Microservices Architecture-

1. Training

1.1 trainJDTitle(List[JD, JobTitle])

Output: 1.1.1 JD embedding model, 1.1.2 Title embedding model

Model: Asymmetric Bi-encoder where one of the networks behaves as an embedding extraction model for the JD and the other just a transformation of embeddings of the title. Diagram: https://www.sbert.net/examples/applications/cross-encoder/README.html?highlight=encoder , https://www.sbert.net/docs/package_reference/models.html?highlight=asym#sentence_transformers.models.Asym .

Link: https://colab.research.google.com/drive/15CXVzmU7_00qEUviaXCxTuu1rvxFhxSH?usp=sharing

Data: https://github.com/binoydutt/Resume-Job-Description-Matching/blob/master/data.csv

Note that this architecture enables to predict job title even if the exact term is not their

1.2 trainJDSkills(List[JD, [skillList]])

Output: 1.2.1 JD embedding model, 1.2.2 Skills embedding model

Model: Same as 1.1

1.3 trainLocationExtractor(List[JD, [locations]])

Output: Location extraction model

Model: Multilabel Classifier (Loss - BinaryCrossEntropy, Activation - Sigmoid)

Details: Embeddings could be DistillBert or Word2Vec, both can be finetuned

Note: CNN-LSTM/CRF NERs (Eg: spacy) also an option

1.4 trainIndustryExtractor(List[JD, Industry]) , Output: Industry extraction model

Output: Location extraction model

Model: Multilabel Classifier (Loss - CategoricalCrossEntropy, Activation - Softmax)

Details: Embeddings could be DistillBert or Word2Vec, both can be finetuned

1.5 trainEduDegExtractor(List[JD,[EduDeg]]) , Output: Education Degree extraction model

Output: Location extraction model

Model: Same as 1.3

Note: The same models above can be trained on a dataset containing both resumes & JD. Using a common model for both of them is robust since the model generalizes well and applies the knowledge of Jd into resumes and vice-versa. In case of major structural differences and context lengths, it is more robust to have different models for Resume. Hence, this decision is a function of EDA and experimentation

1.6 trainRanker(List[JD, Resume])

Model: Cross-Encoder

Details: Input is prepared by using Parsers at 3. Eg input:

(

“Job Title: Software Engineer \n Skills: Java, Python \n Location: Hyderabad \n Education Degree: Msc \n Industry: IT ”,

“Skills: Flask, Python \n Education Degree: MS \n Industry: IT ”,

1.0

)

The first string in the input triplet contains JD information extracted and concatenated using Parsers at 3. The second index in the triplet corresponds to resume and the third is 1.0 or matching score between 0 and 1 if available in the human annotated data. The model scores between 0 and 1. To further enhance accuracy, the input to the encoder can be appended with an extractive summary for both resume and JD.

Embeddings: sentence-transformers/all-MiniLM-L6-v2 or distilbert-base-uncased

Link:https://github.com/hammad7/plagarism/blob/master/FinetuneBert_cross_encoder_sentence_transformers_continuous_outputs.ipynb

2. VectorDB (Utility Service)

2.1 indexDocument (doc)

This service uses FAISS (or even ElasticSearch supports vector retrieval) to store embeddings for efficient retrieval. Document Schema:

{

DBId: <primary id in DB>,

Text: <name of title/skill>,

Embedding: n- dim embedding,

Type: skill/jobTitle,

sourceDocType: JD/Resume,

derivedSummary: <pre-extracted skills \n title \n location \n degree for Jd/Resme>

}

2.2 indexJobTitle(str: jobTitle)

Uses the model trained at 1.1.2 to encode the jobTitle and indexes using indexDocument utility at 2.1

2.3 indexSkil (str: skill)

Uses the model trained at 1.2.2 to encode the Skill and indexes using indexDocument utility at 2.1

2.4 indexResume (str: Resume)

Uses the model (or similar) trained at 1.1.1 to encode the Resume

Use Parsers at 3 to derive skills, Degree, Industry, location to store in derivedSummary field

indexes using indexDocument utility at 2.1

This method is called **everytime** a new candidate submits resume

2.5 indexJD (str: JD)

Uses the model trained at 1.1.1 to encode the JD

Use Parsers at 3 to derive skills, Degree, Industry, location to store in derivedSummary field

indexes using indexDocument utility at 2.1

This method is called **everytime** a new JD is posted

2.6 queryIndex(n-dimenstional embedding, type)

This queries FAISS/ES index to retrieve top-n titles/skills using cosine similarity and returns them with match scores

2.7 getDocs(JDId/ResumeIds)

Return documents, specifically the derived summary

3. Parser

3.1 extractJobTtiles(JDId (or ResumeId))

Fetch JD from DB corresponding to JDId

This function uses the model at 1.1.1 to encode the JD into document embedding and uses the queryIndex function at 2.6 to get top title with a relevant threshold for cosine similarity score eg. queryIndex(embedding = <docEmbedding>, type = jobTitle). The score of 0->1 can be scaled to 0->100

3.2 extractSkills(JDId (or ResumeId))

Similar to 3.1 but with top-n retrieval of skills instead of top title. Uses model at 1.2

3.3 extractLocation(Jd/resume)

Use model at 1.3 to get location

3.4 extractIndustry(Jd/resume)

Use model at 1.4 to get Industry

3.5 extractEduDegree(Jd/resume)

Use model at 1.5 to get Education degrees

4. Scorer

4.1 computeMatchingScores(List[Jd,Resume])

Score uses model at 1.7

4.2 **shortListResumes**(jdID)

Fetch Jd from DB using JDId

Fetch derivedSummary for JD from vectorDB at 2.7 , this will be common for all triplets. Fetch derivedSummary for all DB Resumes from vectorDB at 2.7, these will be at second index of the triplets. Prepare input to cross-enocer and call 4.1 to get scores between 0->1 and scale to 0->100

By Product- Recommendation engine:

- similar jobs using JD embeddings,
- similar candidates using resume embeddings,
- Job recommendations for candidates.

All this is possible since the model architecture inherently transforms the embeddings into a common vector space

Possible shortcomings:

1. Context length is 512 which means JD will be truncated after approx 400 words. Solution- Use LongFormer, or use extractive summarization before feeding into input
2. High latency for cross Encoder retrieval if the number of resumes are large in number say, 10k for model at 1.6. Solution- use a Bi encoder as a pre-step to fetch top-1000 resumes and then run the cross encoder

There are other details from engineering perspective in DB schema, caching, load balancing that can be added if required. Also, train-test split wasn't performed due to lack of data.

Deployment commands:

```
docker build -t jobTitleService .
sudo docker run -v /home/mohd/data:/jobTitleApp -p 9000:5000 jobTitleService
```

Usage:

```
curl -X POST -H "Content-Type: application/json" -d '{"jobDescription": "The employee will work on the Data & Analytics team"}' http://localhost:9000

curl -X POST -H "Content-Type: application/json" -d '{"i":11}' http://localhost:9000/testJD
```