Capstone Project: LIRecommend

In this document, I will give a summary of some different resources and articles I've found that deal with very similar problems to the one in this project.

I will discuss how they relate to what I'm doing, what valuable things I learned from them, and how I might leverage what I learned. I will be attempting to reproduce available solutions and running trials from the guides which spell out code in this jupyter notebook: Without further ado I list the resources and then I will talk about each one.

This is the first article/guide I found that could be really helpful as a resource for my project:

https://medium.com/@abbasbehrain95/creating-an-ai-powered-job-recommendation-system-50ce1cd12d36

And this is another interesting one:

https://towardsdatascience.com/building-a-job-recommender-for-non-technical-business-roles-via-nlp-and-machine-learning-626c4039931e

---

And somewhat interesting tiny thread:

https://community.openai.com/t/job-recommendation-system/647428

As far as techniques that could be used for exploratory data analysis and visualization in this project, there's a lot of stuff in this Kaggle notebook that boils down to basically the same thing that I am or will be doing, particularly since it too deals with a job-related corpus. I am doing something similar to what was done in this notebook, and after investigating I should ultimately visualize some of the insights returned from the phrase vectorization and clustering work I've done:

https://www.kaggle.com/code/liliyak/job-recommendation-analysis.

I reproduce results/follow along with the code provided in this Kaggle notebook using the provided input data files, in my own Jupyter notebook

LIRecommend-ReproduceAvailableSolutions.ipynb which I will submit for this assignment.

The following is an article about MD Job Genie, which is an online tool that was adopted into Maryland's existing workforce services portal. It is an AI-based career recommendation engine that was trained on five years of unemployment insurance tax records in order to make best-fit suggestions to users. The article is rather interesting: https://www.route-fifty.com/infrastructure/2023/02/ai-powered-career-recommendation-engine-delivers-more-job-options/382792/

With regard to the first resource linked from Abbas Behrain, I wasn't able to reproduce the author's results with the code provided, but it is reassuring to know that Abbas had a similar idea in mind to build a job recommendation tool based on job posting data. As stated in the headline, Abbas lays out a "Job Recommendation System using Scraped Glassdoor Data, Machine Learning Techniques, and Streamlit Application" he made. His intro states:

> " In today's competitive job market, finding the right career opportunities that align with our interests and skills can be challenging. However, with the advancement in AI/ML technologies, we can leverage data-driven systems to provide personalized job recommendations based on user preferences and historical job data. This article will explore the technical aspects of building a Job Recommendation System using data scraped from Glassdoor and creating a Streamlit application."

This is very similar to a pitch I would envision for my own Capstone project. Except I will be scraping LinkedIn Jobs postings to generate my dataset and I am not sure that I will be making a Streamlit application, although it might be a nice way to tie up my project, and is something my mentor has suggested as well.

The project overview also states that cloud deployment was used. This is yet another consideration of my project. Will a cloud deployment be necessary and if so how should it be done?

Again, “the system collects job data from Glassdoor, preprocesses and cleans the data using custom components, and then applies machine learning algorithms to generate personalized recommendations**”.** To obtain some insight and takeaways I can leverage for my own project, I will look at each component of this as it draws a parallel to my project.

1. Collects job data from Glassdoor:

jd_data_extractor.py: Using the Selenium driver, this component extracts job data directly from Glassdoor. It automates the data collection process, enabling us to gather a comprehensive dataset of job descriptions for analysis and recommendation generation.

I won’t copy the code here, but I will discuss main takeaways from looking at jd_data_extractor. Unfortunately I haven’t yet been able to get the code to work to inspect the data at each stage of the scraping process to verify my understanding is correct. But here is how I understand it:

A. The extractor code uses a Selenium Chrome web driver to get the page source as html, which is then parsed with beautifulsoup for links which are appended to a list of urls. This list of urls is then dumped into a JSON file. Thus, I suspect the main page which he has hard-coded is a landing page for a specific search in Glassdoor jobs. He specifies the search criteria with ‘locid’ and ‘key’ when he calls `openbrowser(locid, key)` which then calls

```
driver.get("https://www.glassdoor.co.in/Job/jobs.htm?suggestCo
unt=0&suggestChosen=true&clickSource=searchBtn&typedKeyword={}
"
"&sc.keyword={}&locT=C&locId={}&jobType=fulltime&fromAge=1&rad
ius=6&cityId=-1&minRating=0.0&industryId=-1"
"&sgocId=-1&companyId=-1&employerSizes=0&applicationType=0&rem
oteWorkType=0".format(txt[:-1], txt[:-1], locid))
```

Thus the JSON file will contain urls for different job postings yielded from the search on Glassdoor Jobs.

B. Once this JSON file containing posting URLs is generated, an empty dataframe is created, and the list of URLs is iterated over to scrape data from each job posting again with selenium and using beautiful soup like so:

```
for u in tqdm(url):
    driver.wait = WebDriverWait(driver, 2)
    driver.maximize_window()
    driver.get(u)
    soup = BeautifulSoup(driver.page_source, "lxml")
    try:

        header = soup.find("div",{"class":"header cell info"})
        position = driver.find_element_by_tag_name('h2').text
```

```
        company =
driver.find_element_by_xpath("//span[@class='strong
ib']").text
        location =
driver.find_element_by_xpath("//span[@class='subtle
ib']").text
        jd_temp =
driver.find_element_by_id("JobDescriptionContainer")
        jd = jd_temp.text
        info = soup.find_all("infoEntity")
    except IndexError:
        print('IndexError: list index out of range')
    except NoSuchElementException:
        pass
    data[i] = {
        'url' :u,
        'Position':position,
        'Company': company,
        'Location' :location,
        'Job_Description' :jd
    }
    i+=1
    data[i] = {
        'url' :u,
        'Position':position,
        'Company': company,
        'Location' :location,
        'Job_Description' :jd
    }
    i+=1
```

From this we see that Behrain extracts data in the form of 5 features for each Glassdoor job posting that he ends up adding to the empty dataframe he created before starting the loop. Then this dataframe gets saved to a csv file called jd_unstructured_data.csv locally.

C. In this jd_data_cleaner code Behrain also has a function get_jobs() which does not actually leverage beautifulsoup but instead uses only built-in Selenium functions to do button clicks and page navigation, and to match elements of the page source to extract the following information on a specified number of jobs, again yielded by a search on Glassdoor Jobs:

Company_name
Location
Job_title
Job_description
Salary_estimate

Rating
Headquarters
Size
founded
Type_of_ownership
Industry
Sector
Competitors

The italicized fields come from data found in the 'Company' tab of the job posting page and are not always available. The fields in red are also not always available but do not come from a separate tab. The first four fields listed which are not in red or italicized are assumed to always be available. Behrain employs many try and except catches in his code to make sure all possible scenarios are met and everything runs smoothly. The information is passed into a dictionary object for each job posting, and this dictionary object is appended to a list which gets passed as the data to a dataframe, which also gets saved as a csv locally to jd_unstructured_data.csv.

I am not exactly sure why there are two different methods in jd_data_extractor.py that create a jobs dataframe that gets saved to the same .csv file. I suppose that the get_jobs function could replace the need for calling the geturl(driver) function followed by the code that iterates over the posting urls and scrapes data for each one, since it uses button clicks and page navigation as well as Selenium functions to parse the page sources. I'm saying it's possible there is extraneous code provided in this Python file, and that maybe some of the code should be commented out. But I could be wrong, and we will get a better clue as to what's going on when we look further down the pipeline and at the other code to see which functions actually get run.

2. "preprocesses and cleans the data using custom components"

jd_data_cleaner.py: This component is responsible for cleaning the job description data scraped from Glassdoor. It handles tasks such as removing null values, converting data types, and ensuring data consistency. By preprocessing the data, we can enhance the accuracy and reliability of our recommendations.

A. This file contains functions called "convert_salary", "convert_revenue", and "convert_size" for processing the 'Salary Estimate', 'Revenue', and 'Size' columns of our jd_unstructured_data dataframe as well as lambda functions applied to our 'Job Description' and 'Company Name' columns for cleaning and processing the text. More specifically, the following code is run to collect all words in the job description that are not English stopping words and that have more than 2 characters.

```
unstructured_df['Processed_JD']=unstructured_df['Job
Description'].apply(lambda x: ' '.join([word for word in str(x).split() if
len(word)>2 and word not in (stopw)]))
```

Here, stopw = set(nltk.corpus.stopwords.words('english').

B. The following libraries are imported:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
from nltk.corpus import stopwords
```

C. The mean value of the processed size, average salary, and average revenue columns is calculated, and then all null values in those columns are replaced with the mean for the column.

D. Once the unstructured_df is fully processed including the original `'Unnamed: 0','Salary Estimate','Revenue','Job Description'` columns getting dropped, it is used to locally save a new csv file called jd_structured_data.csv.

3. "applies machine learning algorithms to generate personalized recommendations"

job_recommender.py: "It utilizes the TF-IDF vectorizer from the scikit-learn library to transform job descriptions and user preferences into numerical feature vectors. These vectors capture the importance of each word in the documents, enabling the system to find similar job opportunities based on user preferences. The Nearest Neighbors algorithm is then used to identify the most relevant job recommendations."

A. This code imports a custom module called skills_extraction (which I will discuss in the next section) to parse his resume and store all his skills in a list called 'skills'.

B. He defines a function called ngrams which defaults to n=3 that returns ngrams (it defaults to tri-grams) of a string after standardizing and removing unwanted characters from the string.

C. This ngrams function is passed as the value to the analyzer parameter of a TfidfVectorizer that he instantiates, like so:

```
vectorizer = TfidfVectorizer(min_df=1, analyzer=ngrams, lowercase=False)
tfidf = vectorizer.fit_transform(skills)
```

And that he then fits on the skills list.

D. A NearestNeighbors algorithm is then fit to the tfidf vector data with n_neighbors = 1. Behrain defines a function called 'getNearestN' which

returns two lists, ‘distances’ and ‘indices’, from a list of documents. These contain information corresponding to the distance and index of the vector representation of each job posting with respect to the vector representation of ‘skills’.

E. Then Behrain iterates over each posting in order of index and gets the distance of the posting from the vector representation of ‘skills’. He makes a list called ‘matches’ that he appends each distance to and eventually he creates a new column in the jd_df dataframe, read from jd_structured_data.csv, called ‘match’, which stores the distance value. He then uses this as a measure of match confidence and can give top recommendations by sorting jd_df with this column’s values from smallest to largest.

F. The following modules are imported:

```
import re
from ftfy import fix_text
from sklearn.feature_extraction.text import TfidfVectorizer
import re
from sklearn.neighbors import NearestNeighbors
import numpy as np
import pandas as pd
import nltk
from nltk.corpus import stopwords
stopw  = set(stopwords.words('english'))
from pyresparser import ResumeParser
import os
from docx import Document
import src.notebook.skills_extraction as skills_extraction
```

Skills_extraction.py:

A. This file uses the spacy library to create a matcher and define match patterns for skills read from Behrain’s skills.csv file.

B. Then Behrain uses PdfReader from PyPDF2 to extract the text from his PDF resume and find matches to the list of skills from the .csv file. He returns this list of matches in the skills_extractor function which gets called on his resume ‘CV.pdf’ in job_recommender.py.

Streamlit Application:

“To make the job recommendation system easily accessible and user-friendly, we have developed a Streamlit application. Streamlit provides an intuitive web interface where users can upload their resumes. The application processes the user input, applies the

machine learning models, and displays the top-recommended jobs based on the user's preferences and historical data."

__init__.py (streamlit app):

A. Imports the following:

```
import streamlit as st
import pandas as pd
import PyPDF2
from pyresparser import ResumeParser
from sklearn.neighbors import NearestNeighbors
from src.components.job_recommender import ngrams,getNearestN,jd_df
import src.notebook.skills_extraction as skills_extraction
from sklearn.feature_extraction.text import TfidfVectorizer
```

B. This file consists of two functions: 'process_resume' and 'main'

a. 'process_resume' does the skill extraction from the user's resume using the code defined in skills_extraction.py. It fits the TFIDF ngram vectorizer to the returned skills list and then calculates closest neighbors for all postings in the jd_df dataframe, which is imported from src.components.job_recommender. Process_resume returns the top 5 rows in the dataframe sorted by match confidence from smallest to largest (remember the confidence value actually represents a distance).

b. 'main' is the function that runs the streamlit app, defining a title for the application, as well as a prompt that says `"Upload your resume in PDF format".` It also uses built-in stream-lit methods, file_uploader, write, and dataframe, to allow the user to easily upload files and have the results of the recommendation engine displayed directly as a dataframe.

Cloud Deployment with Azure:

"To ensure scalability and availability, we have deployed the job recommendation system using the Azure cloud platform. Azure provides a robust infrastructure that allows us to host our application and handle user requests efficiently...

The deployment process involves the following steps:

- Model Serialization: Serialize the trained model to a format compatible with the Azure cloud deployment.

- Model Containerization: Package the serialized model along with the necessary dependencies and environment specifications into a container using tools like Docker.
- Azure Container Registry: Create a container registry on Azure to store the model container and related artifacts securely.
- Azure Kubernetes Service (AKS): Deploy the model container as a scalable microservice using AKS, which provides orchestration and management capabilities.
- API Development: Develop an API that allows users to interact with the deployed model and request personalized job recommendations.
- Integration and Testing: Integrate the API with other components of the job recommendation system, and perform thorough testing to ensure its functionality and performance.
- Deployment Monitoring: Monitor the deployed model and API to track usage, and performance metrics, and address any potential issues or errors."

Key takeaways from this first resource from Behrain:

- ***It looks like Behrain came up with a method for scraping job data from Glassdoor that could work in general, but he used it to only scrape data from a predetermined set of jobs, and the tool does not do a new query on Glassdoor for each user. This means that the user will not get the most up-to-date recommendations and the tool will only recommend jobs that were included in the original jd_unstructured_data.csv file that Behrain generated. It's likely that many of these open job postings also got closed, meaning they are no longer accepting applicants.***
- ***Whereas I have framed a supervised learning problem, Behrain framed an unsupervised learning problem where his predictions (aka recommendations) are based purely on proximity of a pregenerated set of job postings to the user's resume in terms of Behrain's TFIDF vectorization with custom n-gram tokenization. It is indeed a neat idea for a tool, but the intent behind the tool is fundamentally different from what I intend for my tool.***

- *There are elements to the system he built which I find applicable and useful for my own purpose, however there are also elements of what he built that I question from a practical standpoint or that inherently differ from the problem(s) I am trying to solve. I will discuss further.*

WHAT I LIKE:

- *Behrain has automated the data collection process to some degree by using code to find links from a parent page on Glassdoor Jobs. This way he was able to add many jobs to his dataset quickly. Even though this would be nice for me to be able to do, I can't because the problem I am trying to solve is fundamentally different. What I am trying to do is supervised learning, and what Behrain does with his system is unsupervised learning. This means he does not have to generate labels for any of his data. He can automatically collect data. Currently, the goal behind this project is to enhance LinkedIn Job recommendations by improving matching/filtering by modeling job desirability for different users, which involves a new feedback mechanism involving a 0-3 star rating of the job posting by the user, rather than just saving or liking the job. (Thus, to prove out the whole mechanism behind the recommendation system I built, I had to curate a dataset with manual labels based on my own job preferences, one that is a good enough representative sample of all jobs that exist.) Even though curating the dataset for this project requires manual work to generate labels for each datapoint, this type of automation could still prove extremely useful in my project to be able to easily evaluate the recommendation system on unseen jobs (that can get automatically collected). I may also consider incorporating more techniques from unsupervised learning besides clustering (which I have already employed in my feature engineering to make categories for the set of phrases in the dataset) into the final tool to bolster accuracy and possibly even make the recommendations more interesting, by using collaborative-filtering. But so far neither a content-based or collaborative filtering recommendation system in their pure forms have been tried; what I have been able to do so far is similar to a content-based recommendation system, but it is not exactly the same thing because our predictions don't boil down to distance or similarity scores but more complex decisions being made by a tree-based algorithm (we've found that so far Random Forest gives the best accuracy). I also tried linear regression which showed promise but gave wildly fluctuating results depending on details in the feature engineering process.*
- *Behrain has built a streamlit app and provided the code for doing so, which will prove helpful should I choose to do the same thing. Building a streamlit app seems like a great and easy way to make the tool available to the public.*

- ***Without having tested it for myself, in theory Behrain's tool could provide job recommendations/matches to users that are more closely aligned with their current skill sets stated on their resumes than a platform like Glassdoor would if it doesn't use pattern matching and other techniques from NLP to extract info about the user from the resume itself, rather than just data from the user profile itself. However it's possible Glassdoor combines both.***

WHAT I DON'T LIKE:

- ***The code provided by Behrain didn't work for me even after some troubleshooting and trial-and-error, so unfortunately I wasn't able to see his mechanism in action.***
- ***Behrain recommends jobs from an already collected set stemming from a parent page of Glassdoor jobs of a specific search containing criteria like job title 'data scientist' and location 'bangalore'. Again this means that the user will only be recommended jobs from a pre-generated pool, which is likely a small percentage of jobs that exist in a specific region of the world at some past date, and most of these job postings probably closed to applicants soon after Behrain generated the dataset for his tool. However, if the released version of the tool generates new searches in Glassdoor periodically to automatically scrape jobs and frequently get updated versions of jd_structured_data from which to recommend fresh jobs, then Behrain has addressed this issue that I am seeing.***
- ***Behrain scraped data about each job besides just job description text that is useful, such as salary info, location, and company info, and yet these features end up being ignored in the final model which only looks at the proximity score of the custom TFIDF representation of the user's skills parsed from their resume with the custom TFIDF representation of the job description text. So it seems like the extra data he scraped is just there for him to be able to display to the user once the recommendation is made and is not used by the AI.***
- ***On the topic of resume parsing, I'm not convinced it is even that useful considering Glassdoor and other platforms parse your resume already.***
- ***This tool recommends jobs that the user is most qualified for or the best fit for, but not necessarily the jobs that the user wants most. The intent behind my project is not to help users identify jobs that they are most qualified for but rather jobs that they would most desire. I claim that a lot of the time people don't want the jobs they are most qualified for. And I claim that platforms like LinkedIn and Glassdoor could do a better job of modeling a user's desire rather than just qualifications with respect to a given job. And to do this I propose obtaining feedback from users on jobs in the form of a 0-3 rating rather than doing content-based or collaborative filtering-based recommendations on saved or liked jobs.***

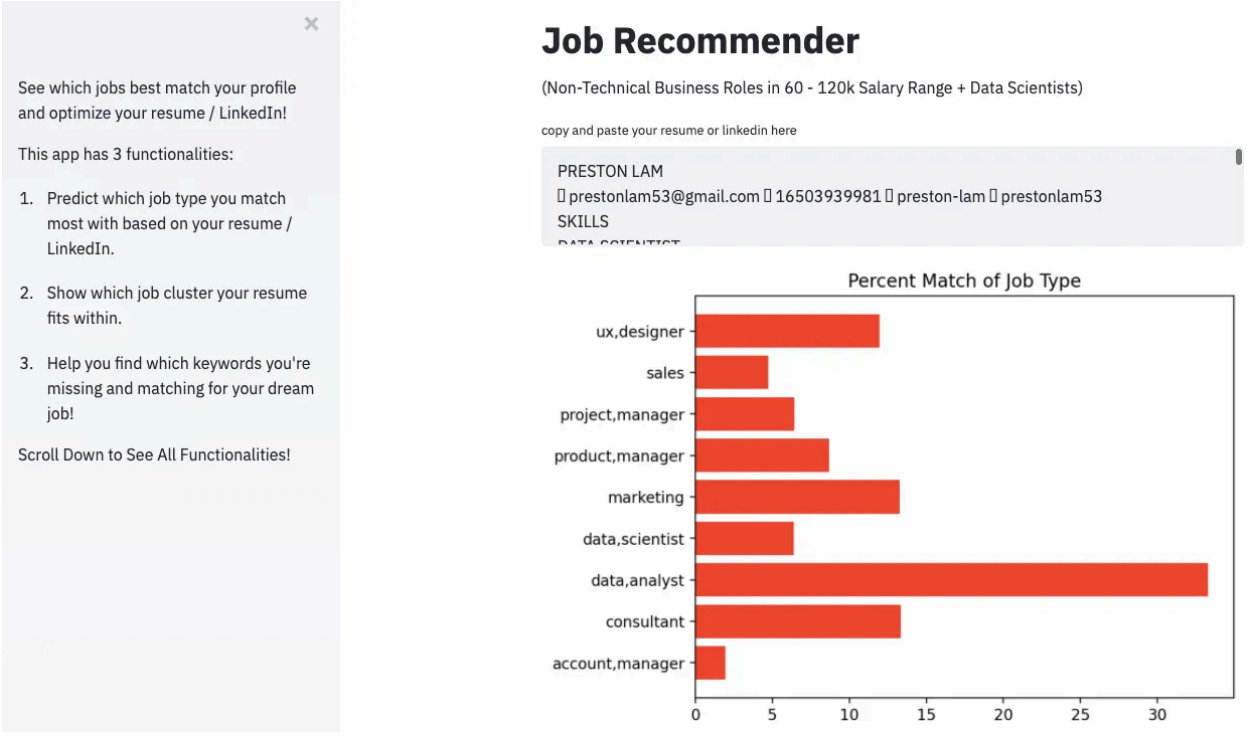
I will now summarize and discuss takeaways from the second resource linked:
https://towardsdatascience.com/building-a-job-recommender-for-non-technical-business-roles-via-nlp-and-machine-learning-626c4039931e

This is an article, by Preston Lam, outlining the features of an app he built to recommend jobs to people with non-technical backgrounds, and how he built the app.

“The app combines NLP techniques such as topic modeling with classification-style machine learning in order to determine the best fit for you. You copy and paste your resume / LinkedIn into the text box, and the app parses the text and presents you with ML-driven analysis of which jobs you fit and why.”

My first comment is that the link provided for the app goes to an empty page, so I am not able to see this tool in action either, which is a bit disappointing.

Here is a screenshot showing what the Job Recommender App is supposed to look like.



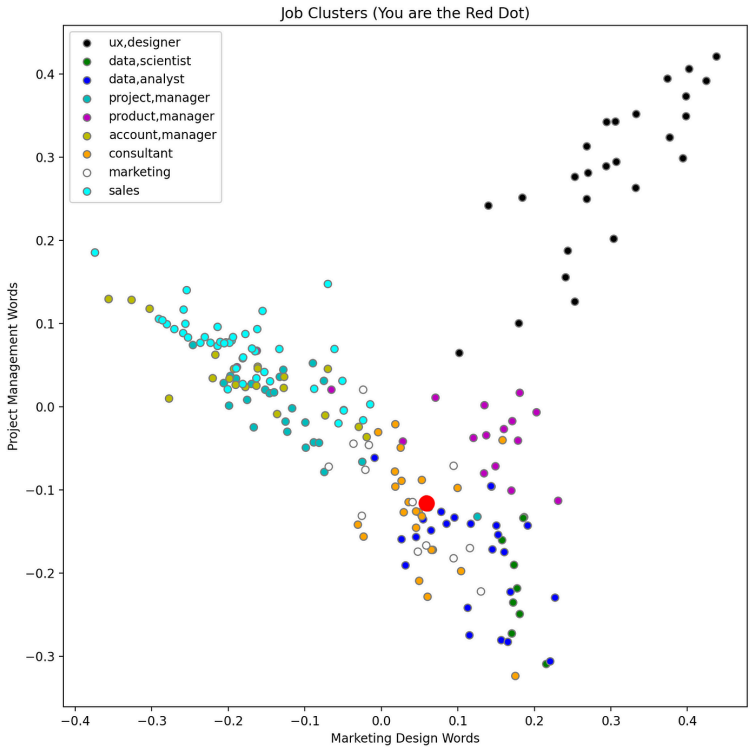
I think this is a neat idea and I think Lam is on to something when he says “I believe that this issue (of not knowing where to focus the job search) is quite common for people with non-technical backgrounds.” Obviously less technically trained people entering the workforce are going to have a less concrete idea of what it is they are capable of working on for a job. I think in general knowing where to focus the search is difficult for younger people or recent college grads, or people new to a particular industry in general. Being still young in my career with only about 5 years of technical experience across a few jobs, I’ve often wondered to myself, well I know Python and I have a

strong background in math and physics, but should I be applying/searching for development positions at research labs, software engineering positions at companies, test engineering positions, data analyst or data scientist positions? Etc. Seeing this chart displaying percent match of job type is interesting and I think is a useful insight to be able to report to a user, particularly since they would be unsure of what their best fit or 'niche' would be. Just looking back at my younger, more naive self, having some kind of way to visualize different career options and their levels of practicality or feasibility based on my experience would have probably saved me some headache, disorientation, and indecision related to my impending job search struggles. So I will say that I really like the idea or premise behind Lam's tool. It's to help people get a better sense of where they can be dangerous entering the workforce, but not necessarily to model which jobs a user would prefer or why. I find this resume-to-job-type matching to offer more insight than the resume-to-job-posting matching provided by Behrain's tool, particularly for younger, greener people entering the workforce. I could see this tool being used to help supplement the refinement of a user's resume, to help them identify and focus on different career niches they can better cater their resume to, perhaps utilizing different versions of their resume. It can help users have a more concrete idea of what jobs to apply to and what skills and/or experience to highlight or omit from their resume given which titles they are seeking. So let's take a closer look at how Lam's tool is supposed to work.

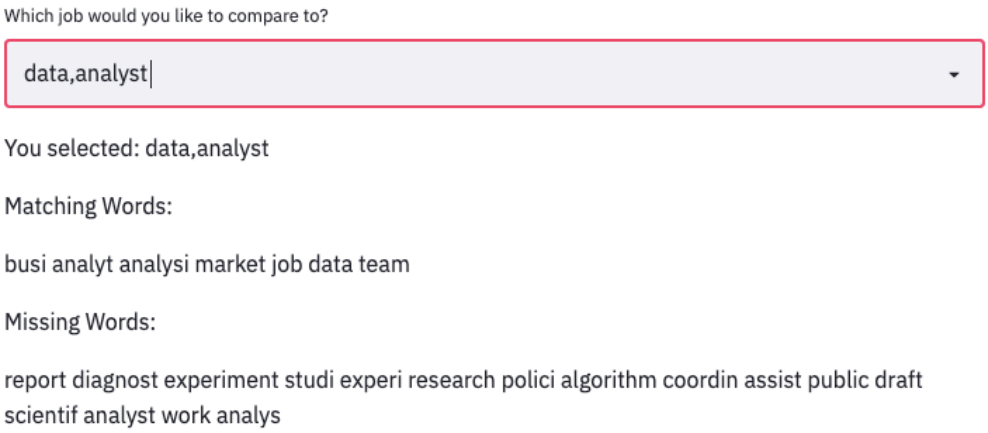
The App has 3 Features:

Feature 1: Return percent match by job type.

Feature 2: Return a chart of where your resume fits in with the other job positions based on topic matches. This chart hopefully provides some sort of explanation as to your results from Feature 1.



Feature 3: Select a dropdown of different job archetypes and see which keywords your resume both matches and doesn't match.



I think these are all potentially useful features, but I really like features 1 and 2. Feature 2 is neat because it can allow the user to easily learn about the characteristics of different job types and

how they differ. Basically the user can get a sense of not only what job types their skills are most pertinent to but also a nice picture of the overall landscape of jobs, as far as how jobs differ or share proximity by certain measures. I think Lam envisioned and (hopefully) ended up building a really neat and potentially useful tool. He links to this code https://github.com/prestonlam53/building_job_recommender_3 which I will read, attempt to execute myself, and discuss, but first I will summarize the steps he lays out for his project in the article.

Step 1: Scoping the Project

He starts off talking about Step 1 by saying that "The most important part of a data science project is scoping — that is, planning your project so that it fits your time and effort constraints, yet is still capable of answering a valuable question. There's so much data and avenues for exploration that a certain domain can possess, the sheer amount can be overwhelming. You thus need to be clear about what issue you are trying to tackle, what specific data you'll look for, and what is the minimum threshold for success."

I'm glad Lam makes this point because it is important. Although I have done some scoping and planning for my project up to this point, I would have benefitted from doing more scoping, particularly scoping the time required to complete each component.

Lam states that the key question he needed to answer for his scoping was figuring out what job types to analyze. He explains:
"So many different jobs exist in the non-technical business space. I felt that if I included too many job applications, the project would not have its intended effect. The modeling might be less accurate, and the app might ultimately be too cluttered and unfocused to be helpful for the end user."
Ultimately, Lam settles on 2 criteria for jobs to analyze:

"

1. The jobs must be business roles that don't require technical skills. This excludes roles such as software engineers or medicine or acting.
2. The job must be between the 40–120k salary range. This captures the range at which people with 0–3 years of experience typically earn for.

”

I think these constraints that Lam came up with are extremely important to his project scoping and ultimately allow him to narrow down the intent and know exactly what kind of value his app is meant to provide.

After identifying these constraints, Lam thought of some broad job archetypes that made sense to use as base classes for his model, and also sought feedback from his colleagues by surveying them on what job types they were interested in. Then, he initially chose the 5 most popular jobs to start with as the basis for his analysis, and once he was confident his model could correctly distinguish between job types, he then added 3 more as well as “Data Scientist” out of his own personal interest.

Step 2: Scraping Data

Lam sought to use job postings in the respective job types that were within the constraints he defined for his project. He first looked at LinkedIn and Indeed, but found that their sites were too difficult to scrape because of dynamic loading. So he settled on using job postings from Glassdoor, which also uses dynamic loading, however, he used code from someone else's data science project who scraped Glassdoor and modified it to suit his needs. Lam states that he was quite fortunate for this because otherwise building his own scraper would have taken him far longer.

After getting the scraping code working, Lam scraped 40 posts per job type and then inspected the text files he generated closely to find that a lot of the posts he scraped were for jobs that were not really aligned with any of the job types like ones that were in construction, or did not satisfy the criteria of being attainable for someone with 0-3 years of experience. So he set up filters to remove disqualified data and ended up with the text descriptions of 149 job postings across the 9 different job types as the dataset he was going to train a model on.

Step 3: Data Cleaning and Topic Modeling

"The next step was cleaning the text descriptions. I used standard cleaning techniques such as removing punctuation and capitalization, then tokenized and stemmed the words to standardize semantically. Lastly, I put the data in an array format using a vectorizer."

This seems more or less pretty standard, but I will take a closer look at exactly how Lam processed his data by looking at and taking a stab at running the code he provides.

"Once the data was in an analyzable format, I performed topic modeling, trying several techniques but ultimately landing on TruncatedSVD. The optimization factor was how accurately I could predict job types with the classification model I built in Step 4. I came up with 20 different topics in total."

So far, this is the most interesting or curious thing that Lam has mentioned about his project that I would really like to know more about in detail, so I definitely need to look at his code. I asked ChatGPT what TruncatedSVD is and it said that it "is a dimensionality reduction technique, particularly useful when dealing with large sparse matrices, commonly encountered in text data analysis, recommendation systems, and other applications…

Applications of TruncatedSVD include:

- Dimensionality reduction in text data analysis, such as topic modeling and document clustering.
- Feature extraction in recommendation systems and collaborative filtering.
- Noise reduction and compression in image processing and signal processing.

"

Based on this high-level description of what TruncatedSVD is used for, it may prove to be a valuable technique in my own project.

Step 4: Build Classification Algorithm

"After topic modeling, I used the topic-document matrix and fed it into a classification algorithm. Optimizing for accuracy, I ultimately settled on a random forest classifier. The model returned ~90% accuracy on validation sets, showing strong competency in predicting correct job types for each job description.

Next, it was time to give the model a functional purpose. I used the above topic model to transform one's resume, and then used the classification model above to predict which jobs the resume fit best with. I then extracted the percent match by job type, giving a more nuanced view of one's best job match — for instance, 60% project manager, 40% product manager — and giving the end user multiple career paths to investigate."

Again, this info could prove very useful in my own project, but I need to see the code in order to get a much clearer picture of how Lam's tool works.

Step 5: Build PCA Chart building Function

"When I showed people the results of their respective job match percentages, they asked *why* they fit in with those groups. I had a pretty tough time explaining the underlying mechanisms of the models to a non-technical crowd, and I thus decided to show them a simplified chart to explain these job match results.

First, I reduced the topic-document matrix into two dimensions using PCA. Then, I plotted out each job type according to the reduced dimensions. I also applied the dimensionality reduction to one's resume and was able to plot where one's resume held against the other jobs on the chart. In interpreting the new PCA features, I saw that they were heavily geared towards 2 topic types: Marketing related keywords, and project management related keywords."

This component to Lam's tool will also be helpful to look at in more detail as I will also need to be able to elaborate on my tool's results and provide more insight to users, and the techniques used by Lam may be applicable to doing this.

Step 6: Build Keyword Matching Feature

"Users are interested in *how* they can improve their resumes for a better chance at whichever job they are targeting. I decided to make a matching keywords function in order to make the app more useful.

In this feature, the user selects which job they are interested in from a dropdown, and the app returns which keywords are matching and missing from their resume. People can see both where their resumes currently stand, and what words and experiences they could put in for a more targeted application.

This feature was probably the easiest one to make. I used the same topic model above to come up with the most significant words in each job type — say top 20 — and used list comprehensions to see which words matched or missed."

This is another neat feature of the tool Lam built. It serves as an example of how one can expand on the basic modeling and provide more value to users. Though adding search functionality to my tool might be overly ambitious or beyond the scope of my project, I do feel encouraged to at least consider nuances to the functionality of the app I end up building.

Step 7: Write and Deploy App

Just like Behrain did with his job recommendation tool, Lam also used Streamlit for writing his app. Lam states "I then used Heroku and git to upload it to the web, although in retrospect it would've been far easier to use streamlit's newly released app deployment features."

This is obviously useful information for me to use to build my tool.

I will now look at the code that Lam provides a link to.

First, I am looking at this Jupyter Notebook, which is stored in a folder called ".ipynb_checkpoints".
https://github.com/prestonlam53/building_job_recommender_3/blob/master/.ipynb_checkpoints/testing%20stuff%20out-checkpoint.ipynb

In this code he defines several functions: create_clusters(), plot_PCA_2D(data, target, target_names, user_data), and transform_user_resume(pca_model, resume) which he uses to test different aspects of his tool in the next code cells of the notebook, like so:

```
X_train, pca_train, y_train, y_vals, pca_model = create_clusters()
resume = "hi i'm preston and this is my resume"
user_input = re.sub('[^a-zA-Z0-9\.]', ' ', resume)
user_input = user_input.lower()

user_input = pd.Series(user_input)
transform_user_resume(pca_model, user_input)
```

create_clusters is a function which fits a 2-dimensional PCA model to the topic data, which comes from the already generated 'topic_df.csv' file.

plot_PCA_2D(data, target, target_names, user_data) allows the user to see where the text from their resume falls in terms of marketing design and project management related words relative to the set of job postings Lam prepared.

transform_user_resume(pca_model, resume) takes in the user's resume and then does the following to process the resume:
-custom data processing involving tokenization and count vectorization
-transforming document in the form of count vectors into topic model-space.
-transforming document as represented in topic model-space again with the 2D PCA model.

Now, we can probably learn more by looking at the custom code he imports:
`import process_data as pda`, which comes from
"https://github.com/prestonlam53/building_job_recommender_3/blob/master/process_data.py".

This Python file is 173 lines of code and defines the following functions:

tokenize_stem
display_topics
return_topics
process_data
predictive_modeling
predict_resume

get_topic_classification_models
main

I will discuss what each one of these functions does.

tokenize_stem(series): This code uses the built-in TreebankWordTokenizer of the nltk.tokenize module to perform tokenization and the built-in PorterStemmer from the nltk.stem module to perform stemming on a pandas series (which presumably contains strings).

- I may consider using these already available utilities for my own data processing as well, but we will have to do some trial-and-error to see how different approaches used in the data pre-processing and feature engineering stages yield different performance/results in order to ascertain whether these particular methods give better results.

display_topics(model, feature_names, no_top_words, topic_names=None): returns a list of topics based on the .components attribute of the 'model' passed to this function.

return_topics(series, num_topics, no_top_words, model, vectorizer): this code calls tokenize_stem on the 'series' that is passed to this object, and then does further processing on the resultant series to transform the vectorized text data into topic modeling space. Then the 'document_topic' matrix, topic names, and the model itself, as well as 'vec' and 'topic' are returned by this function.

process_data(): uses the functions above to read in files, model, and return a topic_document dataframe. First a 'jobs.csv' file is read in order to create a dataframe jobs_df whose 'Description' column is then passed to return_topics, as well as the arguments (20, 10, TruncatedAVD, TfidfVectorizer), in order to generate a topic dataframe with 20 topics using TruncatedSVD as the model and TfidfVectorizer as the base class for the count vectorizer used on the data after it is altered by tokenize_stem (which gets called in return_topics). Once the topic dataframe is fully created and has the 'job' column, this process_data function returns the topic_df, topic_model, vec, and topic_list objects. Remember that *topic_model*, *vec*, and *topic_list* were returned by return_topics, and that *topic_df* was created from *doc* which was returned by return_topics.

predictive_modeling(df): fits, optimizes, and predicts job class based on topic modeling corpus. This function takes in the dataframe df and does a train_test_split on the dataframe. Then, with code that is commented out in the function, it looks like Lam performed a grid search on the following param_grid: param_grid = {'n_estimators': [100,300, 400, 500, 600], 'max_depth': [3,7,9, 11]} to evaluate best hyperparameter choices for sklearn's RandomForestClassifier algorithm on his data, which appear to be n_estimators=500 and max_depth=9, since he picks those values when instantiating the RandomForestClassifier in the uncommented code before fitting it to the training data. He then takes the mean of 5-fold cross-validation scores to get an accuracy estimate of the RandomForestClassifier model on the training data, and then calculates the accuracy on the test set by calling rdc.predict(X_te). predictive_modeling returns the rfc model.

predict_resume(topic_model, model, resume): transforms a resume based on the topic modeling model and returns prediction probabilities for each job class.

get_topic_classification_models(): calls process_data() and predictive_modeling on *jobs_df* to return the topic model, the predictive model, and vec.

main(resume, topic_model, predictor, vec): run code that predicts resume. This function contains the following 4 lines:

```
doc = tokenize_stem(resume)
doc = vec.transform(doc)
probabilities, classes = predict_resume(topic_model, predictor, doc)
return classes, probabilities[0]*100
```

This thread https://community.openai.com/t/job-recommendation-system/647428 wasn't super insightful however it does contain some ideas people have about how to solve a similar problem to what Lam's tool, the one I just looked at, does. Basically OP wants to build a job recommendation system based on a submitted resume. One user tried using the ADA embeddings model to compare the texts and give similarity scores based on cosine similarity, but reports the model not being very nuanced/accurate. Another user who replied gives some interesting ideas I may come back to, and also mentions some more sophisticated approaches mentioned by someone else in a related thread that I want to check out. It's a very interesting thread:
https://community.openai.com/t/comparing-texts-using-ada-embeddings/641607/18