Machine Learning Driven Job Recommender System

COE70A/B
(Computer Engineering Capstone Design)

Zeshan Fayyaz
Saima Munir
Mohammad Shahbaz Yousaf
Tabish Rashidi

FLC: Dr. Alagan Anpalagan

April 17, 2022

Certificate of Authorship

We hereby certify that we are the authors of this document and that any assistance we received in its preparation is fully acknowledged and disclosed in the document. We have also cited all sources from which we obtained data, ideas, or words that are copied directly or paraphrased in the document

Signed by Zeshan Fayyaz, Saima Munir, Mohammad Shahbaz Yousaf, and Tabish Rashidi

Acknowledgement

Thank you to our patient research supervisor, Dr. Alagan Anpalagan. Their conceptualization, insightfulness, guidance, professionalism, and technical expertise is what has made this project what it is.

This year-long project, involving thorough research and technical implementations is a means of applying the valuable skills learnt at our time as undergraduates at Ryerson University.

CONTENTS

Abstract

The overwhelming surplus of content on the web makes it difficult for users to extract what they desire. This typically results in a poor or no decision being made; this phenomenon is known as information overload [1]. Recommender systems (RS) aim to provide users with an enhanced online search experience where only relevant results are displayed. RS have been deployed and extensively explored in the last decades with the advent of online shopping platforms, such as Amazon or eBay. In this project, we present a novel machine learning-driven RS approach that aims to simplify finding relevant careers based strictly on the qualifications of a candidate. This project is implemented by deploying a knowledge-based RS that uses explicit information about a user - that is, a resume. Our intelligent system, trained on 20,000 real career descriptions, works closely with the frontend to provide accurate information as simply as possible. Although the backend consists of many running processes and algorithms to predict the most relevant careers, we ease the burden of information overload by making the process as simple as possible for the candidate.

Our model uses Latent Dirichlet Allocation (LDA) topic prediction [2] on a large NLP-based preprocessed corpus to extract the topmost prevalent topics of any document. It is of utmost importance that the deployed model has contextual awareness to understand the document and to create high-quality topics. The frontend, built by Django, calls the pre-trained model to efficiently iterate through all documents and display the topmost relevant career descriptions to any given resume. Through extensive training and testing, our model has demonstrated highly effective matching capabilities. The proposed method allows for a more contextually aware feature extraction process, which demonstrates effectiveness throughout all processes.

Index Terms: *Information overload, machine learning, recommender system, similarity, preprocessing*

I. Introduction and Background

As the amount of information available online dramatically increases daily with the advancement of technology and the number of users, it may be challenging to find precisely what you are looking for. This is known as information overload [3]. This excess of information occurs when a user has trouble processing the amount of information available, leading them to become distracted, anxious, or even depressed. It may also cause the user to feel overwhelmed and frustrated and therefore choose an option that isn't ideal. Combining this phenomenon with the already stressful process of finding a new career makes things worse. To resolve this problem, we propose an innovative machine learning-driven recommender system (RS) to overcome information overload in the ever-expanding plethora of online job listings. Our solution utilizes natural language processing (NLP) and machine learning (ML) techniques to intelligently match available job listings to any given resume and simply display the top N most relevant listings.

RS have grown in popularity in the last two decades. As the popularity of online stores increases, the number of choices a user must make on a web application also increases. The utilization of RS cannot be overstated, as it aims to validate its choices by minimizing its error [1]. A resume contains explicit information about a user, then matched to an appropriate listing. Therefore, we utilize the structure of a knowledge-based RS algorithm along with its error and metric calculations. Combining this RS, NLP topic retrieval, and ML techniques is the foundation of our prediction model. We propose an LDA model [4] that trains over an NLP-based preprocessed corpus containing 20,000 real job descriptions over ten epochs with a batch size of 100.

The effectiveness and quality of our topic modelling depends on the preprocessing quality. In NLP, there are a few common steps to ensure we are training our model over a smaller subset of words containing the same amount of contextual information. This can be done by the removal of unnecessary stopwords. We then iterate through all documents to remove all punctuation marks. Next, we perform lemmatization, stemming, and tokenization to break up strings and reduce them into their simplest forms. Finally, we filter over the extreme features by removing words that don't appear over 20 times and those that feature in over 50% of documents. The goal is to remove as many redundant words as possible and be left with valuable information that our model can quickly train over.

Our trained model iterates through all documents and produces an ordered list of the most dominant topics. This information is appended to a dataframe and passed to the similarity metric to determine the most relevant jobs. This extensive process produces accurate results because of its contextual awareness of determining underlying topics.

Previous work on topic retrieval has been performed successfully with exceptional accuracy. Typically, the Latent Dirichlet Allocation (LDA) topic modelling technique is used for this task as text retrieval is related to latent topics of a corpus. Besides this, NLP preprocessing techniques exploit context-based information to help ease the training complexity of the ML model.

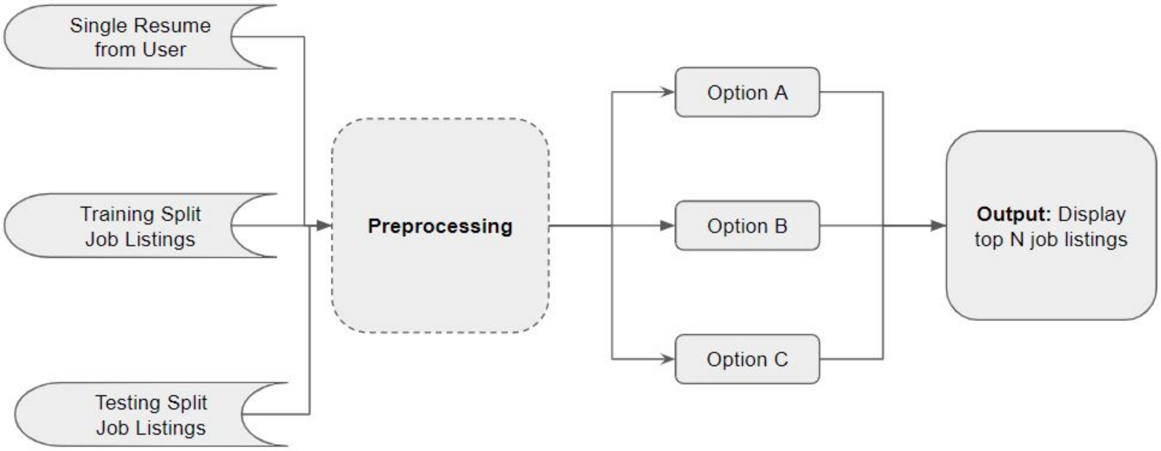

Single Resume from User
Training Split Job Listings
Testing Split Job Listings
Preprocessing
Option A
Option B
Option C
Output: Display top N job listings

Fig. 1: A rough visualization for the project methodology.

As seen in Fig. 1, we desire to display the top N job listings after obtaining the user input, training, preprocessing, and the ML method. We categorize the three ML options as (a) TensorFlow Recommenders [5], (b) pyLDAvis [6] and (c) Low Rank Matrix Factorization [7]. In the next sections, we describe our training choice and parameters.

The structure of this report begins by describing the overall project objectives and end goal. Next, for our design, we outline the theory and how each component works with one another. As this project can be completed in many ways, we then describe the potential alternative designs for the machine learning and backend. The material and components list is then given. Following this, we begin outlining our extensive testing. First, we describe the testing procedures followed by the performance results. We then analyze our performance results. We conclude this comprehensive report by outlining our concluding results, problems faced, and potential future work. Additional information can be found in the appendix.

II. Objectives

The overall objective of this project is to make it as simple as possible for a user to find a relevant job in our database. We do not want the user to enter tedious information to reduce the corpus manually. The approach should be hands-off, as the LDA model should perform most of the computation during training. Therefore, we want to exploit the learning and computational capabilities of intelligent systems to match jobs.

1) Store a large dataset of (20,000) tech job postings
2) Train an LDA topic modelling system that outputs the most prevalent topics per document
3) Create a similarity metric that, given the topics of a resume and any job listing, determines the similarity score
4) Create a fully functional web application that:
   a) Takes in a user's resume as input as well as optional location-specific parameters
   b) Tests the resume on the pre-trained LDA model
   c) Apply the similarity metrics
   d) Outputs the top 5 best job posting recommendations

III. Design and Theory

A. Design

1) Data Preprocessing: The quality, separation, and user-interpretability of our outputted topics greatly depends on the effectivness of our preprocessing.

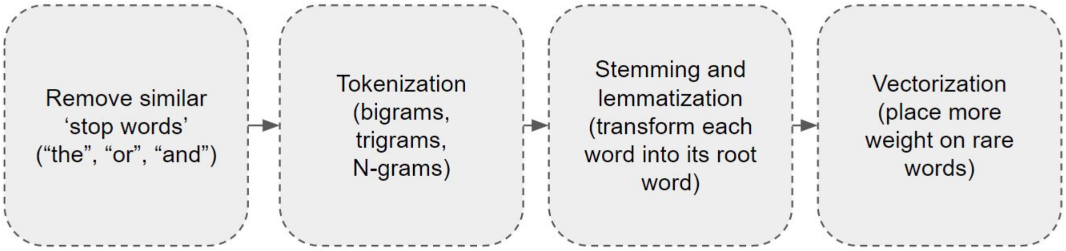

Remove similar 'stop words' ("the", "or", "and")
Tokenization (bigrams, trigrams, N-grams)
Stemming and lemmatization (transform each word into its root word)
Vectorization (place more weight on rare words)

Fig. 2: Flow of preprocessing our corpus.

We utilize libraries such as NLP and NLTK to implement our preprocessing steps. It is crucial to remove many words so our training is more efficient. Training over a fewer amount of words with the same quality of contextual data is highly advantageous and drastically speeds up training.

2) High Level Design: Fig. 3 illustrates how the application operates functionally from a high-level perspective. It also generally describes how the application passes data between its constituent parts to achieve the business logic of the application.

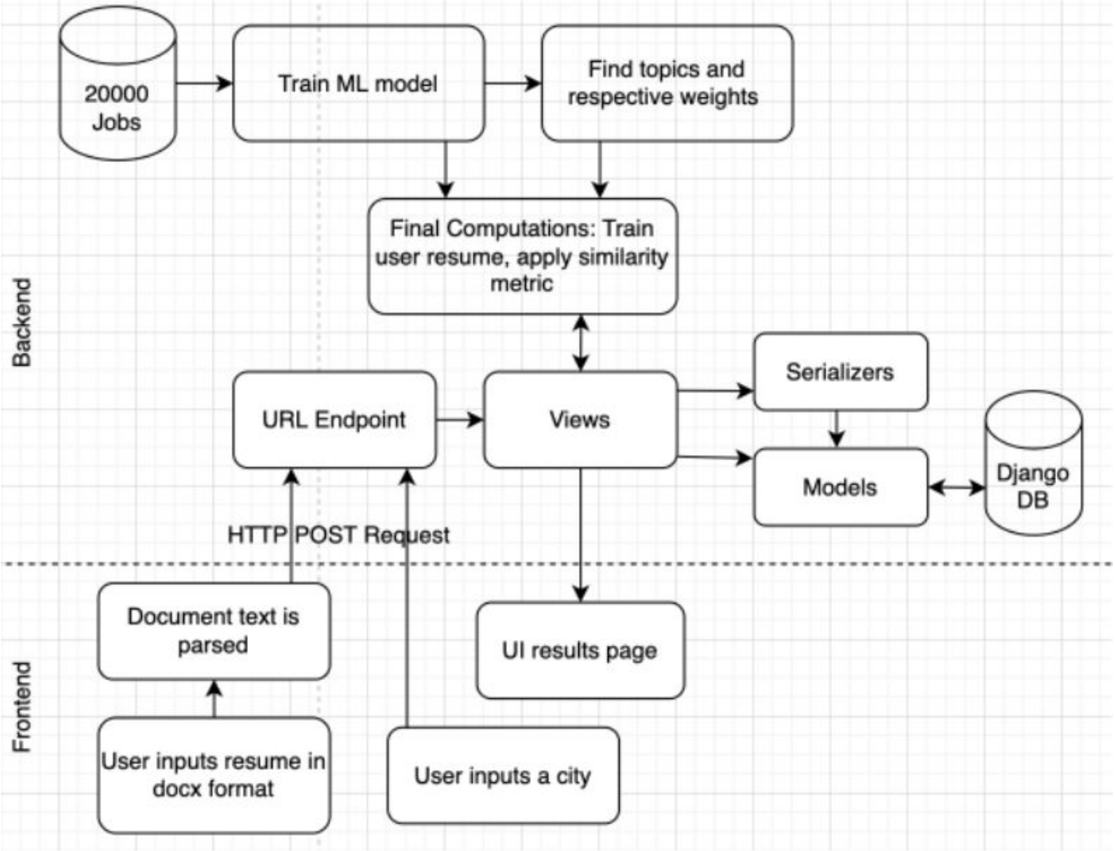


Fig. 3: Backend and frontend connectivity sequence.

First, there is the 20,000 job postings dataset which was trained on the LDA model to determine its topics and respective weight distributions. All of this data is saved and stored to be used later.

The interaction begins on the frontend side, where a client or user of the application uploads a Docx file to the application to the UI. This document is validated, parsed and saved to local application memory. After that, the user is prompted to enter a city to constrain their search results to a single locale. Once chosen in the UI, that city is also saved to local memory and passed to the body of an HTTP POST request. This POST request contains the necessary details the backend application needs to process the data.

This user data is posted on the backend endpoint server. A model was set up to declare the fields for the user input data. Then in the portion of the view, this data can interact with the entire backend, as the view helps structure the code. It is sent to the serializer, which retrieves these objects and serializes them by unpacking the objects into JSON format to be passed around servers. Also, from views, the data is sent as inputs to another function to perform the computations.

This function has stored the model that has been trained and the resulting topics and weights from the jobs dataset. Then, the user resume data is trained on the model to find its topics and weights, and the jobs corpus is reduced based on the user-selected city. From here, the matching algorithm to find the similarity metric is applied between the resume and jobs topics and weights. And the corresponding data of the top 5 job postings with the highest similarity is returned to the frontend.

3) Machine Learning Design: Fig. 4 depicts the flow of events for our LDA topic modelling system. It is imperative that we choose a highly accurate number of topics, as we do not want the user entering the backend and modifying this instance. As will be determined in later sections, based on the SSE and coherence plot, we choose to train over 12 topics.

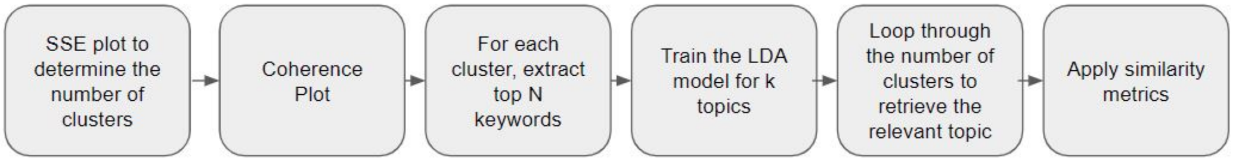

SSE plot to determine the number of clusters
Coherence Plot
For each cluster, extract top N keywords
Train the LDA model for k topics
Loop through the number of clusters to retrieve the relevant topic
Apply similarity metrics

Fig. 4: Flow of machine learning for topic modeling.

For each document, we then iterate through and extract the topmost prevalent topics. Based on this training, the ”test” can be considered the resume, which is fed into the trained model. Once we have all topics listed, the similarity metrics are applied.

B. Theory

1) Latent Dirichlet Allocation (LDA) Model: LDA is a generative statistical model that is commonly used in natural language processing (NLP). For our project, we train our LDA model against a preprocessed corpus of job postings in order to perform topic extraction. Once our model is trained, we preprocess and input in a resume and extract the top N most relevant keywords. Due to our bag-of-words definition, the list of possible keywords for the resume and job postings will be identical. The appearance of any keyword and its order of importance will determine if a posting is relevant to the resume. We use the ordered list output to provide us with an accuracy, which will be compared against the ground truth model.

As mentioned, the LDA model is used for topic extraction in NLP processes. LDA was proposed in 2003 as a ”three-level hierarchical Bayesian model, in which each item of a collection is modelled as a finite mixture over an underlying set of topics.” For our project, the topic probabilities can be a representation of a particular document. An LDA model is able to use contextual information to determine similar (although unique in phrasing) documents.

An LDA model relies on the following assumptions:

- Words that appear together frequently are likely to be close in meaning
- Each topic is a mixture of unique words
- Each document is a mixture of unique topics

LDA is a generative model; that is, it aims to determine how an article is generated from its topics.

Dirichlet prior of the topics determines how sparse or mixed our topics are. Specifically:

- $\alpha < 1$ corresponds to a low Dirichlet prior.
  - A more even mix of topics (distinct topics)
- $\alpha > 1$ corresponds to a high Dirichlet prior.
  - An uneven mix of topics

In the figure above, the left triangle refers to a low Dirichlet prior, where the documents are fairly separated with distinct topics. The right triangle depicts the distribution of topics in documents when there is a high Dirichlet prior; in this case, there is no distinct topic per document.

Dirichlet prior of the terms determines how sparse or mixed our topics are amongst the terms.

An LDA model works by optimizing the following hyperparameters:

- Determine the Dirichlet distribution of topics over terms
- Determine the Dirichlet distribution of documents over topics
- Determine the probability of a topic appearing given a document and the probability of a word appearing in a given topic

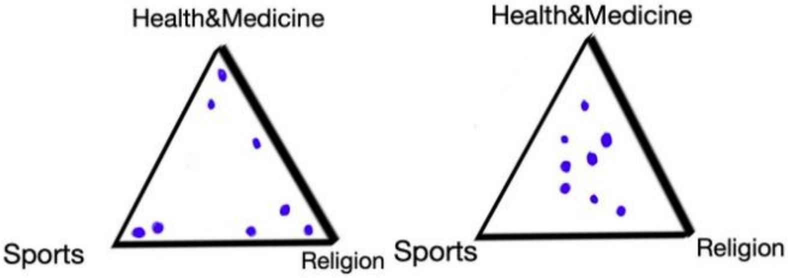


Fig. 5: An example of distribution of topics using an LDA model.

$$\underbrace{P(W,Z,\theta,\phi;\sigma,\beta)}_{\text{Total probability}} = \underbrace{\prod_{i=1}^{K} P(\phi_i;\beta)}_{\substack{\text{Dirichlet}\\ \text{distribution}\\ \text{of topics}\\ \text{over terms}}} \underbrace{\prod_{j=1}^{M} P(\theta_j;\alpha)}_{\substack{\text{Dirichlet}\\ \text{distribution}\\ \text{of documents}\\ \text{over topics}}} \underbrace{\prod_{t=1}^{N} P(Z_{j,t}|\theta_j)}_{\substack{\text{Probability}\\ \text{of topic}\\ \text{in a given}\\ \text{document}}} \underbrace{P(W_{j,t}|\Phi z_{j,t})}_{\substack{\text{Probability}\\ \text{of word}\\ \text{appearing}\\ \text{in topic}}} \tag{1}$$

The total probability of the LDA model is how likely a certain topic, Z, appears in any given document and how likely certain words, W appear in those topics.

Balancing a term's frequency in a particular topic against the term's frequency across the whole corpus is known as relevance. This is an example of the LDA model using contextual information to determine the popularity of a keyword in a topic. This allows us to differentiate between special, unique words and frequently occurring words. We aim to aid the LDA model in this portion of the training by utilizing preprocessing.

Relevance can be calculated as:

$$r(w,k|\lambda) = \lambda log(\phi_{kw}) + (1-\lambda) log(\frac{\phi_{kw}}{p_w}) \tag{2}$$

where,

- r - relevance
- w - term in our vocabulary
- k - topic our LDA model has predicted
- λ - adjustable hyperparameter
- ϕkw - the probability of a word appearing in a topic
- pw - the probability of a term appearing in the entire corpus

2) Ground Truth and Cosine Similarity: The ground truth of a machine learning system is the ideal expected result, and being able to measure this is crucial. To establish a ground truth for the LDA system, a cosine similarity operation is used, which measures the similarity of 2 documents. The benefit of this algorithm is that it does not consider the difference between the sizes of the documents. Cosine similarity is used to compare raw input data against preprocessed input data and a resume against a job description to establish the ground truth for the LDA model's results.

The cosine similarity algorithm functions by projecting two vectors, arrays of words from the respective documents, in a multidimensional space. Each dimension in the space corresponds to a word in the document. The cosine similarity function then measures the angle between the two documents, and the smaller the angle, the greater the similarity.

To implement this function in python, the natural language toolkit ($NLTK$) and $TF - IDF$ are used. $TF - IDF$ refers to the term frequency-inverse data frequency. TF, $tf_{i,j}$, for each document is calculated by taking the amount of times a word, $n_{i,j}$, appears in a document and dividing it by the number of words in a document, given as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \tag{3}$$

The IDF calculates the weight of rare words of all documents in the corpus by finding the log of the number of documents, N, divided by the number of documents with the word w, df_t. This is given as:

$$idf(w) = log(\frac{N}{df_i}) \tag{4}$$

The TF-IDF is the product of the TF and IDF, which is:

$$w_{i,j} = tf_{i,j} * log(\frac{N}{df_i}) \tag{5}$$

To implement the $TF - IDF$ operation, the $TfidfFVectorizer()$ class can convert documents of words to a matrix of $TF - IDF$ features. Within this class, the $fit_transform$ method is used to learn the vocabulary, and IDF then transforms documents to return a sparse matrix of $TF - IDF$ weighted document terms. The final result is the similarity score, which is between 0 and 1.

3) Database Management System: A database is an organized collection of structured information, or data, typically stored in a computer system. However, databases are more recognized by their application counterparts called Database Management Systems (DBMS). The DBMS provides a higher-level interface for users to interact with the database itself, usually involving either storing or retrieving data [8]. Popular DBMS applications include MySQL, PostgreSQL, SQL Server, and SQLite. All of these systems use Structured Query Language (SQL) as the language to carry out the desired commands.

The focus of this section will be on SQLite and its functionality as a database. In the context of the project, this database will be used to store the user resumes and store a large set of job descriptions. Each user will have the option to input their resume, which will then be stored with a unique ID. That resume will then be passed to the machine learning model, referencing the database of job descriptions to return the best matching jobs.

To discuss SQLite further, we must first understand how SQL is working at a higher level. SQL works on what is known as a *transactional* system where any request is met with a response according to the ACID properties [9]. *Atomicity*, groups all the commands of a transaction as a single unit and ensures that they will either succeed or fail. *Consistency* ensures that data is in a consistent state when it starts and ends. *Isolation* is to make sure that any intermediary state of a transaction is hidden from other transactions. *Durability* is to make sure that any changed data is not suddenly undone after the transaction.

SQLite works as a high-performance database. It is mainly employed for keeping data in web applications and parts of larger, more robust applications. It is also lightweight, i.e. less memory intensive and open-source. It differs from other databases because it is considered to be a *serverless database* meaning. In contrast, other DBMS runs in a different address space than the application; SQLite runs within the same address space with the same heap [10]. Other DBMS will receive an SQL statement from the client application, perform the query and then return the result, i.e. *client-server* style. But SQLite will interpret the SQL in the same thread and return the result.

The architecture of SQLite is shown above [11]. Although there are many topics to discuss within that diagram, a high-level understanding of the architecture is necessary. The main idea here is that SQL commands are converted into bytecode and then passed to the Virtual machine, which will then execute the correct commands on B-Tree, Page Cache and OS Interface.

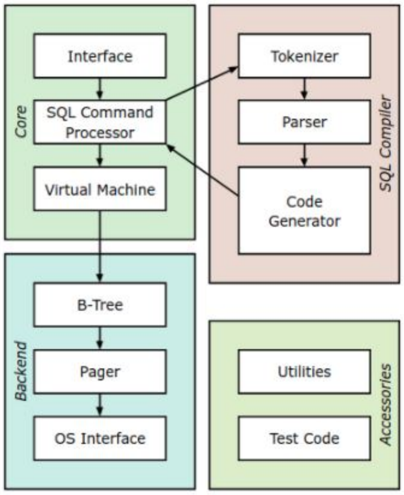


Fig. 6: A typical SQLite architecture.

First, the SQL statement is passed from the interface to the tokenizer, breaking the SQL up into tokens; the parser assigns meaning to these tokens based on context. The code generator creates the bytecode and then passes it down to the B-Tree. The B-Tree is not a Binary Tree; it's another similar data structure that allows any number of children per node. Each table has a B-Tree, and it's used to index the Page Cache quickly. This Page Cache is meant to keep data from the disk in fixed sizes. Finally, the OS Interface is intended to provide the base functionality for reading, writing and closing files on disk. It is meant to be OS-agnostic, i.e. works across Windows and UNIX systems.

4) Web Application: Web applications are pieces or pieces of software that exist on a network. Web applications are the flagship products for many companies and organizations that intend to provide services to their clients. They perform the business logic, which defines or constrains how the business is supposed to operate [12]. Users interact with the web application through a user interface, providing data that the web application processes and returns the result to the user. Internally, web applications are designed in a particular manner known as the application's architecture. This specifies the class of the web application and provides parameters for developers when the application is being developed. For the web application of this specific project, the 3-Tier architecture was chosen.

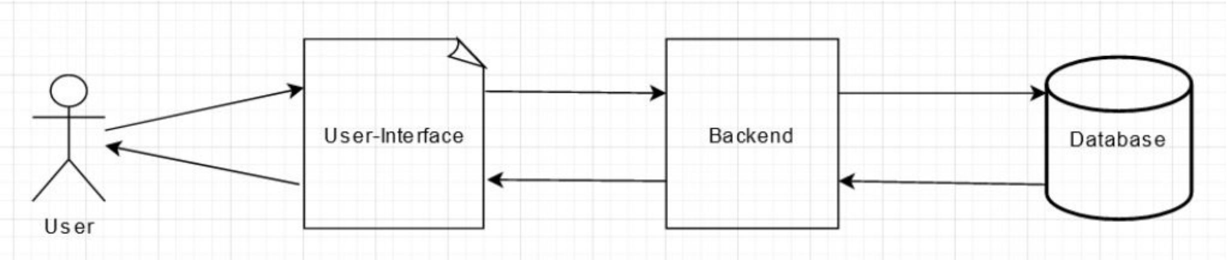


Fig. 7: User-interface architecture.

A 3-tier architecture is composed of a user interface, a backend, and a database [13]. The backend is the most critical piece as it provides the business logic functionality and orchestrates the connection between user and database. It is essentially a server that runs or contains code that allows the correct logic to be executed and, depending on the output, may or may not interact with the database. In a 3-Tier architecture, the user only has direct access to the user interface through a network connection. They interact with the user interface and are provided with options to enter or view data. Viewing data is a simple fetch from the database. Requests from the user can be classified into different types, but the four most relevant HTTP methods are [14]:

- GET - Retrieves data

- POST - Submits entry to the specified resource, causing change on the server
- PUT - Replaces the target resource with the new request payload
- DELETE - Deletes the specified resource

These methods are used in most applications to perform the most common functionality found in web apps. For example, a GET request would display data in a dashboard. A POST request would be able to add an entry to a list of customers. The application outlined below will use a 3-Tier architecture and the HTTP methods to realize a web application that accepts user input. Using an ML model in the backend will match the input to a result in the database and return this to the user.

5) Front-End Development: React: The front-end or "client-side" of a web application is the portion of the website that users can see and interact with on the web browser, whether texts, images or even buttons and navigation menus. Building a suitable front-end of our job-recommender system web application is crucial for the user experience of the application. If an application is easy, simple to use and visually appealing, the application will increase user engagement, increase website traffic and increase growth.

Building a suitable front-end can be the difference between the success and failure of an application. Front-end web focuses on improving user experience by creating the quality design, functionality and navigation experience of the website. The goal of creating a simple and user-friendly front-end application is so users can easily input their resumes and obtain job recommendations as to the output.

Front-end web application frameworks and libraries are essential for building fast, high-performing and user-friendly web applications. There are various front-end frameworks considered for our job recommender system web application, including Angular, Vue and React. The front-end application will be developed using the following technologies: HTML, CSS and mainly the ReactJS framework. ReactJS is a free and open-source javascript front-end library for building fast and interactive website user interfaces. The reason ReactJS was chosen as the basis for the front-end application is that it is a quick, scalable and easy-to-use front-end web application framework due to its highly efficient and reusable feature components. Currently, ReactJS provides the best possible rendering performance compared to all other front-end frameworks in the market. ReactJS components allow a developer to break down a complex website user interface into smaller components. The ability to break down the user interface into smaller components improves the speed, flexibility, performance and usability of the development process.

ReactJS components are reusable and easily integrable code blocks for building user interfaces. A component is essentially a piece of the user interface, so when building web applications with React, a developer builds independent user components and combines the small components to create the user interface of the system. Every ReactJS application has at least one App component, otherwise known as the root component. The root component contains other child components. Every ReactJS application is essentially a tree of smaller components merged to build a complex UI. A component is implemented as a class with a state and render method; the state is the data displayed when the component is rendered, whereas the render method describes how the UI looks. For the use case of our project, we will be breaking up the front-end application into components App, Resume Input button and a Results Container component which will display the recommended jobs generated from our ML model.

6) Backend Development: The backend of the application is what handles all of the functionality that is not visible to users. It includes the handling of databases, servers, apps, and other functions [15]. The backend framework used for this application was Django, which is a high-level and open-source Python framework [16].

Django is based on the Model-View-Template design pattern. The model acts as the data interface and is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database, which in this project is SQLite3. The View is the user interface; it is what is seen in the browser when the website is rendered. It is represented by the HTML, CSS, and Javascript files. The template consists of static parts of the desired HTML output and some special syntax describing how dynamic content will be inserted.

To automate data transmission between the relational database and the app's model, Django uses a technique called Object Relational Mapping (ORM). ORM automatically creates and stores python object data in the database without having to write SQL queries. Hence, using ORMs allows for more rapid development and lets the project be portable.

Other necessary components of Django are models, URLs, serializers, and views. A model essentially is an object or table to define data fields. The URLs set up the endpoint server that allows developers to view the data. The purpose of serializers is to render the available information into formats that can be easily accessible and used by the frontend. The serializer unpacks the user input objects to be transformed into JSON format to be passed around servers. The View is used to interact with the backend and helps structure code. It takes a request and returns a response. In the Views, the objects are retrieved, serialized, and the data is sent as inputs to any other functions.

In addition, in the Django rest framework, REST APIs were set up to then be used by the ReactJS frontend through making Axios calls (i.e. POST requests)

IV. Alternative Designs

A. Machine Learning Designs

1) TensorFlow Recommenders: TensorFlow Recommenders is a Python data science library built on Keras. It is highly scalable for building web-scale recommender systems. It includes the complete workflow of creating an RS, such as having data preparation, model formulation, training, evaluation, and deployment. It can build and evaluate flexible recommendation retrieval models, freely incorporate item, user, and context information into recommendation models, and train multi-task models that jointly optimize multiple recommendation objectives. The retrieval system is constructed using two-tower retrieval models, converting user input into an embedding and identifying the best options in the embedding space. Individual layers and metrics can be easily customized while ensuring that these individual components work well together.

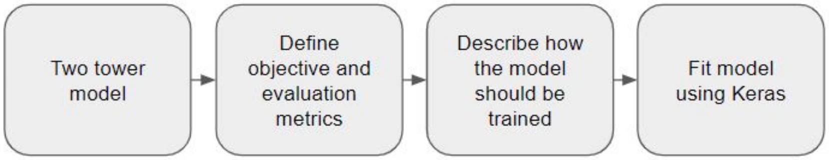

Two tower model
Define objective and evaluation metrics
Describe how the model should be trained
Fit model using Keras

Fig. 8: Flow of events for machine learning driven backend using TensorFlow.

where the two tower model can be further desribed as:

- Turn the defined paramaters into embeddings
- Define our high dimensional vector space
- Define retrieval models using Keras
- Implement TFRS BruteForce layer to validate model recommendations

The implementation of TensorFlow Recommenders, as shown in Figure xx, involves:

1) Setting up user and item representations.
2) Converting raw user ids into contiguous integers by looking them up in a vocabulary
3) Mapping the result into embedding vectors.
4) Having a BruteForce layer to sanity-check the model's recommendations - indexed with precomputed representations of users, and allows one to retrieve top items in response to a query by computing the query-candidate score for all possible candidates.

2) Low Rank Matrix Factorization: This design uses a collaborative filtering algorithm. The main idea is that there are hidden structures in data sets, and by uncovering them, a compressed representation of the data could be created and stored in a matrix. This matrix can be further factored into multiple low-rank matrices. By doing matrix factorization, one could more easily apply other machine learning techniques such as dimension reduction, clustering, etc. The benefits of this technique include simplifying the process of clustering and matrix completion and being a flexible framework that can be applied to various prediction problems.

The implementation of Low-Rank Matrix Factorization would be done by applying the matrix with Content Filtering by assigning the rows being users and columns being the type of job or field of work the user fits the best with. It would collect feature data for each job and map it to the feature, along with mapping feature data from the user to the component itself. By multiplying both matrices using matrix multiplication, the estimated strength of the connection between the users, job posting and type of job (field) would be found. Another way to use Matrix Factorization is to apply Collaborative Filtering. User preference data is generated using Machine Learning and predicting the preference data for both the user's preference and jobs that match the user profile; the machine learning algorithm keeps running until it arrives at a set of numbers that predict data with the lowest errors overall.

3) Topic Modeling using pyLDAvis: Although LDA topic modelling was implemented in the final project, an alternative topic modelling design was also proposed. The implementation steps are shown in Figure xx. This method would use the pyLDAvis library and functions to create a UMAP to visualize the topic modelling. The intertopic distance map can be used, along with a similarity metric, to display the top N results.

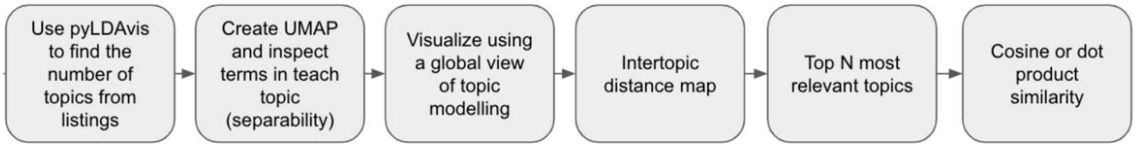

Use pyLDAvis to find the number of topics from listings
Create UMAP and inspect terms in teach topic (separability)
Visualize using a global view of topic modelling
Intertopic distance map
Top N most relevant topics
Cosine or dot product similarity

Fig. 9: Flow of events for topic modeling using pyLDAvis.

B. Backend Designs

1) Python Flask: This framework was proposed as it is easy to understand and ideal for beginners since it is simple to set up and begin using immediately. It allows users to extend on all features offered in the base flask, and almost all parts of it can be changed, unlike in other frameworks. Flask allows for testing with a built-in development server and debugger and a RESTful client, all of which are made lightweight. In general, Flask is suitable for experimentation.

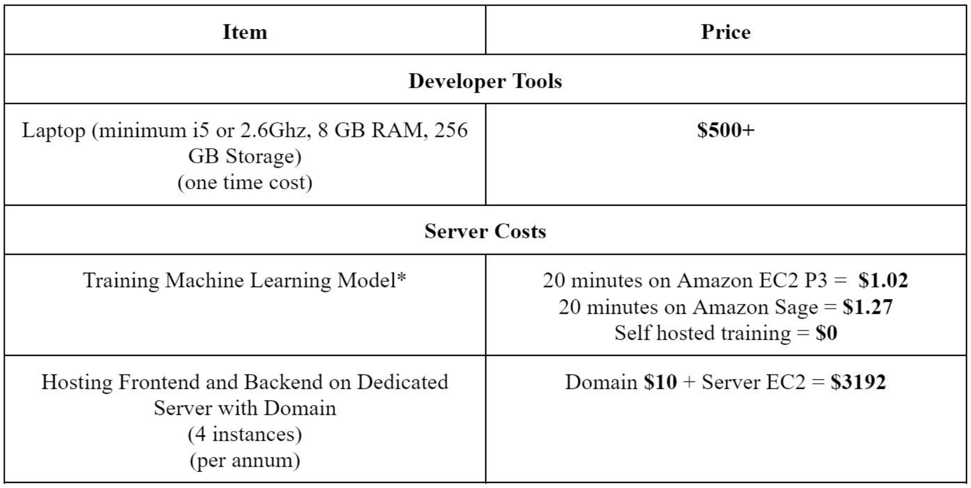
Flask was not implemented due to the fact that an execution-based server that handles requests one at a time may need external modules to help handle this, more modules would increase the application size, and it is not suitable for large applications. In addition, support for Flask is not as large as Django's since the developer community is smaller. Also, in comparison to Django, there is no admin site to take care of the records from the connected database; it does not contain Object Relational Mapping, which would allow for direct communication with the database; it is less standardized and would have slower MVP development.

2) NodeJS: Node.js was another alternative backend technology option. It is used more on client-side applications and focuses on running code on more private servers.

While it can allow for many solutions and solution types, it can also be crippling to new developers as it is not as beginner-friendly as Django or flask. A lot of code must be written by the developer, and they must know which libraries and modules to import and use, in addition to having a good understanding of asynchronous programming. Hence, it takes time to develop applications. The emphasis of Node is on APIs, not a whole web application. It is tough to interface with relational databases, and many developers cite this as a nuisance. Node is well equipped to handle some niche applications because, at its core, it is a server (WebSocket server, file upload, data streaming) instead of a web development framework.

V. Material and Components List

A. Hardware



Item	Price
Developer Tools	
Laptop (minimum i5 or 2.6Ghz, 8 GB RAM, 256 GB Storage) (one time cost)	**$500+**
Server Costs	
Training Machine Learning Model*	20 minutes on Amazon EC2 P3 = **$1.02** 20 minutes on Amazon Sage = **$1.27** Self hosted training = **$0**
Hosting Frontend and Backend on Dedicated Server with Domain (4 instances) (per annum)	Domain **$10** + Server EC2 = **$3192**

*Note: This is the price of retraining the model per dataset, prices will increase as the number of datasets increases. These prices are coming from comparing Amazon offerings on servers with GPU acceleration. The GPU used for the purposes of this project was NVIDIA QUADRO RTX 8000. This is equivalent to Amazon's EC2 P3.2XLarge Servers in terms of performance. Google Cloud Services and Microsoft Azure may have similar services. [17] [18] [19] See Appendix for calculations.

B. Software Libraries and Dependencies

These frameworks and libraries are largely open-sourced and free to use for many applications under the MIT, MPL, Apache, or GPL licenses. The main software components used include:

- Node.js / React
- Python
- MaterialUI
- Django
- Pandas
- NLTK: Natural Language Toolkit
- Sklearn
- Numpy

VI. Measurement and Testing Procedures

In creating this project, although there was not sufficient time to write automated unit tests for the frontend, however, there was time to do manual user-acceptance testing. This type of testing included a system user who would perform the same tasks as the required end-user. This type of testing was carried out during the project's development and allowed developers to identify the UI bugs early on. There are a couple of general guidelines [20] [21] when working to validate a user interface in a web application; the developers have chosen the following criteria:

- User input validation
- Navigational validation
- Aesthetics (spacing, color, CSS etc.)
- Element visibility
- File type validation
- Device Adaptability
- API functionality
- Performance and Response Time

One of the most effective ways to test the specified criteria is a Test Case Plan. Although this is mainly used with unit tests, applying it to manual testing can increase test coverage and quality of testing and improve the overall quality of the application. However, the table also serves as a guideline for the future unit testing that is to be done.

Thus it also includes expected responses. The table below is a representation of the Test Case Plan. It is organized so that each page covers the test criteria where applicable.

Feature	Test Case	Description	Input	Expected
Homepage	Aesthetics	Renders the correct color scheme, divs, buttons and inputs are spaced correctly 3D background is animated as intended	none	Render HTML, CSS
	Visibility	User is able to see different page elements (Buttons, City Input) clearly	none	Proper color contrast between elements and background
	Input Validation	When user wants to search by city, the only valid options are from dropdown menu	Click ’cities’ input	Page renders dropdown menu for city selection with auto-complete
	File Validation	When user uploads a file, the file type is validated for docx	Upload file	Validated: Uploads file to backend APIInvalid: Alert ‘Invalid’ to screen
	Device Adaptability	Look and feel of the screen does not change when switching to a different device. Buttons and fields don’t overlap on smaller devices.	Switch device	Render expected HTML, CSS and spacing
	API Functionality	API returns an expected list of 5 relevant jobs in the city and metadata in expected JSON format	Click ‘Find Matching Jobs’	Backend returns JSON object
	Performance	Page runs smoothly with no lag and API call returns within a reasonable time	Page scrolling + mouse navigation	Page runs smoothly API call returns in <1s
	Navigational Validation	When clicking ‘Find Matching Jobs’ navigates to intended screen	Click ‘Find Matching Jobs’	Moves to Job Listings Page
Job Listings Page	Aesthetics	Renders the correct color scheme Card elements are spaced correctly, 3D background is animated as intended, numbers are animated on cards, cards	none	Render HTML, CSS
	Visibility	User is able to see different page elements (Cards, Buttons, Text) clearly	none	Proper color contrast between elements and background
	Device Adaptability	Cards do not overlap and break page if on another device	Switch device	Render expected HTML, CSS and spacing
	Page Functionality	Each card stores a different job based on information from API call	none	Cards render with correct information
	Performance	Page runs smoothly with no lag	Page scrolling + mouse navigation	Page runs smoothly
	Navigational Validation	Page correctly navigates to intended screen	Click ‘Description’ Button	Moves to Description Page
Description Page	Aesthetics	Renders the correct color scheme, divs and elements are spaced correctly, 3D background is animated as intended	none	Render HTML, CSS
	Visibility	User is able to see different page elements (job description, job title) clearly	none	Saved to page
	Device Adaptability	Text does not become too small on a smaller device for large job descriptions	Switch device	Render expected HTML, CSS and spacing
	Performance	Page runs smoothly with no lag and API call returns within a reasonable time	Page scrolling + mouse navigation	
	Navigational Validation	When clicking ‘Find Matching Jobs’ navigates to intended screen	Click ‘Find Matching Jobs	Moves to Job Listings Page

Other quality assurance checks were done during development, like logging and debugging. This provided visibility into the actual functionality of the code as opposed to the expected functionality of the code.

The ML portion of this project has gone through extensive user-testing, as demonstrated in the next sections.

VII. Performance Measurement Results

Topic modelling using LDA requires training for a specific instance of several topics. As previously mentioned, our training stream consists of 20,000 real job descriptions. One crucial hyperparameter needed for training is the number of topics. The question then arises of how do we determine the number of relevant topics in our corpus. It is not possible to go through the entire database manually and determine the number of topics, as this will likely result in human error of not recognizing unique topics, incorrect assigning, not understanding contextual clues, and time constraints of thoroughly analyzing a database of 20,000 job postings.

To determine this, we deploy two methods, including deciding the coherence plot [22] and the SSE plot [23]. These visualizations help us to understand the uniqueness of topics within our corpus, as well as the quality of topics. A coherence plot measures how different topics are from one another and considers human interpretability. The coherence plot is given by training our model over topics. In this case, we train over 8 to 24 topics. By analyzing the plot, we can see that the highest coherence occurs in 12 topics.

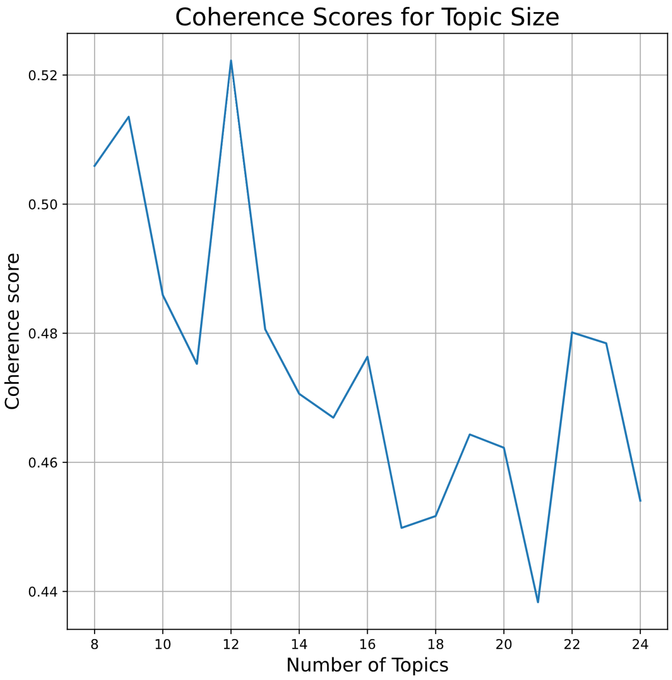


Fig. 10: Coherence scores from 8 to 24 topics, with an approximate 16 hour training time.

To verify our results, we also deploy Sklearn document classification clustering, which uses K-means clustering to group our documents into topics. This plot can be found below. Typically, the elbow method is used to select the optimal number of clusters by drawing a line plot between SSE (sum of squared errors) vs the number of clusters and finding the point representing the “elbow point.” However, we cannot see a defined elbow of the plot and therefore revert to the findings of the highest coherence score, 12. It is important to note that the quality of the outputted topics, as well as the effectiveness of these plots, are susceptible to preprocessing.

It is also crucial to note that by nature, a machine learning model has no understanding of the meaning of words. An LDA model utilizes deterministic probabilities to create topics from words - the goal is not to develop coherent topics for the user; it is to cluster the documents based on their topic-wise and contextual similarity.

Principal component analysis (PCA) [24] and t-distributed stochastic neighbor embedding (t-SNE) [25] can be used to create visualizations to help us understand the quality of the topics. PCA is a mathematical technique to

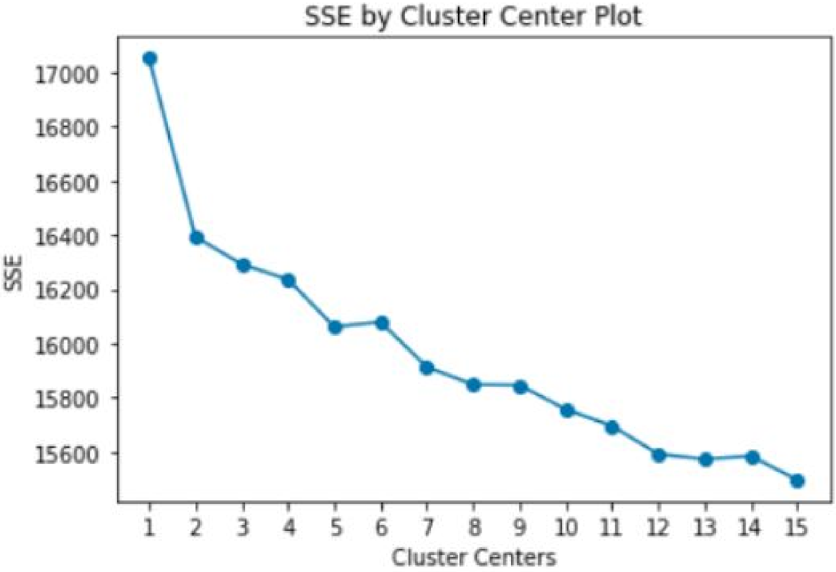


Fig. 11: SSE plot as generated by Sklearns document classification clustering.

create principal components and reduce dimensionality to visualize the separation between clusters, in our case, topics. Visualizing the topic variation is done to strengthen our choice of 12 trainable topics further. Unlike PCA, t-SNE is a probabilistic and statistical method for visualizing high dimensional data by giving each datapoint a location in a two or 3-dimensional space. Both methods are unsupervised and non-linear used for data exploration. Both of these visualizations can be found below.

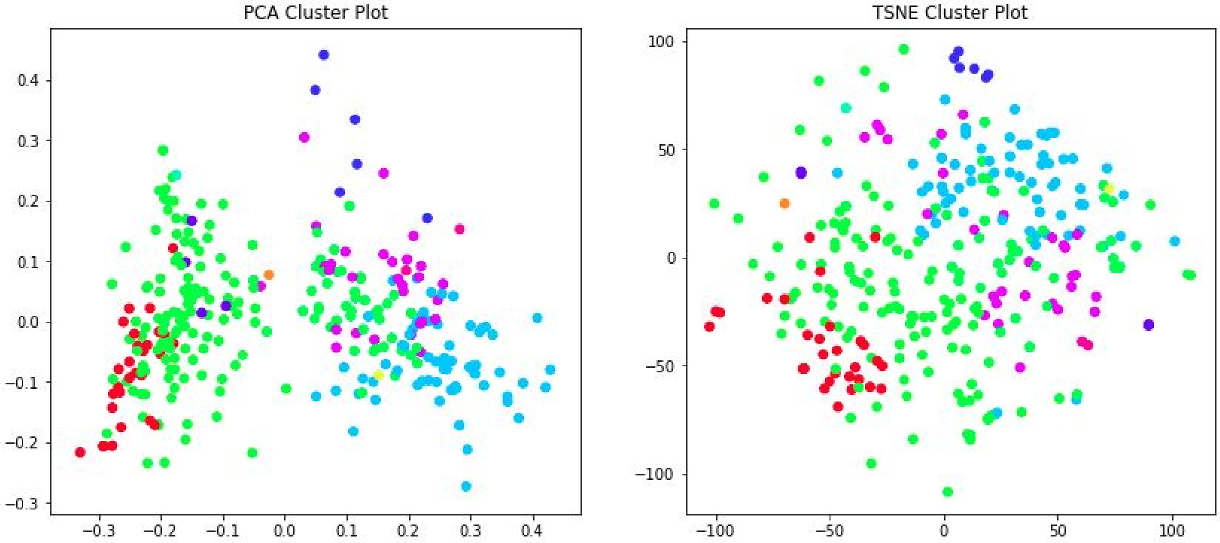
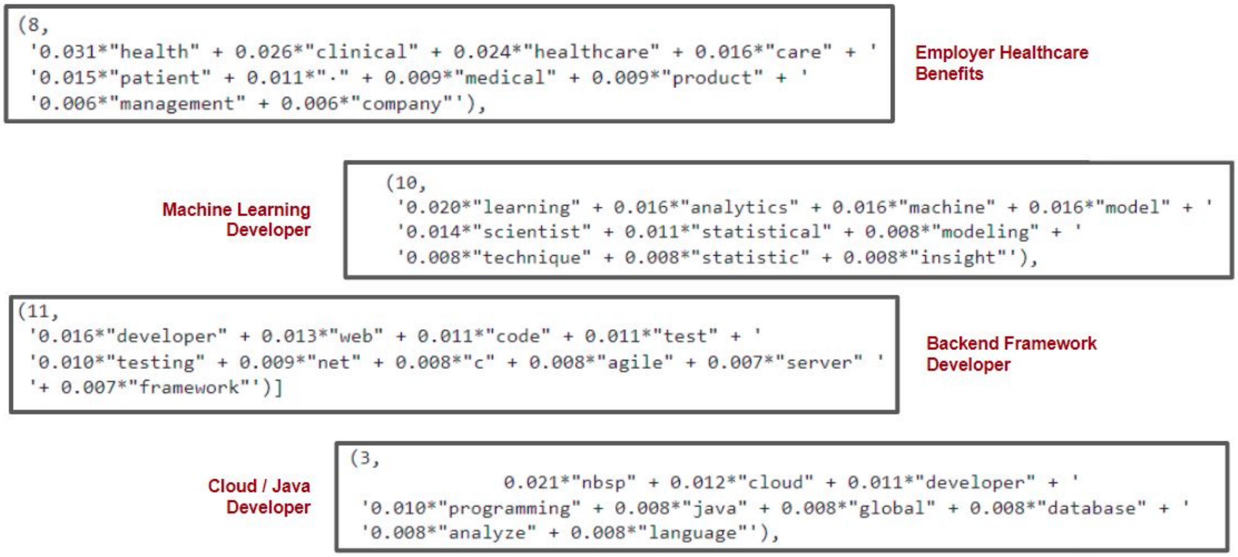


Fig. 12: PCA (left) and t-SNE (right) topic separation visualizations.

The separation of topics as demonstrated by our preprocessing and visualized using the PCA plot is highly effective. Although there is overlap between topics, we can note defined clusters in both plots. As preprocessing continues to perform better through other, more intelligent forms of preprocessing, we expect even more distinct topic separation.

VIII. Analysis of Performance

Latent Dirichlet Allocation (LDA) is an unsupervised learning statistical topic modelling algorithm used to discover abstract topics in a dataset based on the words contained in the document. The LDA model generates topics based on the frequency of words in a set of documents. After the rendering of the LDA model, objects were outputted that contained information on topics that were learnt and collected from the machine learning models analysis. The LDA model discovers these topics within the jobs dataset and outputs them. Below are some example snippets of objects that were outputted from the LDA model. As we can see from the objects, each of the corresponding words shows the dominant topics that are prevalent in the document. The numbers denote the frequency of the word or dominance of the particular topic within the document. The higher the number assigned to the word or topic, the more dominant the topic is within the document.



```
(8,
 '0.031*"health" + 0.026*"clinical" + 0.024*"healthcare" + 0.016*"care" + '
 '0.015*"patient" + 0.011*"·" + 0.009*"medical" + 0.009*"product" + '
 '0.006*"management" + 0.006*"company"'),
```

```
 (10,
  '0.020*"learning" + 0.016*"analytics" + 0.016*"machine" + 0.016*"model" + '
  '0.014*"scientist" + 0.011*"statistical" + 0.008*"modeling" + '
  '0.008*"technique" + 0.008*"statistic" + 0.008*"insight"'),
```

```
(11,
 '0.016*"developer" + 0.013*"web" + 0.011*"code" + 0.011*"test" + '
 '0.010*"testing" + 0.009*"net" + 0.008*"c" + 0.008*"agile" + 0.007*"server" '
 '+ 0.007*"framework"')]
```

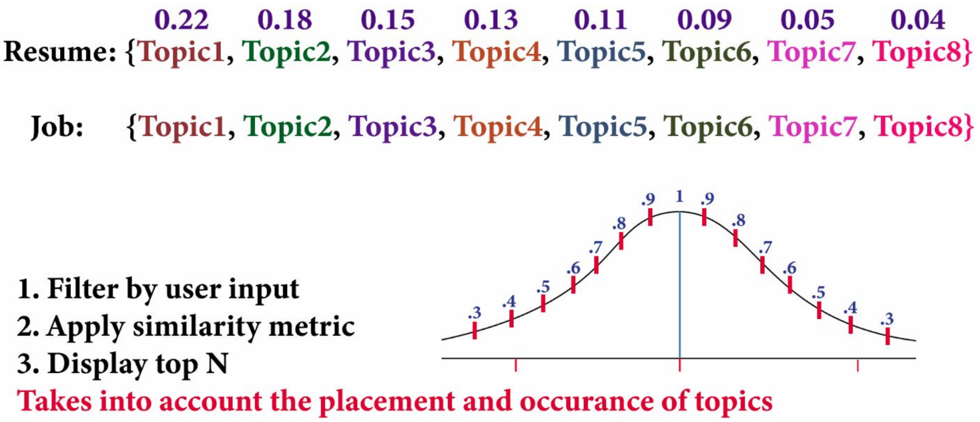
```
(3,
              0.021*"nbsp" + 0.012*"cloud" + 0.011*"developer" + '
 '0.010*"programming" + 0.008*"java" + 0.008*"global" + 0.008*"database" + '
 '0.008*"analyze" + 0.008*"language"'),
```

Fig. 13: The outputted topics do not need to be user interpretable, however, we can manually assign labels.

Reducing the jobs data set corpus was done to analyze and test the performance of the LDA model. This was done to improve the run time performance and provide a more accurate set of results for the user. This reduction of corpus script works based on a user's preference for jobs recommendations from a specific country, state and city. The selected values would then be used to filter out and clean the data set for the training of the LDA algorithm. Training the LDA model with the original jobs dataset of 20,000 technology jobs took an average of 10+ hours to compute 20 LDA models, approximately 15-20 minutes of training time per LDA model. After implementing the corpus reduction script, training time was significantly reduced to only 15-20 minutes to train an LDA model over one topic instance.

Although the LDA model can accurately extract topics from a given document, the point is to contextualize the data processed for the user. Users are looking to find the degree to which their resume is similar to the job description document. To do this, we propose a number called the 'similarity metric,' a number between 1-100, that indicates document similarity that is computed by a similarity algorithm. This algorithm must consider the importance of topics (how many times they occur) and the relative importance of the topics to the other document (how important it is in one document vs. the other). The similarity algorithm is best explained by the following diagram

As mentioned before, a total of 8 topics are extracted from both the resume and job description. They are then presented in order of importance from left to right, i.e. the topics on the left can be said to occur more frequently than the topics to the right. The purple numbers at the top would represent the ideal similarity score, i.e. if the two documents had the same topics and order. If Topic 1 is both the same topic and placement in both resume and job, a similarity score of 0.22 is assigned. Coincident Topic 2's have a score of 0.18 and so forth. We can then add these


0.22 0.18 0.15 0.13 0.11 0.09 0.05 0.04
Resume: {Topic1, Topic2, Topic3, Topic4, Topic5, Topic6, Topic7, Topic8}
Job: {Topic1, Topic2, Topic3, Topic4, Topic5, Topic6, Topic7, Topic8}
.3 .4 .5 .6 .7 .8 .9 1 .9 .8 .7 .6 .5 .4 .3

1. Filter by user input
2. Apply similarity metric
3. Display top N

Takes into account the placement and occurance of topics

Fig. 14: Visualization of similarity score weight distributions.

numbers for a perfect score of 1 for two identical documents. Furthermore, ideal scores in purple exponentially decay as the topic moves further to the right, modelling that topics additionally from Topic 1 become less critical.

For example we can compute the similarity score for two identical documents,

Resume Topics: [Learning, Analytics, Machine, Model, Scientist, Statistical, Modeling, Numpy]

Job Topics: [Learning, Analytics, Machine, Model, Scientist, Statistical, Modeling, Numpy]

Similarity Score: 0.22 + 0.18 + 0.15 + 0.13 + 0.11 + 0.09 + 0.05 + 0.04 = 1

We then map to a scale between 1 and 100 for the user. In this case, the score is 100.

However, most documents will have non-similar topics and differing frequencies of topics. To achieve this, a correction multiplier is applied. This correction multiplier is modelled by the function in yellow; it ensures that if two topics are similar between the resume and job, it can compute the distance between them and proportionally reduce the ideal score. If Topic 1 is in the first position in the resume, but in the job description, this same topic exists as Topic 4, we must reduce the score of similarity by a factor of 0.7, 3 spaces away from its original position. Thus the score then becomes 0.7 × 0.22 = 0.514. This needs to be done across all topics in both documents.

To give an example we can reorder the topics from the last example.

Resume Topics: [Learning, Analytics, Machine, Model, Scientist, Statistical, Modeling, Numpy]

Job Topics: [Modeling, Analytics, Model, Machine, Learning, Numpy, Scientist, Statistical]

Learning: 0.22 × 0.6 = **0.132**

Analytics: 0.18 × 1 = **0.18**

Machine: 0.15 × 0.9 = **0.135**

Model: 0.13 × 0.9 = **0.117**

Scientist: 0.11 × 0.8 = **0.088**

Statistical: 0.09 × 0.8 = **0.072**

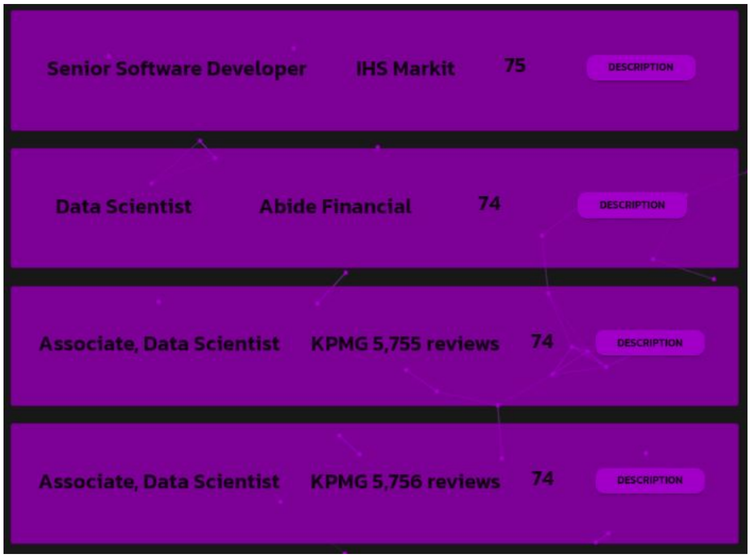
Modeling: 0.05 × 0.4 = **0.02**

Numpy: 0.04 × 0.8 = **0.032**

Similarity Score: 0.132 + 0.18 + 0.135 + 0.117 + 0.088 + 0.072 + 0.02 + 0.032 = 0.776

Which we can then map to a score of 78. This similarity metric is the underlying idea behind the testing below. The idea is that generally, we would like to see resumes of a certain kind match the role to which their skills are most closely aligned.

To check if our machine learning LDA algorithm and similarity metric was performing as expected, further testing/analysis of the project was done by feeding our LDA model various input resumes. This was done to test and analyze the behaviour and robustness of our machine learning LDA model and see if it would perform consistently and still be stable even in the case of input resumes that are curated to different subfields within technology. This would also test the similarity metric as we would expect to see various jobs presented to other resume inputs. Due to the 20,000 jobs data set containing technology jobs, we tested the LDA model with two different resumes that were designed for specific different technology jobs in Data Science and Product management.



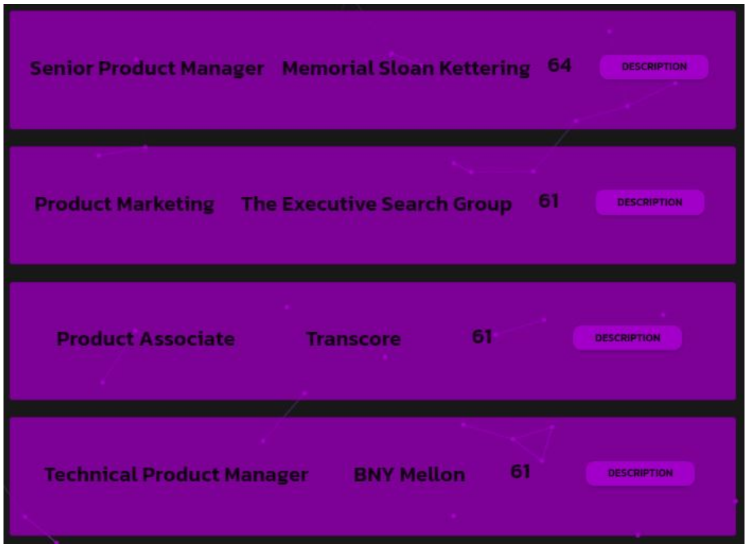
Senior Software Developer IHS Markit 75

Data Scientist Abide Financial 74

Associate, Data Scientist KPMG 5,755 reviews 74

Associate, Data Scientist KPMG 5,756 reviews 74

Fig. 15: Final outputted jobs for a data science specific resume with software development qualifications.



Senior Product Manager Memorial Sloan Kettering 64

Product Marketing The Executive Search Group 61

Product Associate Transcore 61

Technical Product Manager BNY Mellon 61

Fig. 16: Final outputted jobs for a product developer resume.

IX. Conclusion

In conclusion, all of the main objectives for this machine learning-driven job recommender system matching a user's resume to the most relevant jobs in the database have been met. A database storage system was made to store the 20,000 jobs dataset, an LDA model was implemented and trained from the dataset, and a web application was developed to train a user's resume to find their top 5 best jobs.

Some minor discrepancies between the initial project plan and the result were that no real-time database was implemented. The jobs dataset is limited to software engineering and data science jobs. The lack of a real-time database was due to issues encountered during web scraping (which will be expanded on shortly). Similarly, the limitations of the jobs dataset career fields were also due to problems with web scraping being unsuccessful.

Thus, the significant unresolved difficulties of this project involve web scraping, analyzing preprocessing, and not finding an accurate way to measure the ground truth of the LDA model or an alternative method of measuring the LDA model's accuracy. Several methods were tried when attempting to web scrape, such as implementing the selenium tool and running the Octoparse tool. The issue was that the web scraping was attempted on the LinkedIn API, which is highly secure and difficult for ordinary users to access. LinkedIn was ultimately blocking the scraping of data from their API. Another unresolved issue was that the SSE plot did not indicate an elbow to determine the number of topics for which to train the LDA model. This signals that preprocessing needs to be improved; however, it was difficult to analyze the preprocessing performance objectively. As a result, it was difficult to determine how to improve its quality. In addition, although a cosine similarity algorithm was implemented to measure the ground truth, it remains an inaccurate way of measuring ground truth since the cosine similarity algorithm itself was not developed objectively. Hence, more research needs to be done to determine how to measure the accuracy of LDA models. Despite these difficulties, a functional system was still implemented and can be built upon.

A few key additions could be made to improve this intelligent job recommender system further and generate more real-world utility for users. First, a real-time database would have to replace the current static and outdated 20,000 jobs dataset to make the job matches more relevant. This real-time database would be updated once a day, and its data would be generated by web scraping suitable job posting information found through LinkedIn or GoogleJobs. To increase the scope of what this system can accomplish, the jobs dataset would not be limited to software engineering and data science jobs. All other fields of work would be added. Finally, to allow this system to be used by recruiters in addition to job seekers, another real-time database would be implemented. This database would store data from performing web scraping on people's LinkedIn profiles, then train the LDA model on this dataset to find people's topics and weight distributions. After that, the specific job posting would be trained on the model to see its topics and respective weights. Finally, the similarity metric algorithm would be applied between the information extracted from the candidate's dataset and the job posting to determine the best candidates for the position. Hence, many features can be added to this job recommender system for more excellent utility. In addition, the model can also be semi-supervised to obtain user feedback during the training process to improve accuracy further. Preprocessing may also be enhanced by implementing it as another ML model. For example, the number of stopwords can be learnt through unsupervised approaches. Although time-consuming, these changes will enormously enhance the model and reduce the training time of the LDA model.

In summary, the primary goal of this project of recommending ideal jobs to job seekers has been a success. Although difficulties have been encountered, practical problem-solving skills were employed to make necessary changes and adjustments. Furthermore, this job recommender system can be expanded and built on to provide more valuable services for job seekers and recruiters.

References

[1] Z. Fayyaz, M. Ebrahimian, D. Nawara, A. Ibrahim, and R. Kashef, "Recommendation systems: Algorithms, challenges, metrics, and business opportunities," *applied sciences*, vol. 10, no. 21, p. 7748, 2020.

[2] Z. Tong and H. Zhang, "A text mining research based on lda topic modelling," in *International Conference on Computer Science, Engineering and Information Technology*, 2016, pp. 201–210.

[3] J. Jacoby, "Perspectives on information overload," *Journal of consumer research*, vol. 10, no. 4, pp. 432–435, 1984.

[4] H. Jelodar, Y. Wang, C. Yuan, X. Feng, X. Jiang, Y. Li, and L. Zhao, "Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey," *Multimedia Tools and Applications*, vol. 78, no. 11, pp. 15 169–15 211, 2019.

[5] "Tensorflow recommenders," https://www.tensorflow.org/recommenders, accessed: 2022-02-11.

[6] "pyldavis," https://pyldavis.readthedocs.io/en/latest/readme.html#:~:text=pyLDAvis%20is%20designed%20to%20help,an%20interactive%20web%2Dbased%20visualization., accessed: 2022-02-11.

[7] J. Abernethy, F. Bach, T. Evgeniou, and J.-P. Vert, "Low-rank matrix factorization with attributes," *arXiv preprint cs/0611124*, 2006.

[8] "What is a database?" https://www.oracle.com/ca-en/database/what-is-database/#WhatIsDBMS, accessed: 2022-02-11.

[9] "An introduction to transactional databases," https://www.mongodb.com/databases/types/transactional-databases, accessed: 2022-02-11.

[10] "How sqlite works," https://www.sqlite.org/howitworks.html, accessed: 2022-02-11.

[11] "Architecture of sqlite," https://www.sqlite.org/arch.html, accessed: 2022-02-11.

[12] "Business logic. investopedia," https://www.investopedia.com/terms/b/businesslogic.asp, accessed: 2022-02-11.

[13] C. Kambalyal, "3-tier architecture," *Retrieved On*, vol. 2, no. 34, p. 2010, 2010.

[14] "Http request methods," https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods, accessed: 2022-02-11.

[15] "What is backend development: Skills, salary, roles more — simplilearn," https://www.simplilearn.com/tutorials/programming-tutorial/what-is-backend-development#:~:text=Backend, accessed: 2022-02-11.

[16] "Django," https://www.djangoproject.com/, accessed: 2022-02-11.

[17] "Amazon ec2 p3 instances," https://aws.amazon.com/ec2/instance-types/p3/, accessed: 2022-02-11.

[18] "Amazon sagemaker pricing," https://aws.amazon.com/sagemaker/pricing/?nc=sn&loc=3, accessed: 2022-02-11.

[19] "Amazon ec2 on-demand pricing," https://aws.amazon.com/ec2/pricing/on-demand/, accessed: 2022-02-11.

[20] "Ui testing: A beginners guide," https://www.testim.io/blog/ui-testing-beginners-guide/, accessed: 2022-02-11.

[21] "Ui testing: A comprehensive guide," https://www.perfecto.io/blog/ui-testing-comprehensive-guide, accessed: 2022-02-11.

[22] D. Mimno, H. Wallach, E. Talley, M. Leenders, and A. McCallum, "Optimizing semantic coherence in topic models," in *Proceedings of the 2011 conference on empirical methods in natural language processing*, 2011, pp. 262–272.

[23] C. Shi, B. Wei, S. Wei, W. Wang, H. Liu, and J. Liu, "A quantitative discriminant method of elbow point for the optimal number of clusters in clustering algorithm," *EURASIP Journal on Wireless Communications and Networking*, vol. 2021, no. 1, pp. 1–16, 2021.

[24] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[25] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[26] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[27] "Tf idf: Tfidf python example," https://towardsdatascience.com/natural-language-processing-feature-engineering-using-tf-idf-e8b9d00e7e76, accessed: 2022-02-11.

[28] "Sklearn feature extraction text tfidfvectorizer," https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html, accessed: 2022-02-11.

[29] "Latent dirichlet allocation: Intuition, math, implementation and visualisation with pyldavis," https://towardsdatascience.com/latent-dirichlet-allocation-intuition-math-implementation-and-visualisation-63ccb616e094, accessed: 2022-02-11.

[30] "React," https://reactjs.org/, accessed: 2022-02-11.

[31] "What is react?" https://www.w3schools.com/whatis/whatis_react.asp, accessed: 2022-02-11.

X. Appendix

Amazon calculations:

20 minutes on Amazon EC2 P3: For a p3.2xlarge

3.06 $/hr 0.333 hrs (20 mins) = $1.02/20 min runtime

20 mins on Amazon Sage EC2 P3: For a ml3.2xlarge, Accelerated ML optimized version of the p3.2xlarge

3.825 $/hr 0.333 hrs (20 mins) = $1.27/20 min runtime

Hosting Frontend and Backend on Dedicated Server with Domain (per annum)

Average Domain Price

$10

EC2 Server Cost with Specifications

Type: t4g.large

OS: Linux

vCPUs: 4

Memory: 16 GB

Number of Instances: 4 (spread across different locales and for load balancing)

Storage: 50 GB (extra space for live updates)

Total Price: $266/monthly $3192/annum