PHASE 5: DEPLOYMENT AND PRODUCTIONIZATION

1. REQUIREMENTS

1.1. Applications

First we need to install following applications,

- Python 3
- Python 3 PIP
- Gunicorn

1.2. Python Libraries

Then we need to install python libraries,

- Flask
- numpy
- pandas
- pdfplumber
- matplotlib
- fuzzywuzzy
- fuzzywuzzy[speedup]
- joblib
- gunicorn
- nltk
- gensim
- python-Levenshtein
- scikit-learn==0.24.2

2. WEB FRAMEWORK

The web framework allows us to easily create websites. There are multiple web frameworks that we can use for deployment like flask, django, etc. We can even build an API in python and create the web interface in another language.

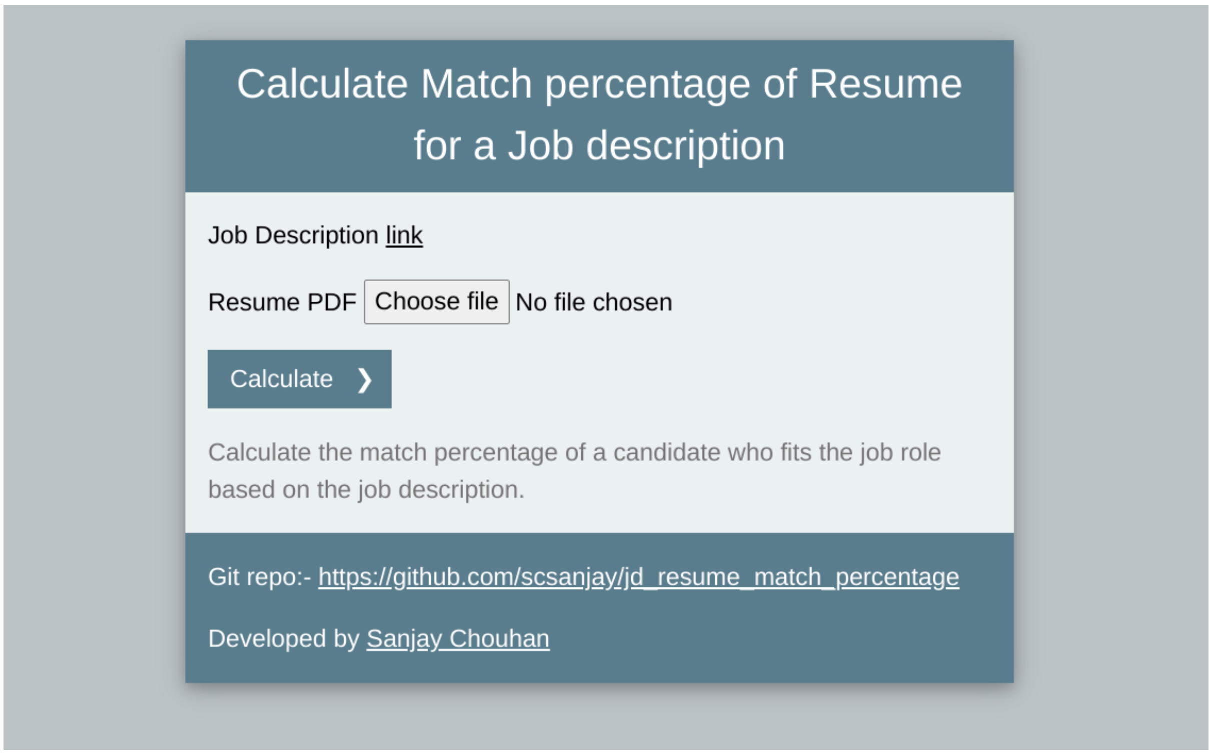
2.1. Flask

I have decided to use Flask which is a micro web framework in python.

It is generally used for creating APIs and building small websites. We will build a simple web-interface so we can use flasks for our task.

2.1.1. Homepage

I have built a homepage where the user can upload the resume. The interface is very simple as we can see below.



Calculate Match percentage of Resume for a Job description

Job Description link

Resume PDF Choose file No file chosen

Calculate ❯

Calculate the match percentage of a candidate who fits the job role based on the job description.

Git repo:- https://github.com/scsanjay/jd_resume_match_percentage

Developed by Sanjay Chouhan

I have fixed the Job description to reduce complexity. And also because we trained the model with only one job description so it will not perform well with other job descriptions.

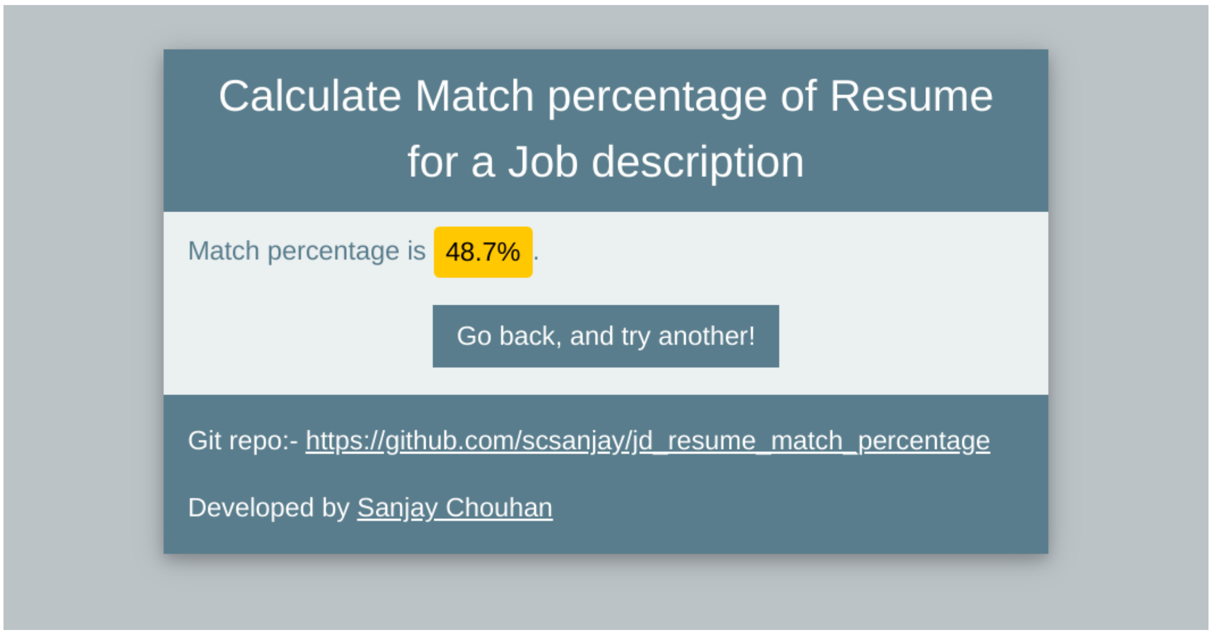
Then we have the input field to upload a resume. It will only accept pdf files of less than 3MB.

And we have the “Calculate” button to submit the form and show the result.

2.1.2. Predict page

On the predict page I have simply displayed the match percentage. And we have a back button to take back to the homepage so that the user can try again with a different resume.

Then we have the predict page to show the calculated match percentage,


Calculate Match percentage of Resume for a Job description

Match percentage is 48.7%.

Git repo:- https://github.com/scsanjay/jd_resume_match_percentage

Developed by Sanjay Chouhan

In an ATS system we would have a different approach. There we would have form which generally get filled automatically when we upload the resume. And the user is allowed to modify the data in case there are some mistakes. This way we get structured data and we can train better models.

Also in ATS we do not show the match percentage to the user but show the match percentage to the hiring manager in the backend.

2.2. Flask code

2.2.1. match.py

In match.py I have initialised flask. It will work as an entry point. Here I have mentioned the constants like upload folder, max size of the resume, etc.

I have created a route for the homepage which will be shown when the user browses either */* or */index*. For the homepage we have just rendered the template.

Another route is for the predict page (*/predict*) which only accepts POST requests. Here at first I have added some constraints which will redirect back to the homepage if not met.

Then I have uploaded the file to the server. And loaded the pdf to preprocess it. After preprocessing I have passed it through the models to get the prediction. And displayed the result on the prediction page.

2.2.2. helper.py

The helper.py contains all the preprocessing related functions. I have created this so that we do not clutter all the codes.

2.2.3. models folder

This contains all the preprocessors, models and some static values.

2.2.4. static folder

The static folder is for css, js and favicon.

2.2.5. templates folder

We have stored all the html templates in this folder.

2.2.6. uploads folder

This folder will be used to save the user uploaded resume files. This is a temporary folder so we should clean it from time to time.

3. HOSTING

3.1. WSGI Server

While lightweight and easy to use, Flask's built-in server is not suitable for production as it doesn't scale well. So for productionising a flask based website we need to use some other WSGI Server (Web Server Gateway Interface HTTP server).

There are various options productionisation of Flask apps.

Some of the hosted options are,

- Heroku
- Google App Engine
- Google Cloud Run
- AWS Elastic Beanstalk
- Amazon Lightsail Containers
- PythonAnywhere

- Azure App Service

And some of the self hosted options are,

- Gunicorn
- uWSGI
- CGI

3.1.1. Gunicorn

Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX. It provides a perfect balance of performance, flexibility, and configuration simplicity. With worker parameters it allows us to manage a number of processes to run parallelly for scalability.

3.2. Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in AWS. EC2 is highly scalable and can be launched with only a few clicks. Amazon EC2 is IAAS (Infrastructure as a Service). Here we get a virtual computer with some RAM, CPU, hard disk, OS, etc. We can use this to host a website. And Amazon gives us full control over it.

First I launched a t2.micro instance of amazon EC2 which is a free tier instance. It has only 1 CPU and 1 GB RAM.

Then I created an Elastic IP address and assigned it to the EC2 instance. Because the IP address of the instance might change if we restart the system.

Also we need to create and assign a security group which allows TCP connection to the port on which we will be running the Gunicorn. Otherwise we won't be able to access the instance from the browser.

After the instance setup I transferred all the files of the web app to the EC2 with SFTP. Then I connected to the instance's terminal with SSH and ran the gunicorn command to serve the website.

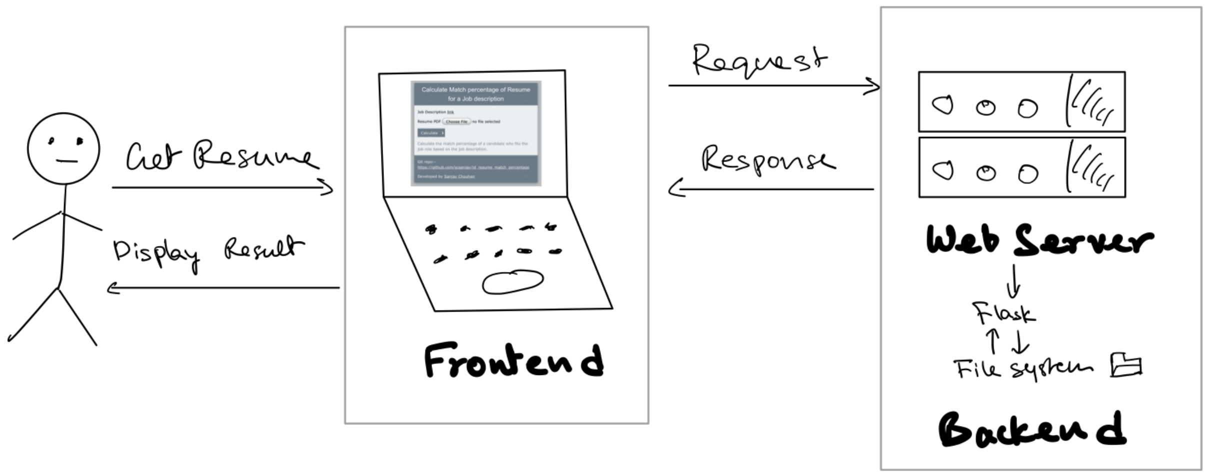
4. OPTIMISATION

I have used some simple optimisation hacks to run the website on a small server.

Since the job description is fixed, I have preprocessed the job description and calculated BoW and average word2vec representation of the job description. This will certainly improve the prediction time.

The pretrained word2vec is 3.7GB that means we won't be able to run it in a system with 1GB RAM. So I limited the vocab size to 3 lakh and also I changed the data type of the representation from float32 to float16. With this I was able to reduce the size to 190 MB.

5. ARCHITECTURE DIAGRAM



6. SCALABILITY AND LATENCY

6.1. Scalability

We can easily scale the website because we are hosting on EC2 instance and using gunicorn. So whenever required we can change the instance type and use a larger instance. Also we can spawn more instances of the application with gunicorn's worker parameter.

We can use caching for the homepage to make it faster. Also we can store the word and the corresponding word2vec representation in an indexed table. This will help with RAM and the speed because then we don't have to load the word2vec model and we can get the representation in O(1) time.

6.2. Latency

On the EC2 instance where we have lots of constraints the latency is 3.9 seconds including the file upload time. Which is good.

On local system where we don't have any constraints the latency is only 1.2 seconds including the file upload time.

So if we want to improve the latency we can increase the RAM and run more workers. And also we can increase the network bandwidth of the EC2 instance. With caching we can improve the load time of the homepage.

If we want more speed we can use C++ to implement the models.

7. REFERENCES

- AppliedRoots. [https://www.appliedroots.com/]
- Flask. [https://flask.palletsprojects.com/]
- Serve Flask with Gunicorn. Digitalocean. [https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-18-04]