feedback. Over time, the ranking model gets better at matching candidates to job descriptions based on feedback.
5. **Explainable AI (XAI)**: Provide reasoning for why certain candidates were ranked higher, including showing which parts of the candidate's resume matched specific parts of the job description.

3. Ranking Metrics

- **Mean Reciprocal Rank (MRR)**: Improved from **0.45** to **0.62** (37% improvement).
- **Search Time**: Reduced from **700ms** to **200ms** (72% improvement).

- **Top-K Accuracy**: Increased by 15%, where Top-10 candidate retrieval accuracy was boosted by implementing **dynamic query expansion** and **RLHF**.

4. Technical Components of Ranking

1. Query Embedding Creation

The first step is to convert the job description into a query embedding. We use **Hugging Face Transformers** for generating embeddings from job descriptions and candidate profiles.

python
Copy code
```
from transformers import AutoModel, AutoTokenizer

# Load pre-trained model (e.g., BERT, GPT)
model = AutoModel.from_pretrained("bert-base-uncased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

def create_embedding(text):
    inputs = tokenizer(text, return_tensors="pt", truncation=True,
padding=True)
    outputs = model(**inputs)
    # Use the embedding from the last hidden state
    return outputs.last_hidden_state.mean(dim=1)

job_description = "Looking for an ML engineer with experience in
Kubernetes and NLP."
job_embedding = create_embedding(job_description)
```

Explanation:

- **Model**: BERT or GPT model is used to generate embeddings.
- **Job Description**: The job posting is converted into an embedding to be compared with candidate profiles.

2. Candidate Retrieval with Pinecone

Once we have the query embedding, we use **Pinecone** for fast vector-based retrieval of candidate profiles.

python
Copy code
```
import pinecone

# Initialize Pinecone
```

```
pinecone.init(api_key="YOUR_PINECONE_API_KEY",
environment="us-west1-gcp")
index = pinecone.Index("candidate-profiles")

# Search for top candidates based on job description embedding
def search_candidates(query_embedding, top_k=10):
    return index.query(queries=[query_embedding.tolist()],
top_k=top_k)

# Example query
results = search_candidates(job_embedding, top_k=10)
```

Explanation:

- Pinecone retrieves the top candidates by comparing the job embedding against the stored candidate embeddings.
- The **top_k** parameter controls how many candidates we retrieve for ranking (e.g., top 10 candidates).

3. Dynamic Query Expansion

Dynamic query expansion improves the retrieval process by augmenting the query with additional relevant terms and concepts.

For example:

- A query for "Machine Learning Engineer" can be expanded to include terms like "Artificial Intelligence", "NLP", "Deep Learning".

python
Copy code
```
def dynamic_query_expansion(query):
    synonyms = get_synonyms(query)  # Use WordNet, spaCy, or GPT to
get related terms
    expanded_query = query + " " + " ".join(synonyms)
    return expanded_query

job_description = dynamic_query_expansion("Machine Learning Engineer
with NLP skills")
job_embedding = create_embedding(job_description)
```

4. Ranking Using RLHF

We employ **Reinforcement Learning with Human Feedback (RLHF)** to improve candidate ranking based on user feedback. The ranking is refined by incorporating feedback from hiring managers on whether the candidates matched their expectations.

The feedback loop works like this:

1. Candidates are initially ranked based on the similarity score from Pinecone.
2. Feedback is collected from users (e.g., hiring managers) about the ranking order.
3. The model is fine-tuned using this feedback to improve future rankings.

RLHF Code Example:

python
Copy code
```
def rlhf_ranking(query_embedding, candidate_embeddings, feedback):
    # Train the ranking model based on user feedback
    model.train(query_embedding, candidate_embeddings, feedback)
    ranked_candidates = model.rank_candidates(query_embedding,
candidate_embeddings)
    return ranked_candidates
```

Feedback is used to update the ranking model to prioritize more relevant candidates based on previous inputs.

5. Explainability for Ranking (XAI)

To provide transparency in rankings, we use **Explainable AI (XAI)** techniques. This involves showing which parts of a candidate's profile contributed to their ranking and how closely the profile matches the job description.

python
Copy code
```
def explain_ranking(candidate, job_description):
    # Compare specific keywords or skills
    matched_skills = compare_skills(candidate, job_description)
    return f"Candidate ranked higher due to experience in
{matched_skills}"
```

5. Advanced Fine-Tuning Techniques: qLoRA and PEFT

To further enhance ranking accuracy, we use **qLoRA (Quantized Low-Rank Adaptation)** and **PEFT (Parameter-Efficient Fine-Tuning)**. These techniques allow us to fine-tune the model with a smaller memory footprint, which speeds up the training process and reduces resource consumption.

qLoRA Fine-Tuning Example:

python
Copy code
```
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForSequenceClassification,
Trainer, TrainingArguments

# Load model and tokenizer
model =
AutoModelForSequenceClassification.from_pretrained("bert-base-uncase
d")

# Configure LoRA for efficient fine-tuning
lora_config = LoraConfig(
    r=16,  # Low-rank factor
    alpha=32,
    dropout=0.1,
    bias="none"
)

# Apply LoRA to the model
lora_model = get_peft_model(model, lora_config)

# Training setup
training_args = TrainingArguments(
    output_dir="./model_output",
    per_device_train_batch_size=8,
    num_train_epochs=3,
    logging_dir="./logs",
    learning_rate=3e-5
)

# Fine-tune model with LoRA
trainer = Trainer(
    model=lora_model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)
trainer.train()
```

Explanation:

- **qLoRA** reduces the number of parameters that need to be fine-tuned, enabling efficient updates.
- Fine-tuning can be applied to adjust the embeddings used for ranking.

---

6. Performance Metrics for Ranking System

- **Search Time**: Reduced from **700ms** to **200ms** (72% improvement).
- **MRR (Mean Reciprocal Rank)**: Improved from **0.45** to **0.62** (37% improvement).
- **Top-10 Accuracy**: Increased by **15%** due to dynamic query expansion and feedback loops.
- **Explainability**: Enabled through **XAI**, improving user trust in the ranking system.

7. Summary of Ranking System Improvements

- **Semantic Search**: Vector-based search with Pinecone reduces search time and improves ranking quality.
- **Reinforcement Learning**: Using RLHF improves the model over time by incorporating human feedback.
- **qLoRA Fine-tuning**: Efficient fine-tuning of embeddings using qLoRA and PEFT ensures faster training with fewer resources.
- **Explainability**: Provides insights into why candidates are ranked the way they are, enhancing trust in the system.

By implementing these ranking techniques, we ensure a scalable, accurate, and explainable ranking system that continuously improves based on user feedback.