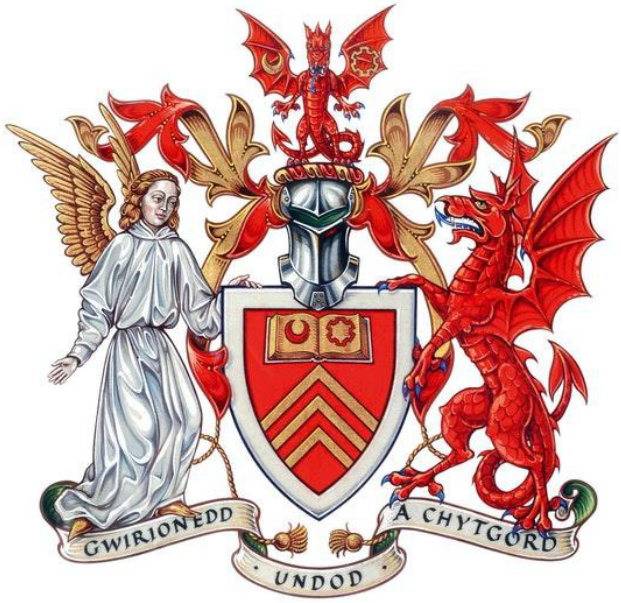
Automatic Generation of Word documents for the Year in Industry Programme

By

Worawat Kimpara

BSc Computer Science

A Final Report Submitted in Fulfilment of the Requirements for the Degree of Bachelor of Science in Computer Science of Cardiff University



School of Computer Science and Informatics,
Cardiff University, Wales, United Kingdom

Supervisor: Martin Caminada
Moderator: Natasha Edwards

May 29, 2020

Abstract

In this project, a tech stack comprised of Django Web Framework, PostgreSQL, and Nginx containerized with Docker to produce a production ready, stable, and scalable web application which utilizes vector space models with Term Frequency, Inverse Document Frequency (TF-IDF) in a search function. This web application was made to assist the Cardiff University Year in Industry Programme with the generation of word documents using information from the Skills Framework for the Information Age (SFIA) and with the implementation of a vector model TF-IDF search using natural language processing to calculate similarity models using TF-IDF to weight words based on their significance calculated based on the rarity of its use in a search. This algorithm, chosen due to its strong performance in comparing the similarity of technical papers in the study 'Text Similarity in Vector Space Models: A Comparative Study', utilizes vector space models and TF-IDF to yield results that surpassed the expectations of me and my client when used to map inputs to the SFIA framework. This search algorithm is implemented using Python in the web application utilizing the Natural Language Tool Kit (NLTK) and Gensim Python libraries. Careful consideration was put into place during the development of the project to ensure that stability, maintainability, and scalability is achieved to a high degree in the final project.

The project is available to be viewed at:

http://sfia.worawat.com

Source code is available at: (Deployment instructions in README.md)

https://github.com/SmilingTornado/sfia_generator
and
https://gitlab.cs.cf.ac.uk/c1744034/sfia-generator

Acknowledgements

I would like to thank my supervisor Dr. Martin Caminada and client Dr. Catherine Teehan for their help and feedback during the course of this project. I would also like to thank my former colleague Pulitz Lertamornthep for always being patient, helping me debug, and teaching me new things when I worked with him during the summer of 2019. Thanks to my family for supporting me from the start. And finally, thank you to my girlfriend Soo Bin Oh, for all her love and support.

Table of Contents

Introduction

Core Aims and Approach

This project intends to deliver a web application to be used by the Year in Industry programme for the automated generation of Word (docx) documents using skills from a skills framework called the Skills Framework for the Information Age (SFIA). This project also aims to implement a search algorithm of some form to take a user input and to try to pair the input with a set of skills. The development of such a project will benefit the Year in Industry as it would decrease the time needed to be spent on menial tasks for students looking to enter the Year in Industry programme. As there are many parts to this project, it will be developed in individual components that would gradually be combined to form the final product. The client will also be involved with the project giving input during weekly meetings which would increase the amount of feedback I will be able to gather from the client and allow me to make iterative improvements until the client is satisfied with the outcome. I will also be approaching this project from the onset with stability, maintainability, and scalability in mind and I will draw on my past experiences in professional web development to ensure that these three key principles are clearly embodied in the final release of this project.

Assumptions

This project will be created based on certain assumptions that will greatly affect the design choices and decisions made throughout this project:

- The project will be used in a production setting by the Year in Industry programme and must have the capability to perform to the needs of the program and be able to be scaled if needed.
- The content used by the project's current features must be able to be easily maintained and modified via a content management system without the need for a developer to modify the source code.
- The deployment process should be made to be as convenient as possible with features implemented with scalability in mind to accommodate for any potential future expansion of its use.
- The project should be able to be easily maintained and locally deployed in a development environment on as many types of systems as possible to ensure that anyone with the desire to, can modify, expand upon, and use the project for their own needs.
- The ability for future localization should also be considered due to Cardiff University's requirement to be as accommodating as possible to Welsh speakers.

Background

Overview of the problem

Currently, students in the Year in Industry programme have to go through the SFIA document to select the two skills that they think are most appropriate for them with the requirement that for BSc students, they must reach level three on at least one of the skills they've chosen and for MSc students, they are required to achieve level four competency in at least one of the skills. Once the student has chosen skills which they deem to be appropriate for them, they must select the range of levels for each with the lower bound being what level they currently believe they're at and the upper bound being the maximum level that they think is feasible to be achieved in the duration of their year in industry. As currently, students have to manually go through the document, select their desired skills and ranges for each skill, then manually populate a template with the information from the SFIA document, and finally format the document to make it presentable, this process wastes a lot of time that could have been used productively elsewhere such as for doing coursework, studying for assessments, or reorganizing lecture notes. With the popularity of the Year in Industry programme increasing by the year, the cumulative time wasted due to this process will also increase thus further highlighting the need for a solution to assist in this process.

Proposed Solution

The core idea of the web application would be a simple form where a user would input the skill codes and levels found in the SFIA document into the form. After submitting the form which would send a POST request to the web application, the web application should return a docx document populated with skill descriptions and a table populated with levels and their descriptions. Adding upon this, a search feature should also be implemented to allow students to input a body of text into an algorithm which would then pre-populate the form with skills which it has calculated to have the highest similarity to their input. This solution will be implemented using a fairly typical tech stack comprised of Nginx, Django Web Framework, and PostgreSQL which in the end will be containerized to allow for easy deployment using Docker. The process in which this tech stack was carefully selected is explained in more depth in the 'Specification and Design' section of this report but the summary of it is that the technologies and algorithms used in this project were all selected with stability, maintainability, and scalability in mind.

Software Development Methodology

As the project will be comprised of developing multiple components which will eventually be combined to achieve the final product, the usual 'Waterfall' software development model would not be as effective for the project as the testing phase is separated from the build phase. Therefore, I decided to opt for the 'Agile' software development model where I would be able to iteratively improve the web application with input from the client until their satisfaction is achieved. The Agile approach was especially suitable for this project as during the weekly meetings with the supervisor of my project, the client would usually present meaning that their feedback could be gathered regarding any new features that I've been able to implement during the course of the week. The Agile approach also helped me better understand my project as time goes on as my lack of personal experience with the year in industry programme means that I had little knowledge of the SFIA framework or how a web application to generate documents with it would be used so it was vital to the development of the project to speak to the client to better understand the framework and how the future web application would be used.

Specification and Design

Pre-Web-Application Preparation

As almost the entire project will be written in Python aside from some HTML, CSS, and configuration files, I decided that I would also use Python to write the scripts used for scraping the SFIA document and generating the docx documents. To scrape the SFIA document, I would first convert the PDF to an HTML document as HTML is much easier to scrape and then utilize the popular BeautifulSoup4 to help trawl through the HTML file and extract the information. For document generation, I will use the 'python-docx' library to add text, format font, text size, and styles, and to create tables and to format them.

Search Algorithm

As it is often time consuming for the students doing their year in industry to read through the entire SFIA document to find skills that match their own interests, the idea of creating a method of searching for skills was brought forward by the year in industry team. The specification for this was simple in that the year in industry student would input a paragraph about themselves and the web application would try to find skills that most closely resemble what they put in. Though the specification was simple, it would be important to choose an algorithm that would take into account factors such as topics or key words as the language used throughout the document is fairly similar and therefore the algorithm would have to understand and weigh the importance of each word in its context.

Chosen Search Algorithm Approach

The two main options of search algorithms for this project were either to use an algorithm that took advantage of machine learning or one that did not. Due to the lack of training data and the complexity that would be required to create training data, the route not involving machine learning was chosen. I modeled my approach to be based around TF-IDF (Term Frequency Inverse Document Frequency) due to the positive results yielded by it in the study "Text Similarity in Vector Space Models: A Comparative Study" by Omid Shahmirzadi, Adam Lugowski, and Kenneth Younge. This approach utilized natural language processing to create vector models with weighting for each word calculated with TF-IDF using the formula:

$$w_{i,j} = tf_{i,j} \times \log \frac{N}{df_i}$$

$$tf_{i,j} = \text{number of occurrences of } i \text{ in } j$$

$$df_i = \text{number of documents containing } i$$

$$N = \text{total number of documents}$$

The theory is that in the vector model, TF-IDF would assign rarer words a higher weight, the vector space model could then be compared using a generated similarities index. This approach appealed to me as the theory indicated that it would be highly effective at matching the similarities of sentences taking into consideration specific key words that it would weight higher with TF-IDF to increase the chances of finding better matches especially when comparing technical texts such as that of the SFIA framework. Creating a bespoke algorithm meant that if made in python with the Natural Language Tool Kit and Gensim libraries, it could be seamlessly integrated into the views file for the web application decreasing the deployment complexity compared to if using an external service such as Elasticsearch or Algolia. Thankfully, there is an abundance of information in the documentation of Gensim on its website made by the creator Radim Řehůřek. The many examples provided in the documentation greatly accelerated my development of my bespoke solution and would mean that if any are found, future optimization to the algorithm could also be easily implemented by modifying the existing code rather than being stuck using a system such Algolia where you cannot change the algorithm or ElasticSearch where it is much more difficult to modify the algorithm especially as it is written in Java meaning that someone wanting to improve the algorithm have to understand and know how to create complex algorithms in Python, but they would also have to know how to do so in Java.

Web Application Framework

The most important decision made during the development of this project was choosing the framework. Though I have built simple API endpoints with Node.js before, my plans to implement a vector model TF-IDF search algorithm which uses natural language processing drew me towards using a Python based web framework such as Flask or Django so I may take advantage of Python libraries such as Natural Language Tool Kit and Gensim though in the end I chose Django for its much more robust feature set compared to the lightweight Flask.

Flask

Flask is a lightweight web framework written in python based on Werkzeug WSGI (Web Server Gateway Interface) Toolkit and the template engine Jinja2. Flask is often used to power microservices because it is lightweight and simple core functionality allows for flexibility in extending its capabilities with other libraries made for Flask or python libraries made for general python use. Flask in its simplest default form does not come with a database abstraction layer meaning that if needed, the developer must import a library such as Flask-PyMongo to use a NoSQL database or Flask-SQLAlchemy to use an SQL database such as PostgreSQL.

Django

Django is a MVC (Model-View-Controller) web framework written in Python and is often thought to be the world's most popular web framework due to the thousands of projects known to utilize its simple, fast, reliable, and scalable abilities from startups to tech giants such as Facebook owned Instagram, Dropbox, and Spotify. Like Flask, if desired, developers can import or create libraries to suit the needs of their web application allowing for the use of various databases and the abundance of libraries for Python developers. Python and Django's flexibility allows developers to create and deploy new features easily and quickly as unlike Flask, Django's built in tools, made by the Django team and optimized for ease of integration, reusability, and speed, handles the core functions such as security and content management via the built in admin panel, allowing developers to spend more time working on the features.

Chosen Option

Though the lightweight approach achievable by Flask is often very useful for creating RESTful APIs and other microservices, For the purposes of this project, Django was chosen as although the included features increased the complexity of learning to adopt the platform, this wasn't an issue as I have had previous experience using Django professionally before meaning that these features would provide more of a benefit than a hinderance. Features such as the admin panel and models system built into Django proved to be very useful to the development of the program and would also be useful for future maintainability as all the documentation needed to understand the core functioning of a Django based web application is in one place; on the Django Project site. As Cardiff University is in Wales and its services should be made to accommodate speakers of the Welsh language, a key feature which played a role in the decision making for choosing the web framework was Django's localization abilities. The localization features in Django would allow for someone to add support for the welsh language in the future which would help satisfy Cardiff

University's need to accommodate Welsh speakers. The admin-panel feature would be of particular importance in the future when the project would have to be maintained and updated to accommodate for changes in the SFIA skills framework. Additionally, considering that the project is meant to be integrated into the university's system it could also be converted to a RESTful API with Django REST Framework to be easily integrated with other university pages using the same front-end rather than having a mismatched page being rendered by Django's built in system.

Databases

Django has a built in SQLite database management system which is suitable for most small projects. However, as this program is intended to be used for the year in industry programme, an SQLite database would not be suitable especially if the web application is expanded to store student information, new frameworks are added, and/or new skills are added. PostgreSQL was chosen to be the database of choice on my deployment mainly due to the vast amount of support that exists for its use with Django. PostgreSQL is also open source, SQL compliant, has some native NoSQL features, and has good performance with complex queries compared to other relational database management systems.

Deployment

As the web application will be used for the Year in Industry programme, it is important that it be deployed on a dedicated server rather than having students run it locally themselves. For the purposes of testing, I personally used a T2.Micro instance from Amazon Web Services (AWS) to run my test deployment at http://sfia.worawat.com/ as I'm most familiar with the AWS platform. However, as I did not use an Amazon Machine Instance using their in-house proprietary Linux distribution and opted to use the open source Ubuntu 18.04 instead, the process should be identical whether using an instance from AWS, Microsoft Azure, an organization's OpenStack, or a personal system as long as it is also running Ubuntu 18.04. Later on in my project I also containerize the web application and the elements used to support it using 'Docker' and 'Docker Compose' meaning that the deployment of this project is now able to be deployed extremely easily on any system running an x86 version of Mac OS, Linux, or Windows. Though I have not yet tested it on an x64 system such as a Raspberry Pi, I believe it may be possible though the number of dependencies I'm utilizing for this project means that there will likely be some issues encountered that would not be present on an x86 system. However, as almost all servers used in production environments are x86 based, this issue is a fairly minor one.

Web Server Gateway Interface (WSGI)

To run the web server, a WSGI is needed to act as the middleware to quickly serve static files and to determine the best way to route many requests concurrently. This greatly reduces the load on the web application server and improves responsiveness due to static files being served more quickly thus improving the perceived experience by the end user.

Apache

Apache is what I started off using as it is what comes with Ubuntu meaning I did not have to install any additional things to get it working. Apache is one of the earliest WSGIs if not the earliest one. For this reason, its popularity is no longer as dominant as it once was, but it is still very capable and is still used by many large companies. Another reason why I initially used Apache was the abundance of documentation it has as it uses '.htaccess' for configuration which is also very flexible. However, this feature comes at the cost of performance – a costly sacrifice when it comes time to scale.

Nginx

Though I initially used Apache during the development of my project, I eventually switched to Nginx to utilize its better performance and more scalable feature set. Nginx is currently the world's most popular WSGI and for good reason; it is fast, efficient, and promotes scalability. Nginx from the onset was "Written specifically to address the performance limitations of Apache web servers" ~Owen Garrett (Nginx' project manager) meaning it is better suited for more intensive use than compared to Apache. However, Nginx is not as configurable and is often harder to use than Apache. Though harder to use and less flexible, the main reason I switched to Nginx is its performance with it being able to often handle double the number of requests as Apache in the same amount of time. This increase in performance although likely unnoticeable at a small scale, would greatly improve performance at a larger scale

Containerization

Although the web application was not too difficult to deploy as I've had some formal experience with web development in the past, I would occasionally have to redeploy my web application either voluntarily because of a desire to change something, or involuntarily due to breaking something in the operating system. Initially, I did not think much of it as I was the only one working on this project however, as I later met the post graduate candidate who would be taking over from me to integrate my web application to a future Year in Industry Programme portal, I realized that not only would a complicated deployment mean scaling its use would be more difficult, it also means that any future maintainers or people wanting to build upon the web application would also have to learn how to deploy the web application meaning they would either have to read an extensive deployment note that I would create or they would have to learn how to set up Nginx, PostgreSQL, and the web application itself. Therefore, I decided to containerize my web application using a containerization tool. Unfortunately, I did not have time to explore other solutions such as CoreOS, so I had to use the containerization tool that I am already familiar with, Docker.

Docker

Docker is a virtualization tool used to deploy containerized applications and is often used in the industry to assist in deploying and scaling web services. It is especially usefully when developing in large teams where not everyone might have a system with the same specifications as its operating system virtualization allows everyone to develop in the same environment. The tool I

would be using is called ‘Docker Compose’ which allows developers to configure and make multiple Docker containers to work together. In my case, I would be using Docker Compose to allow users to easily deploy the entire tech stack using a single command once they install Docker, Docker Compose, and they clone the repository. This would enable anyone interested in the project on any platform to work on just the web application without having to handle the rest of the components of my solution.

Implementation and Algorithms

Scripting for document scraping and generation

SFIA Document Scraping

Before creating the document generator and web application, I first had to extract information from the human readable SFIA document in PDF format to a machine-readable format such as CSV or JSON. To allow for better structuring, I opted to use JSON over CSV. The structure of the JSON file could also be easily documented with a JSON Schema. I initially tried to directly scrape from the PDF but using the Python pdf reader library “pdfreader” did not preserve the structure of the document which meant that it would not be feasible to scrape data this way as it’d be difficult to determine the individual sections I’d like to scrape. Therefore, I first converted the PDF to an HTML file using ‘pdftohtml.net’ which allowed me to scrape the document using the ‘BeautifulSoup4’ web scraping library for python to identify the types of headers using their style classes and HTML tags. After analyzing the document for the pattern, I created a script to extract each skill and each level and stored it into a dictionary and then appended the dictionary to a list. Using the Python ‘json’ library, I created a file named ‘sfia_reference.json’, formatted the list, and dumped the contents into the file, which was then ready to be used to generate the documents.

Document Generation

I made a python script with a library called ‘python-docx’ This allowed me to edit text, create text, create tables, add page breaks, and change fonts and styles. To start, I used the form currently used by the Year in Industry Programme to create a template by deleting the blank tables and any information which I would have to replace. I created one template for the student version and one template for the employee version and I have the program select which one to use depending on the value of the parameter for it when the function is called. Next, I created functions to add descriptions and create a table for the skills and then depending on the input parameters, the program would generate the document with one or two skills and would order them in a way which would minimize the risk of the table on the first page spilling into the second. In the case where the skill does not have all the levels of a skill that has been requested in the parameters, only the skills that exist are used to populate the table. One issue I did run into was that the table would not auto-size even when I generated a table with the auto-sizing property. Therefore I had the program adjust the width of each cell by calculating for each skill the proportion of characters it had compared to the total and adding that to a minimum width to ensure that there wouldn’t be an excessively skinny column with one word per line.

Django Web Application

Starting the Django Project

To start, I created a python 3 virtual environment using the tool 'pipenv' on my desktop with the command 'pipenv shell' once in the shell, I used the command 'pip install django' to install the Django core libraries and ran the command 'django-admin startproject SFIAGenerator' to create the project. I then did a test run with 'python manage.py runserver' and navigated to 'localhost:8000' in my browser to make sure that the test server was working. I then stopped the server by using CTRL+BREAK and created an app called 'Generator' with 'python manage.py startapp Generator'

Creating models

I created three Django models the main one was for the skill information and had the skill's full name, the skill's unique abbreviated four-character code, and the skill's description. The second model was for the levels in each skill and had the foreign key to the skill it belonged to, the level's level number, and the level's description. The final model was a model to store uploaded JSONs for processing and only consisted of a single file field.

Admin Panel Functionality Additions

To allow easy editing of levels belonging to each skill, I had to customize the admin panel for showing skill information to also include the levels with foreign keys matching the skill. To do this, I defined a Django admin Tabular Inline class for the levels and appended the inline class to the Django admin for skills.

Importing JSONs

In order to quickly import JSONs with skill information into the models, I created an admin action in the admin panel which would take the selected JSON and take each skill and either update old skills with the information or create new skills if the skill hadn't been added before.

Form View

To display the form to the user, I created a django template using the popular library Bootstrap and JQuery to make the form look nicer and to have a ranged slider. To serve the form to the user, I created a simple view to return the form as an HTTP response using the Django render function when the user's browser sends a GET request. The user would then fill out the form and press submit which would send a POST request to the same URL but in another tab. If using a JQuery library like I did, ensure that in the 'data-ajax="false"' is put in the tag for the form

Validating POST and returning docx document

In the same view function as the one used to GET the form, I added an 'if' statement to check whether the request is a GET or a POST request. If the request is a GET request, the web application would return the form but if the request is a POST request, the web application would check to ensure that at least the first skill code has been entered and that it is a valid skill code. It would then check to see if a second skill were inputted and if one has been inputted, it would check the validity of it. This is done using a validation function. Once the skills have been validated, the web app then uses the document generation script that I had created before using the docx python library and returns the docx object to the user. In the case where the inputted skills are invalid, the program instead returns a page informing the user that the request was invalid. This page closes itself after a few seconds using a simple JavaScript script.

Vector Model TF-IDF Similarity Search

This feature allows for the user to input a body of text into the field which is then run through a vector model TF-IDF search to find the skill with level with the most similar description. The code can be seen in the appendix as 'Appendix A' and can be broken down into the following steps:

1. The body of text is broken down into sentences which is then broken down into a list of words and this list of words is appended to a list.
2. The library 'Gensim' is then used to generate a dictionary, which is a list of unique words, from the list of lists of words for each sentence and then each word is assigned a number to represent it.
3. A corpus or 'bag of words' is then generated which measures the frequency that a word is used in a sentence.
4. A TF-IDF (Term Frequency, Inverse Document Frequency) is then generated for the corpus which identifies the importance of each word in the corpus. This is done so that rarer words are given a higher weight than common words when later comparing similarities.
5. 'Gensim' is then used again to create a similarity measure object which will be used to compare with later.
6. For each sentence in each level and skill description, the similarity to the similarity model must now be calculated using the following steps:
    a. Each sentence is tokenized as before by breaking down the sentences into a list of words.
    b. The corpus or 'bag of words' for the sentence is then generated using the dictionary created earlier in step 2.
    c. Each word in the sentence is then run through the similarity measure object which uses the weighting calculated in the TF IDF in step 4 and then the values are summed up.
    d. The score for each sentence is then found by averaging the sum.

7. For each level or skill, the average similarity for all the sentences in its description is calculated and then stored in a list of dictionaries with the skill code as its key. In the case where a level from the same skill has already been added to the dictionary, if the new level has a higher similarity, then the existing one is replaced otherwise the new one is discarded
8. The skill code key with the highest similarity score is then found and stored with the 'max' function which gets the highest score in the list.
9. To find the second highest scoring skill, the highest scoring skill is then removed from the list and then the 'max' function is then used again to find the highest scoring skill currently in the list and then the code is stored.
10. The web application then renders a form to generate the word document prepopulated by the skills which were most similar. If no similar skills were found, the form is not populated.

Additional Features

Skill Browser

To avoid having to have the user have the SFIA reference PDF next to them when filling in the form, I created a way of browsing skills. When the user requests the page, the site renders the page using the list of skills so that in the case a skill is added or removed, the site self-updates. If the user then clicks on one of the skills in the list, a page for that skill is then rendered by taking the skill code from the URL and finding the corresponding model and then using the information of the model to populate the page along with the levels that have the skill model as the foreign key.

Skill Selector

Using the skill browser, the user can select skills they want interactively rather than having to manually input the skills into the form. To do this, I created new URL patterns and modified the views to show the list of skills and to show the details of an individual skill. When the first skill is selected by clicking the link in the skill details page, the user will be sent to the list of skills but now, the skill code for the selected skill will now be in the URL. The user can now browse for the second skill that they would like to select. Once the user finds the one that they want to select, the detail page will have a form submit button that will send a POST request with the skill codes in the body of the request to the form page which will then render the form with the skill codes pre-populated in their fields. As the form page is also the endpoint for submitting the form to request the docx document, the endpoint checks the body of the request first to see whether it is requesting the document or the pre-populated form.

Deployment

Even once the web application had been completed, the method of deploying it was too convoluted in my opinion and therefore to enable fast and convenient deployment, I utilized Docker to containerize my project. To ensure fast and efficient development, I also preserved the ability to run the project locally without using docker by installing the requirements with or without an environment and then running 'python manage.py runserver' where it will run on localhost port 8000.

Docker Containerization

To containerize my project, I created two 'Dockerfiles' which are the files that define how to set up an environment for a service. One of them creates the container for the Django web application and the other creates a container for the Nginx service. Fortunately for simplicity sake, there is already a ready-made image of PostgreSQL which does not need to be configured with a Dockerfile and can just be passed the parameters for its login credentials in the Docker Compose file. If any of the parts of the tech stack is replaced by an external service, the part handling that component can just be disabled by removing it from being created in the Docker Compose file.

Django Container

To build the container for the Django web application, the Dockerfile first pulls an image called 'python:3.8.3-slim' which a Debian based slimmed down Linux image with not much more than what is needed to run Python 3.8.3. There is a lighter image called 'python:3.8.3-alpine' however, various issues arise when trying to install the Python library 'Gensim' which is unfortunate as Gensim is essential to the web application's vector search algorithm. After pulling the image and installing it, the Dockerfile then makes a directory called 'code' where all the files for the container will be put. It then copies the requirements file into the 'code' directory and runs 'pip install -r requirements.txt' to install all the needed packages. One small quirk of the Natural Language Tool Kit (NLTK) is that pip doesn't install all of the files it needs to do certain tasks so once all the requirements are installed, I have to run an additional command 'python -c "import nltk; nltk.download('punkt')"' which downloads the 'punkt' module which is needed in my vector search algorithm. Once everything has been installed, the Dockerfile finally copies the rest of the files in the directory into the 'code' directory. Additionally, to avoid having to change the settings file between the non-docker environment and the docker environment, I separated off the settings file so that by default, the development friendly non-docker one is used and then I configured docker to specifically use the one I made for docker.

Nginx Container

The deployment for Nginx is a bit more complicated as the Nginx server must be configured first. To configure it, I told it which port the Django web app is running on (port 8000), which port to listen to (port 80), and the location of the resources that it will send out such as static files. Once the configured, all that the Dockerfile does is defines the work directory, copies the configuration file to it, and then copies all the project files into a directory called 'code' where Nginx will access the static files from.

Deploying on Linux

To make it as seamless as possible, I took some time to make some scripts so if deploying on Ubuntu 18.04 or some other Linux distributions, it only takes these four commands to get the project running:

1. git clone https://github.com/SmilingTornado/sfia_generator.git
2. cd sfia_generator
3. sudo bash docker-compose-install.sh
   - If using Windows or MacOS, consult the Docker Documentation for installation information
4. docker-compose up -d

To add a superuser, execute the command 'sudo bash createsuperuser.sh' and input the credentials. And to edit the secret key, use vim to edit SFIAGenerator/settings/base.py. As Django automatically restarts the server when it detects changes, the changes would be applied automatically.

Scripts

docker-compose-install.sh

This script uses the Advanced Package Tool (APT) to download docker and docker. First it updates the package manager, then it removes any existing docker installation that might interfere, then it installs the docker.io and docker-compose packages, then it starts docker, and finally, it enables the docker service.

createsuperuser.sh

Though normally only taking one line to execute anyways, this script just makes it a lot more convenient. All it does is run the command 'docker-compose run web python manage.py createsuperuser --settings=SFIAGenerator.settings.docker' which runs the command 'python manage.py createsuperuser --settings=SFIAGenerator.settings.docker' in the 'web' container which houses the Django web application. This command allows someone to create a superuser which can be used to access the admin panel. This command is only needed so that whoever uses it can then go and define groups and create users in the admin panel.

Ensuring ease of future project development

To further increase the ease of development, I also preserved the web application's ability to use an SQLite database by having separate settings files where the one used for a development environment would automatically be used by default and the settings for production, in a separate file, would be specifically used by the scripts used to deploy the program via Docker. However, using a well featured interpreter such as PyCharm by JetBrains would allow for the use of the interpreter within the Docker container so in the case where incompatibility across platforms occur when using certain libraries such as graph-tool, using docker would enable non-Unix developers to continue using their regular development flow in Windows rather than having to switch over to using Linux or MacOS. This solution enables the ease of use for

Results and Evaluation

Client Feedback

Overall, the client was very satisfied with outcome of the project noting that the stability and usability of the web application were very good and she showed particular interest in expanding the possible applications of the vector model TF-IDF search so that it can be applied for other skill frameworks such as the Institute of Information Security Professionals (IISP) Information Security Skills Framework. The flexible input for the search function has also meant that the client has reportedly also utilized the search algorithm to purposes other than its original one with at least some degree of success by inputting university module descriptions into the search to try to find their corresponding skills. Following the conclusion of this project, I will likely see how I could accommodate a more flexible use of this search functionality by possibly creating an API allowing users to upload their own documents to search through rather than being restricted to only searching the skills stored in the models of the web application.

Personal Evaluation

Like my client, I am personally also very satisfied with the outcome of my project. The web application, built on Django, is stable, responsive, and to my knowledge is production ready especially as if it is needed to be scaled in the future, it could be easily done. I have very little regrets about the design choices I made as I believe the web application I produced is production ready for use in the real world and to my knowledge, no one has encountered any major issues on the version I've had running publicly at http://sfia.worawat.com. If I had more time, I would possibly explore more containerization solutions such as CoreOS though Docker has performed admirably throughout all my testing without fail and should be more than capable of performing in a production environment especially as it is already used by many large companies to deploy their web services such as JPMorgan as evidenced by their Docker Hub contributions at 'https://hub.docker.com/u/jpmc/'.

Document Generation

The web application performs generation of the docx documents with little to no issues and with formatting issues only arising rarely when the user tries to generate tables with a volume of text too large for the auto fit algorithm to handle which causes the table on the first page to overflow into the next page. To remedy this, I did two things; firstly, the table with a larger amount of text is always generated on the second page where there is more room and therefore almost no chance of it overfilling into another page. In the rare edge cases where both tables would overflow on the first page, the user has the option to generate the tables and their descriptions on dedicated pages by selecting an option on the form. Additionally, the client requested that it be able to generate a document with only one skill table. With all the precautions for proper formatting, the chances that the end user will not be able to generate a nice and properly formatted form are drastically mitigated and in the very rare edge cases where the generation cannot be done at a high enough standard, the user has the option to manually open the docx file and modify the document if needed.

Search Functionality

Though harder to evaluate, I do believe my implementation of the vector model TF-IDF natural language search algorithm has been successful and effective in its role to compare document similarity. Similarly, to the results found in the study 'Text Similarity in Vector Space Models: A Comparative Study' the results were surprisingly good though not perfect. I personally tested the functionality of the search by inputting my full resume from https://worawat.com into the search which does yield a result of skills I am personally interested in though they may not be necessarily the ones I would consider my best match. However, I hesitate to fault the program as there may perhaps be a method of describing myself in a certain format that would allow the algorithm to better perform. To test this function in more depth, data could be collected on what users end up submitting after they've tried out the search functionality via allowing the user to opt into the data collection to ensure that the user knows what their data is being used for.

Future Work

Overall

In the future, the web application will be integrated into a portal made for the Year in Industry Programme which will contain other web applications meant to manage and assist people involved with the Year in Industry Programme. For a more seamless integration, The styling of the web application will have to be made to more closely resemble that of the other ones on the portal by changing elements such as font and color as well as adding elements such as logos and photos. As currently, the page is minimally styled using the Bootstrap and JQuery libraries, doing this would likely not be too difficult though if seamless integration cannot be achieved with the current built in Django template rendering system, it would also be possible to convert the project into a RESTful API backend with Django REST Framework which would allow for the use of more powerful frontend engines such as React.js or if a more lightweight approach is desired, static HTML pages with forms for posting to the API could also do less complex tasks. New skills framework such as the IISP Information Security Skills Framework mentioned earlier could also be added into the project via creating a new Django model to house them and modifying the current features to allow for both the current and future framework to work on the same platform. I also licensed my project under the Apache 2.0 License to ensure it can be easily used in the future

Search functionality

The search functionality has great potential to be improved and its applicability in other applications extended. As the search algorithm is bespoke, additions and improvements can be easily performed on it in the future by any interested people. Things that could be added to improve its effectiveness include adding additional input fields, refining the algorithm, and improving speed. As the search algorithm is built on natural language processing, a fast-moving topic in today's world, it means that innovations in the field have the potential to allow for dramatic improvements to the algorithm. Also, as the algorithm is fairly flexible and could easily be transferred for other document matching purposes, in the case of a new skills framework being added to the web application, the search should also be extended to work with it though additional tuning may be required as the way that each framework is structured may not be the same.

Conclusions

The project was an overall success and exceeded not only my expectations, but that of the client and the supervisor as well. A very usable and stable project was produced which will certainly assist the Year in Industry Programme in the future with skill document generation as well as the novel skill searching feature. My vision to ensure the scalability, maintainability and compatibility across platforms will also enable this project to be used, modified, and scaled as needed for its purposes and should ensure the longevity of its use and maintainability. As this is my first time building a web application of this complexity from scratch, I believe my experiences with professional web development in the past has also greatly contributed to my approach towards the development of my project as it ensured that each element of the project was developed with ease of use in mind not only for the user, but also for future maintainers and anyone wanting to deploy or modify the project to suit their needs. The tech stack chosen also allows for a great amount of flexibility and ease of maintenance as I tried to use tools widely used in the industry with readily available and high-quality documentation. I attribute a lot of the success regarding client satisfaction to the Agile software development process I decided to use over the Waterfall approach. The Agile process allowed me to receive iterative feedback on iterative improvements allowing me to continuously add features with input from the client which greatly helped me ensure that what I was developing was to that of the specification desired by the client.

Reflection on Learning

Web Development Reflection

Though I have had professional web development experience in the past, this was the first time I have developed a web application of this scale from scratch on my own. Whereas in the past, I would only have to know how to locally run the project and edit existing code, this project greatly helped reinforce my knowledge in web development by allowing me to go through the full process of creating the app rather than modifying an existing project that is already in operation. Through this project, I also came to appreciate each individual component in my tech stack as they were all crucial in what I view as they key features of the project that I would attribute to its success. I am particularly satisfied with the containerization of my project as it makes deployment so much more elegant.

Docker Reflection

Though I have used Docker in the past, I never recognized how much it contributes to the efficiency of deployment and development. To get started working on a project, all that is needed is to install Docker and Docker Compose and run the command 'sudo docker-compose up'. This process is magnitudes more efficient than having to set up the database, then set up virtual environments, and then setup the WSGI in the case of a deployment. The experience I have had using Docker throughout the development of this project has had a profound effect on how I will likely perform development in the future if I were to create a system where compatibility and scalability were key requirements. As more of the web is moving to containerization solutions such as Docker, I also believe that it may be of interest to start integrating this into university web application curriculums as if job board requirements are anything to go buy, it is quickly becoming a job requirement for anyone seeking to apply for a DevOps role and I have occasionally spotted it in the 'nice to have' section for back-end developer roles.

Natural Language Processing Reflection

One of the most interesting parts of the project to me was the implementation of my natural language processing-based search. Prior to this project, I have had no experience using natural language though after using it in this project, my curiosity in the field has increased. Following the completion of the algorithm used in the project currently, I have begun to wonder ways I could possibly tune it and to apply it to other uses. Prior to this project if I were wanting to implement a search algorithm, I would likely just utilize an existing service such as ElasticSearch. However, using a bespoke system allows for refinement and possible improvement especially as natural language as a topic is becoming increasing popular every day for use in data science. As I will soon be beginning work at a local data company as an associate software engineer following the completion of my studies at university, the field of natural language processing may become something I will spend more time exploring in the future.

Appendix

Appendix A: Search Function Code

```
def search_similarities(request):
    similarities = {}  # Dictionary to store the calculated similarities
    input = request.POST['input']  # Get the input from the request
    # Create a list of sentences where each sentence has been broken down into a list
of words
    gen_docs = [[w.lower() for w in word_tokenize(text)]
                for text in sent_tokenize(input)]
    # Create a dictionary of unique words
    dictionary = gensim.corpora.Dictionary(gen_docs)
    # Generate bag of words to measure frequency of word use
    corpus = [dictionary.doc2bow(gen_doc) for gen_doc in gen_docs]
    # Calculate Term Frequency, Inverse Document Frequency of words
    tf_idf = gensim.models.TfidfModel(corpus)
    # Create similarity model
    sims = gensim.similarities.Similarity(settings.BASE_DIR + '/Generator/gensim',
                                          tf_idf[corpus],
                                          num_features=len(dictionary))
    # Checking for similarities with level descriptions
    for level in Level.objects.all():
        skill_sim_total = 0

        for sentence in sent_tokenize(level.description):
            query_doc = [w.lower() for w in word_tokenize(sentence)]
            query_doc_bow = dictionary.doc2bow(query_doc)
            query_doc_tf_idf = tf_idf[query_doc_bow]
            sum_of_sims = (np.sum(sims[query_doc_tf_idf], dtype=np.float32))
            similarity = float(sum_of_sims / len(sent_tokenize(input)))
            skill_sim_total += similarity

        skill_sim = skill_sim_total / len(sent_tokenize(level.description))
        # Check if similarities for a skill has been calculated before
        if level.skill.code not in similarities:
            similarities[level.skill.code] = skill_sim
        # If calculated before, check if new description is more similar
        elif similarities[level.skill.code] < skill_sim:
            similarities[level.skill.code] = skill_sim
# This function continues down below on the next page
```

```
# Checking for similarities with skill descriptions
# Same procedure as with for levels
for skill in Skill.objects.all():
    skill_sim_total = 0

    for sentence in sent_tokenize(skill.description):
        query_doc = [w.lower() for w in word_tokenize(sentence)]
        query_doc_bow = dictionary.doc2bow(query_doc)
        query_doc_tf_idf = tf_idf[query_doc_bow]
        sum_of_sims = (np.sum(sims[query_doc_tf_idf], dtype=np.float32))
        similarity = float(sum_of_sims / len(sent_tokenize(input)))
        skill_sim_total += similarity

    skill_sim = skill_sim_total / len(sent_tokenize(skill.description))
    if skill.code not in similarities:
        similarities[skill.code] = skill_sim
    elif similarities[skill.code] < skill_sim:
        similarities[skill.code] = skill_sim
# Find the most similar skill
first_match = max(similarities, key=similarities.get)
# If the maximum similarity score was 0, return the form
if (similarities[first_match] == 0):
    return render(request, 'form.html', {'searched': True})
# Removes the most similar skill
similarities.pop(first_match, None)
# Finds the current maximum similarity score
second_match = max(similarities, key=similarities.get)
# If the new maximum similarity score is 0, return only the first match
if (similarities[second_match] == 0):
    return render(request, 'form.html', {'sk1_code': first_match.upper,
                                         'searched': True})
# Return rendered form with found matches
context = {'sk1_code': first_match.upper, 'sk2_code': second_match.upper,
           'searched': True}
return render(request, 'form.html', context)
```

References

Omid Shahmirzadi, Adam Lugowski, Kenneth Younge: “Text Similarity in Vector Space Models: A Comparative Study”, 2018; [http://arxiv.org/abs/1810.00664 arXiv:1810.00664]