Job Matching Algorithm

Matthew Calapai 28/01/2024

Contents

1 Introduction/Abstract

This paper presents the development and analysis of a job match recommendation engine based on the requirements and background outlined here. The engine suggests jobs to jobseekers based on their skills and job requirements. This paper discusses the algorithm's design, its computational efficiency, and its performance on test datasets. The results demonstrate the engine's effectiveness in matching jobseekers with relevant jobs and its scalability in handling large datasets.

2 Methodology

This study's methodology revolves around the development and testing of an algorithm designed for a job match recommendation engine. The algorithm is implemented in Python, with data processing and manipulation facilitated by pandas and NumPy.

2.1 Algorithm Design

The core of the recommendation engine is an algorithm that reads jobseeker and job data from provided CSV files into pandas dataframes. To enhance computational efficiency and speed, these dataframes are converted into NumPy arrays, as the algorithm is optimized for operation on NumPy's array structure.

The algorithm operates by iterating through each dataset in a nested for-loop structure, where each jobseeker's skills are matched against the required skills for each job. This method

hypothesises a time complexity of $O(n \times m)$, with n being the number of jobseekers and m the number of jobs. The "percentage match" between the jobseeker's skills and job requirements is calculated as required. The final output is a CSV file comprising columns for jobseeker ID, job ID, and the number of matching skills and the matching skill percentage, formatted as per the provided instructions.

2.2 Data Preparation

A test dataset, encompassing both jobs and jobseekers, was generated to evaluate the algorithm's computational efficiency and scalability, as well as its robustness in handling edge cases. This dataset was pre-processed to ensure quality and consistency, primarily through the elimination of rows containing NaN values, ensuring the integrity of the matching process.

3 Results

The performance and efficiency of the algorithm were quantitatively assessed using the prepared test dataset.

3.1 Computational Efficiency

To understand the scalability of the algorithm, the average computation time for increasing sizes of the test dataset was recorded. The dataset size varied from 1 to 100 samples, and for each sample size, the algorithm was executed 100 times. The runtime for each iteration was recorded, and these values were then averaged to yield the average computation time for each sample size.

3.2 Analysis of Results

A plot of the average computation time against the growing number of samples was generated to visually represent the algorithm's performance.

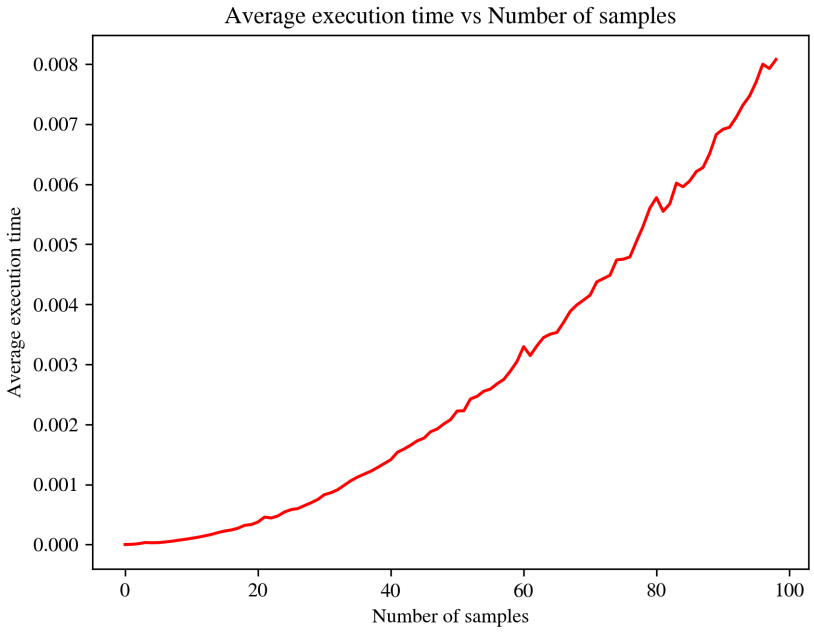

Figure 1: Plot of average computation time against the number of samples in the test dataset.

This plot revealed a polynomial increase in computation time, supporting the hypothesized time complexity of $O(n \times m)$. For reference, the computation was conducted on the 2021 M1 Max MacBook Pro with 32GB RAM.

4 Discussion

The observed polynomial growth in computation time aligns with the initial hypothesis of an $O(n \times m)$ time complexity. This finding has several implications:

4.1 Scalability and Optimization

While the algorithm demonstrates efficiency for moderate-sized datasets, the polynomial time complexity indicates potential challenges in scaling to very large datasets. Optimization strategies, such as parallel processing or more efficient data structures, could be explored to enhance performance.

4.2 Handling of Edge Cases

The preprocessing steps included removal of NaN values for data consistency, although future work could include other methods such as imputation for more robust data preparation.

5 Conclusion

The developed job match recommendation engine effectively matches jobseekers with jobs based on skill sets. Its performance on test datasets confirms the hypothesized computational efficiency, while leaving room for further optimisations for real-world applications.

6 Appendix

6.1 Job Matching Algorithm

```
def simulate_match_jobs(jobs, job_seekers, n_samples, n_iterations):
    job_values = jobs.values[:n_samples] # get first n_samples of jobs
    job_seekers_values = job_seekers.values[:n_samples] # get first n_samples
                                                        of job seekers

    ## predefine skill sets for each job seeker
    job_seeker_skills_sets = [set(js[2].split(', ')) for js in
                                                        job_seekers_values]

    ## predefine skill sets for each job
    job_skills_sets = [set(j[2].split(', ')) for j in job_values]

    ## allocate empty numpy array
    job_matches = np.empty((len(jobs)*len(job_seekers), 6), dtype=object)

    times = []
    for _ in range(0,n_iterations):
        start_time = time.time()

        index = 0
        for js_idx, job_seeker in enumerate(job_seekers_values):
```

```
            job_seeker_id, job_seeker_name = job_seeker[:2] # get first two
                                                items of array (seeker id
                                                and seeker name)

            for job_idx, job in enumerate(job_values):
                job_id, job_title = job[:2] # get first two items of array (
                                                job id and job title)

                matched_skills = job_seeker_skills_sets[js_idx].intersection(
                                                job_skills_sets[job_idx
                                                ]) # get matched skills
                                                 via set intersection
                matched_skills_percent = (len(matched_skills) / len(
                                                job_skills_sets[job_idx
                                                ])) * 100

                ## format as required
                job_matches[index] = [job_seeker_id, job_seeker_name, job_id,
                                                job_title, len(
                                                matched_skills),
                                                matched_skills_percent]
                index+=1

        times.append(time.time() - start_time)

    avg_time = sum(times)/len(times)

    return [avg_time, job_matches]
```

6.2 Additional Code

Can be found on the GitHub Repository.