



Job Analytics

Technical Guide

Daniel O'Sullivan
Student Number: 14307881
Supervisor: Suzanne Little

Table of Contents

● Introduction.....	2
○ Overview.....	2
○ Motivation.....	2
○ Glossary.....	2
● Research.....	4
○ Dataset.....	4
○ Exploring the Data.....	4
● System Architecture.....	11
○ Django Architecture.....	11
○ High-Level Design.....	13
○ Database Schema.....	15
○ Front-End Design.....	18
○ Implementation vs Initial Design.....	22
● Components.....	24
○ CV Parser.....	24
○ Requirement Matching Algorithm.....	26
○ Random Forest Classifier.....	29
● Problems and Resolutions.....	31
● Future Plans and Retrospective.....	32
● References of software used.....	33



Introduction

Overview

Job Analytics is a web application for employers and applicants to post and find jobs. Employers can post jobs on the app and applicants can apply for open positions. When an employer gets a CV from an applicant, the application will examine an applicant's CV and return how qualified that person is for the specific role and will also predict how hard that person will work. The idea behind this is to make recruiting for an open vacancy easier by letting the application do all the work for the employer. All the application needs is a CV. The application will provide the employer with some statistics about their pool of applicants that have applied for their job postings. Job Analytics is built using the Django web framework and uses a Random Forest classifier to predict applicant work ethic.

Motivation


This idea was developed during my INTRA placement. I was working in a team that specialized in predictive analytics and this peaked my interest in this topic along with machine learning. Learning these topics made me start to think about my 4th year project. I wanted to make an application that would be useful to people and make a task or function easier to carry out and it's result more accurate. My mother, who works in human resources, often complains about having to read through countless curriculum vitae to find the perfect candidate for a job. This made me think about making an application that will read through CVs and pick the best candidate. To verify the applicability of this idea, I also pitched the idea to my HR manager at my INTRA company.

He informed me that this idea already exists and that he uses applications like this on a regular basis. However, he stated that an application that also had the ability to predict how well an employee would perform in a role would be very useful to recruiters like himself. I found this idea interesting and set out to research how to build it.

Glossary

Python: Is a widely used high-level, general-purpose, interpreted, dynamic programming language.

JavaScript: Is an object-oriented computer programming language commonly used to create interactive effects within web browsers.



HTML: Hypertext Markup Language is a standardized system for tagging text file to achieve font, colour, graphic and hyperlink effects on World Wide Web pages.

CSS: Is the language for describing the presentation of web pages, including colours, layout and fonts. It allows one to adapt the presentation to different types of devices, such as larger screens or printers.

Django Framework: Is a powerful and flexible toolkit for building Web APIs. It used to implement backend services in web applications.

PostgreSQL: is an object-relational database management system with an emphasis on extensibility and standards compliance.

Bootstrap: is a free and open-source front-end library for designing websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions.

Natural Language Processing (NLP): the application of computational techniques to the analysis and synthesis of natural language and speech.

Natural Language Toolkit (NLTK): is a suite of libraries and programs for symbolic and statistical natural language processing for English written in the Python programming language.

Chart.js: Is a JavaScript library that allows you to include animated, interactive graphs on your website.

Machine Learning: is a field of computer science that uses statistical techniques to give computer systems the ability to learn with data, without being explicitly programmed.

Random Forest Classifier: are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees.

Pythonanywhere: is an online integrated development environment and web hosting service based on the Python programming language.

TinyMCE: is a platform-independent, browser-based WYSIWYG editor control, written in JavaScript. It is a rich text editors for which allows users on websites to create posts.

Scikit-learn: Scikit-learn is a free software machine learning library for the Python programming language.

Tablesorter: is a jQuery plugin for turning a standard HTML table with THEAD and TBODY tags into a sortable table without page refreshes.

Research

Dataset

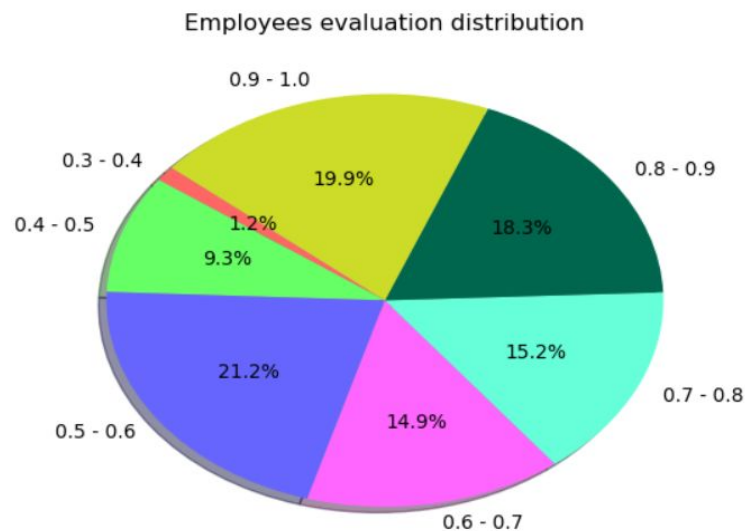
One of the core tasks of Job Analytics is to predict employee work ethic i.e. how well an employee will work. This combined with with an applicant job match score will give the job poster an idea of how well a person will fit the role in question. To achieve this, I needed a suitable dataset. I found the dataset titled “Human Resource Analytics”, on Kaggle.com. Unfortunately it has been removed from the website since I obtained it. It contains 15,000 entries and the following data:

- **Satisfaction level:** How happy an employee is in their role on a scale of 0 - 1
- **Average Monthly Hours:** Average monthly hours at the workplace
- **Time spent at the company:** How long an employee has been with the company in years
- **Work Accident:** Whether the employee has had a work accident or not (1 or 0)
- **Promotion in the last 5 years:** Whether they’ve had a promotion in the last 5 years (1 or 0)
- **Number of Projects:** The number of projects worked on while at work
- **Salary:** What kind of salary the employee is earning (low, medium or high)
- **Left:** Whether the employee has left their previous role prematurely or abruptly (1 or 0)
- **Last Evaluation:** How well an employee performs in their role (0 - 1)
- **Departments:** The department the employee works in

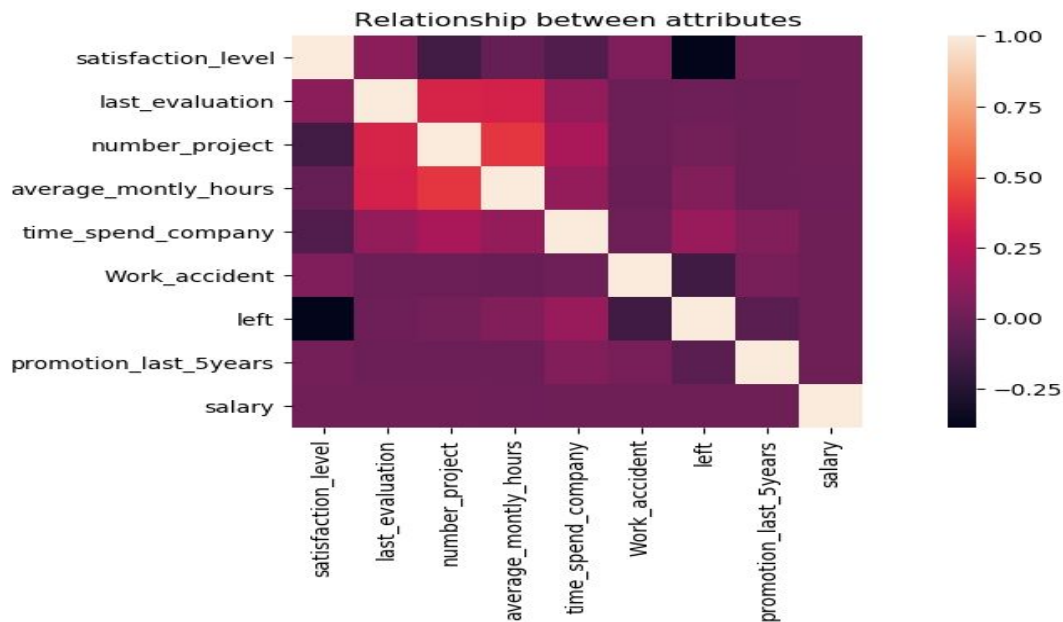
The dataset is primarily used (by users on Kaggle.com) for predicting employee turnover. I wanted to use it to instead predict the *last evaluation* attribute. I decided to construct a data report to see the distribution of data in this dataset and what relationship the attributes have with each other.

Exploring the Data

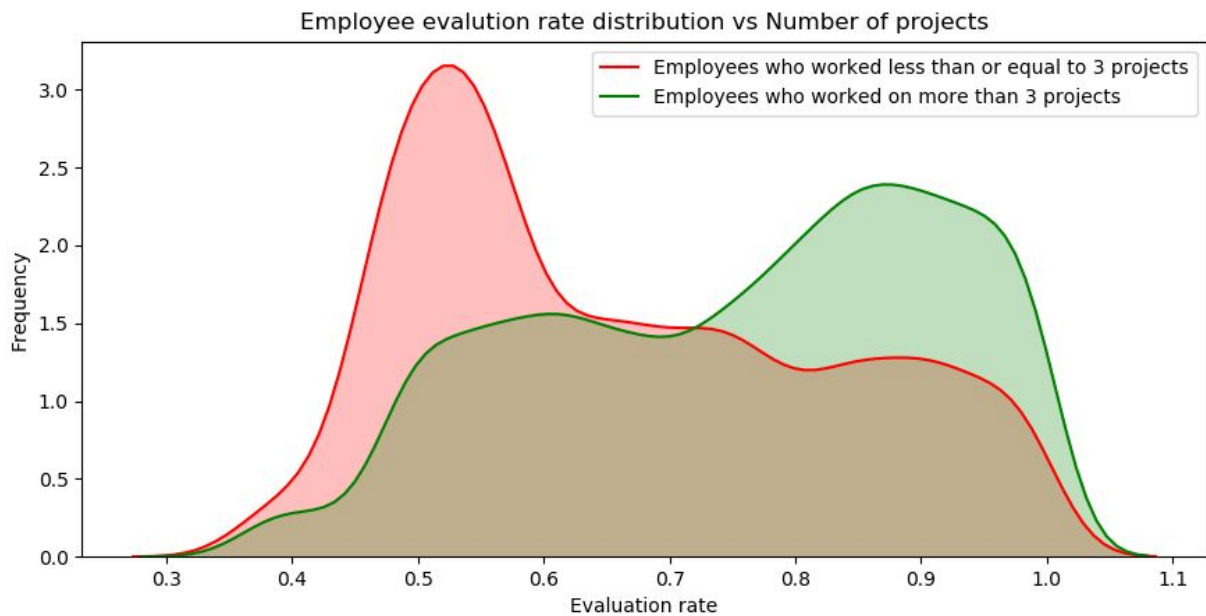
The first thing I wanted to look at was the data I was attempting to predict, *last_evaluation*. The following graphs show the distribution of values in the column:



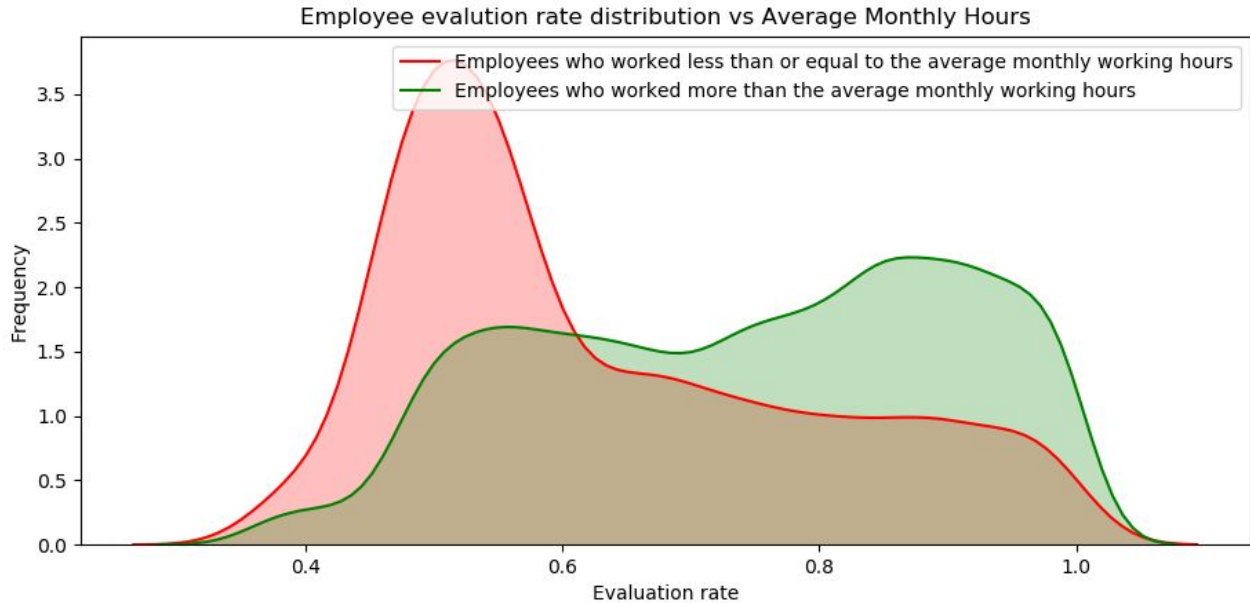
This histogram helped us learn that around 52% of employees in our dataset have an evaluation rate greater than or equal to 70%, leaving 48% to have a lower evaluation rate. There is also a bimodal distribution for employees with low (<0.6) and high (>0.8) rates. I also used a pie chart to see the distribution of evaluation rates more clearly. It was worth noting that there is no evaluation rate in the dataset that falls below the range between (0.3 - 0.4). This needed to be considered when constructing the prediction algorithm later. After taking in these observations, I was left with a few unanswered questions. Why is there this distribution? How does this influence other attributes in our dataset? I wanted to further analyse this evaluation distribution further. I used a correlation heat map plot to see what attributes have an effect on evaluation rates and what other relationship between attributes could I observe.



Observing this graph, we can see what relationship attributes have with each other. In the top left of the graph, we can see a big orange/red square formed across some attributes. This is because there appears to be a positive correlation relationship between last_evaluation, number_project and average_monthly_hours. This could mean employees who do more projects and work more hours are evaluated highly. This is interesting, why would these attributes in particular have a greater effect compared to others? The other attributes like satisfaction level, the time spent at the company, whether they've had a work accident or not, a promotion in the last 5 years and whether or not they left their role prematurely seem to have little effect on employee evaluation. Now that I knew of this positive correlation, I wanted to see how each of these attributes (Number of project and average monthly working hours) had an effect on evaluation rate individually. I used KDE plots to show this. First, number of projects versus evaluation rate.



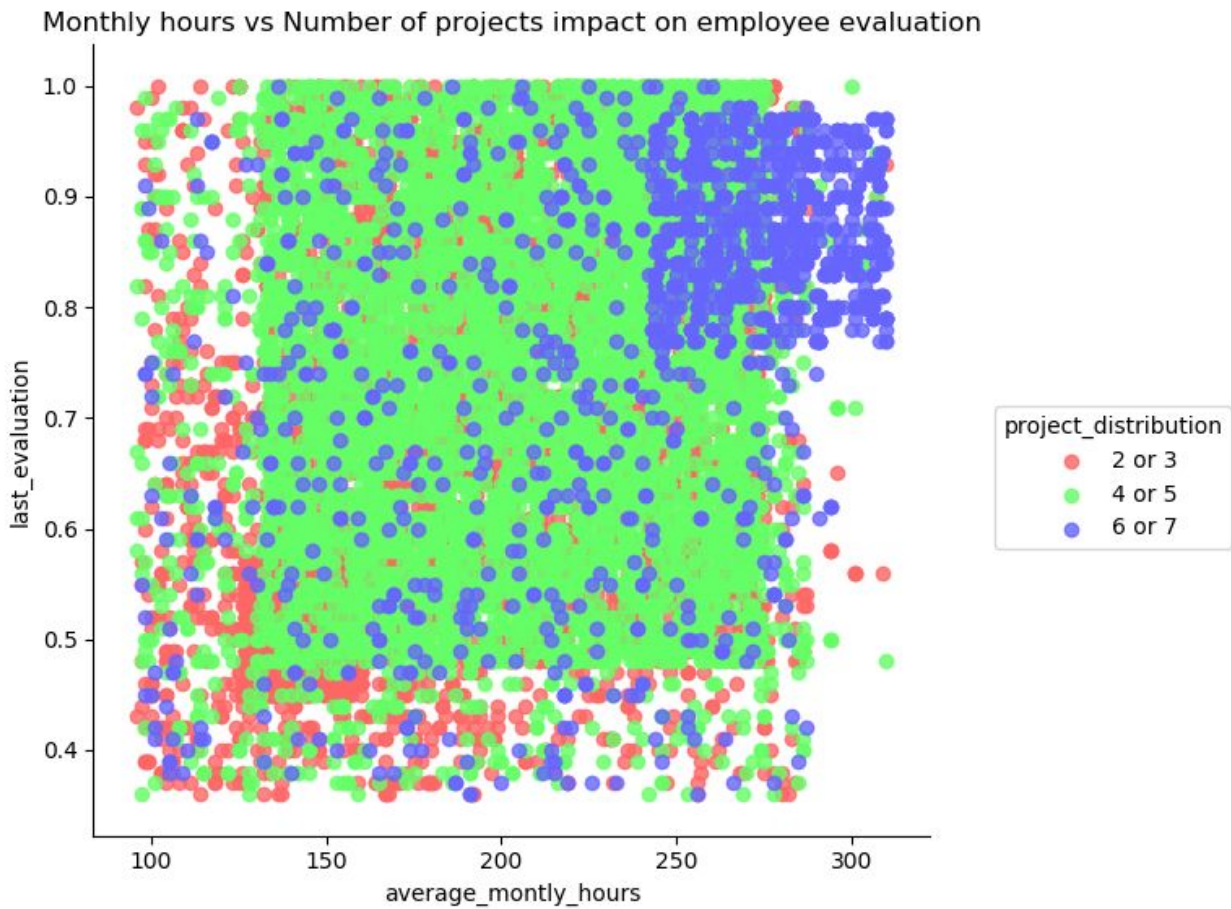
The first graph shows the distribution of employees based on evaluation rate and number of projects worked on. I split it to show the frequency of employees who have worked on less than or equal to 3 projects and those who have worked on more than 3 projects. We can see that employees who have worked on a low number of projects are more likely to be evaluated less than employees working on more than 3 projects. A noticeable feature of this graph is that there's a high frequency of low project number employees being rated from (0.4) to (0.6). There is a similar but slightly smaller frequency of high project number employees. This goes from (0.75) all the way up to (1.0). The KDE plot for evaluation rate vs average monthly hours produces similar results.



In the second KDE plot, I have split the employee frequencies to employees who work less than or equal to the Irish average monthly working hours (156 according to citizensinformation.ie) against employees who work more. Employees who work less are more likely to get an evaluation rate (≤ 0.6). In contrast, employees who work more are more likely to get an evaluation rate (> 0.6).


So in terms of the relationship between evaluation rate and other attributes in our dataset I have learnt two things. The greater the number of projects an employee works on, the more likely they are to work harder. While in contrast the lower the number, the more likely they are to be evaluated lowly. In addition, an employee working more than the average monthly hour rate will be more likely to be evaluated highly than someone who is working less than that.

Now that i've seen how the number of projects and monthly hours worked have an effect on evaluation rate individually, I wanted to see how they impacted together. I did this using an Implot. To make the graph easier to understand and look at, I made the input data the last evaluation and average monthly hours columns. I used the number of projects as the hue for this graph and I grouped some of the values together to show the clusters more clearly.



Observing this graph at first can be a little confusing. But, if you look closely you can see the present of 3 clusters:

- The first cluster is displayed in the top right of the graph and is made up of blue dots. This represents the top evaluated employees in our dataset (around 0.8 to just below 1.0). This shows that employees who work more hours and work on more projects are more likely to be rated highly than those who don't.
- The second and most visible cluster is the big green one taking up most of the centre of the graph. This represents employees who have worked on 4 or 5 projects. The cluster's evaluation rate ranges from just below (0.5) all the way up to (1.0) and it's work hours range from below 150 to below 300. While this cluster of employees do not show a

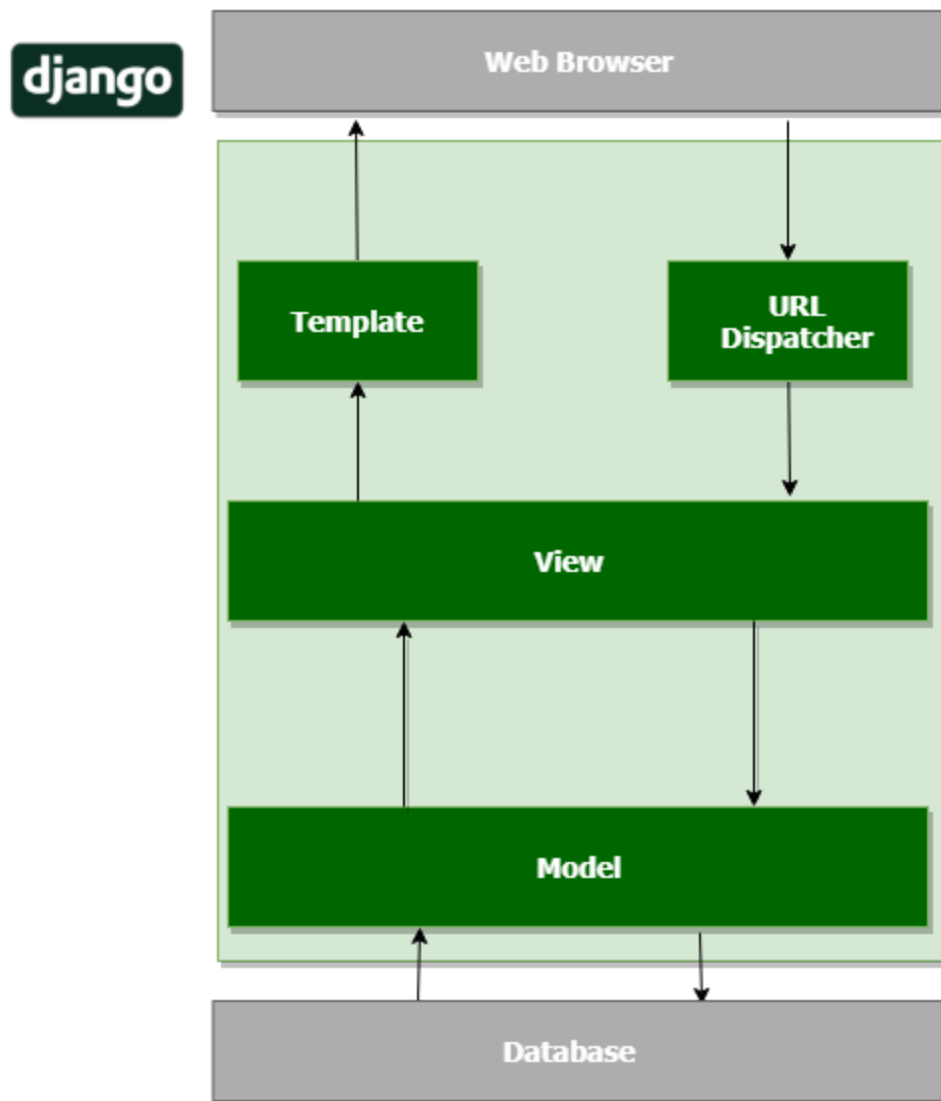


universal high or low evaluation, there seems to be a close to 50/50 chance of being rated high or low in this cluster.

- The last cluster is not so clear at first glance. The red dots representing employees who have worked on 2 or 3 projects are quite disperse. Despite this, we can observe a small red cluster formed in the bottom left of the graph. A small majority of red dots are contained here with the exception of a few outliers. This cluster's evaluation rate ranges from below (0.4) to just up to around (0.75). This cluster represent the employees who work on very few project and low working hours. The majority of these employees have a poor evaluation rating.


System Architecture

Django Architecture



As the Django web framework is such a huge part of my project, I thought it was best to describe it's architecture in detail. The Django web framework is made up of the components shown in the diagram above. They interact in the following ways:

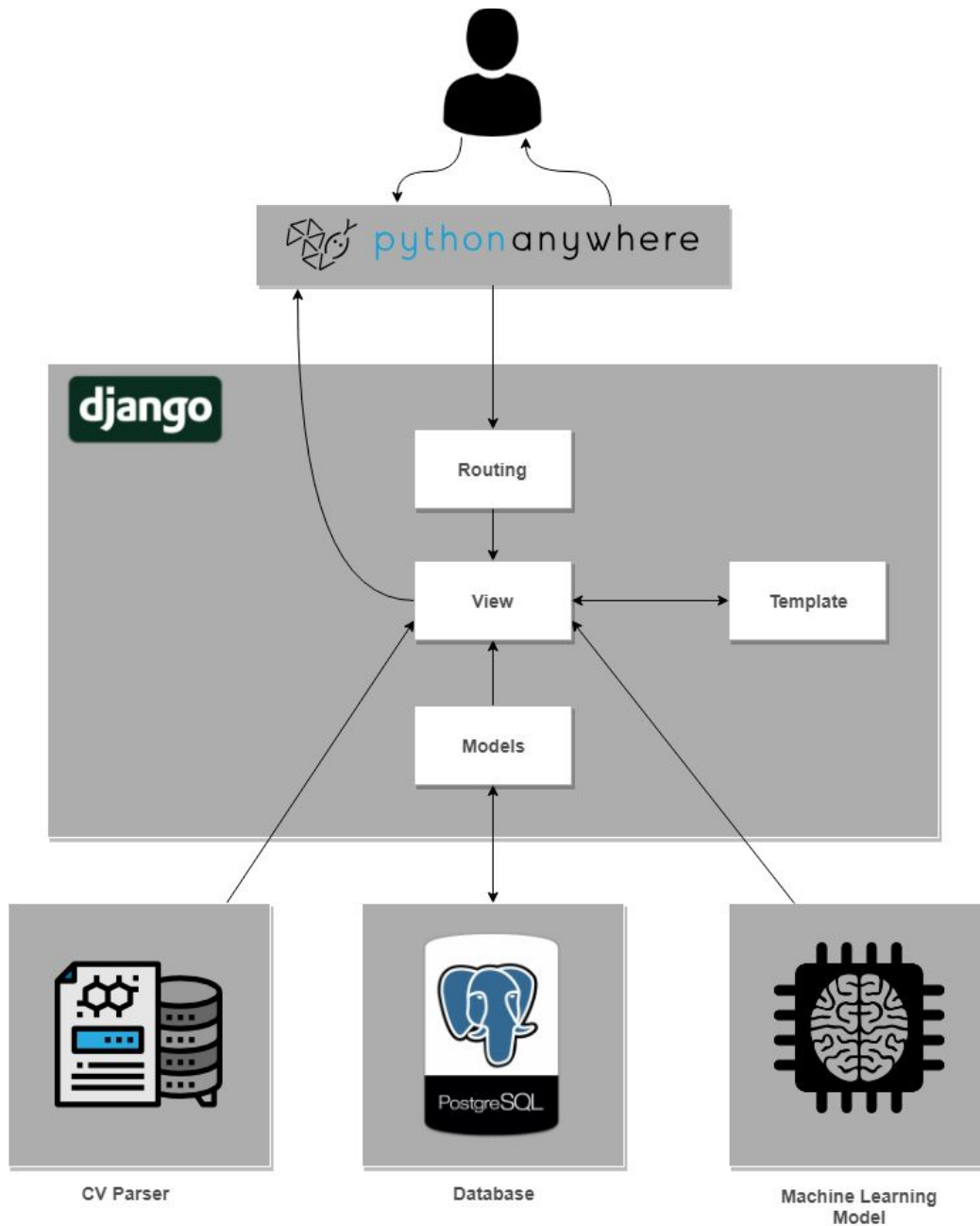
- The model defines the database tables. It uses an object-relational mapper which helps describes the database layout in python. It interacts with the data in the database.
- The view retrieves specific data and then loads renders templates of that data. It is responsible for return a `HttpResponse` object for a requested page and it's content.




Depending on how the view is implemented, it can perform other actions such as reading or writing to the database. In Job Analytics the view carries out the majority of the application functionality e.g. parsing curriculum vitae, predicting employee work ethic, writing models to the database as well pulling them from the database and form validation.

- Django uses a template system. Templates return HTML based on the Jinja templating language for python. This allows you to reuse code snippets and create base templates that can be used for multiple pages that have the same design. The view function passes the data returned from the HttpResponse object to the template to be displayed in the web browser.
- The URL dispatchers maps the requested URL to a view function and calls it. If caching is enabled, the view function can check to see if a cached version of the page exists and return the cached version of the page.

High-Level Design



This diagram represents the main components of the application and how they interact with each other. As you can see, every component is connected to Django and the internals of Django dictate the logic of the application. The database I am using is PostgreSQL, as it has good support for python to be used as a procedural language. It is also relatively easy to configure with Django.

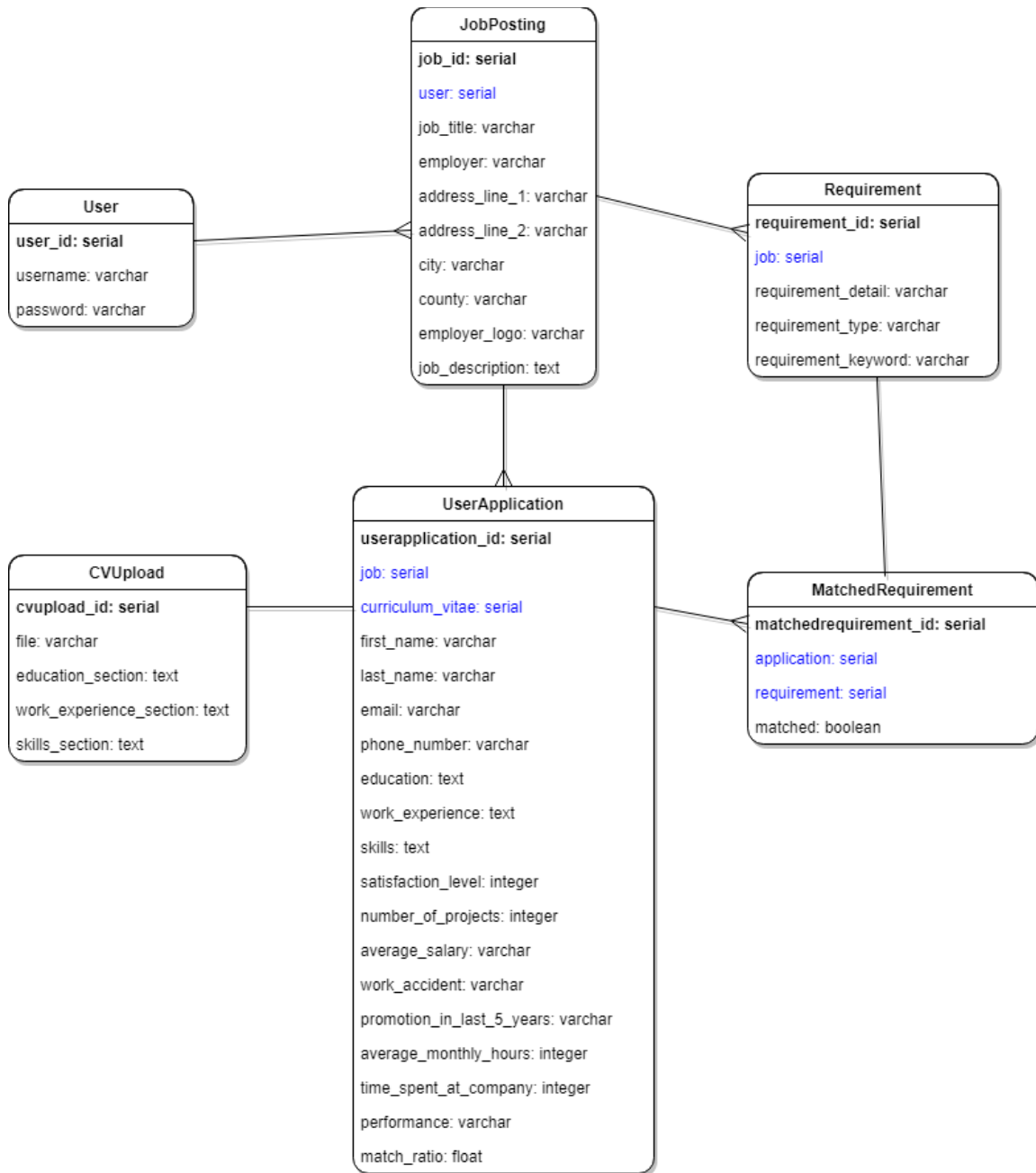


The main feature of my application isn't my database, so picking Postgresql to save time configuring a complex database was preferable. The `models.py` file in Django define database tables. Data can be stored and pulled from the database using these models. The other two external components are the CV parser and the machine learning model.

The CV parser was written using libraries such as the python Natural language processing toolkit (NLTK) and the fuzzywuzzy fuzzy matching library. The machine learning prediction model was made using the python library Scikit-learn. Both of these components are external to Django but are used in the `views.py` file in Django. The view functions in `views.py` call the functions specified in the CV parser and machine learning algorithm and handle their results and events. The view is the bridging point between the user data and the external components. The view also creates the model instances and allows the users to create instance using Django's model forms.

The routing (`urls.py`) in Django maps a url to the view. When it receives a request from the server, the corresponding view is returned to the client's machine. The web hosting service provider is PythonAnywhere, a hosting platform specifically for python applications, such as ones created with Django.

Database Schema




Primary Keys

Foreign Keys

———— 1 - 1 Relationship

————< 1 - M Relationship



This diagram shows the models of the Job Analytics database along with their fields and field types. The diagram also shows how each of them interact with each other. Below are the descriptions for each model.

User: Standard user model imported from Django. Has numerous optional fields but I have decided to utilise the username and password fields only. Used for user registration and login for employers.

JobPosting: Model for job postings on the application. Saves the job information posted by a user. It has the following fields:

- **User:** Foreign key to input in user table. It's the user who posted the job posting.
- **Job_title:** Title of the job posted.
- **Employer:** The name of the company/business that posted the job.
- **Address_line_1:** First line of an address of a company.
- **Address_line_2:** Second line of an address of company. Optional as not all address contain more than two lines (excluding lines given for city/town, county and postcode).
- **City:** The city/town of the job posting company.
- **County:** The county of the job posting company.
- **Employer_logo:** The logo an applicant will see beside the employer's job posting. Optional field and has a default image if one is not specified.
- **Job_description:** The description of the job posted.

Requirement: Model for job posting requirements. Made them separate object rather than field in job posting model as this would make it easier to match individual requirements to CVs. It contains the following fields:

- **Job:** Foreign Key to input in jobposting table. Each requirement must be mapped to a jobposting instance and a jobposting instance can have many requirements mapped to it (1 to many relationship).
- **Requirement_detail:** This field is used to explain the requirement in detail. It is what will be displayed on the job information page in the application.
- **Requirement_type:** The type of requirement specified. Can be 1 of 3 choices; skill requirement, education requirement or experience requirement. This specification will assist the requirement matching algorithm by telling it which CV section to search.
- **Requirement_keyword:** This is the keyword(s) that will be searched for by requirement matching algorithm in the applicant's CV.

CVUpload: Model for uploading curriculum vitae. Using the CV parser functions, information from the uploaded CV is extracted and saved to this model. It contains the following fields:

- **File:** This stores a charfield that is reference to the uploaded file, which is a CV.
- **Education_section:** This field stores the extracted education section from an applicant's CV.
- **Work_section:** This field stores the extracted work experience section from an applicant's CV.
- **Skills_section:** This field stores the extract skills section from an applicant's CV.

The sections are extracted exactly how they are displayed in the CV (with the exception of removing special characters such as newline and tab characters).

UserApplication: This model is used to store information about an applicant when they apply for a job. It is the biggest model in the database and contains information that will be used for the prediction algorithm and the CV parser. It contains the following fields:

- **Job:** Foreign key to input in Jobposting table. It is the job the applicant has applied for.
- **Curriculum_vitae:** Foreign key to input in CVUpload table. It represents that curriculum vitae of the applicant along with the extracted text sections.
- **First_name:** First name of applicant.
- **Last_name:** Last name of applicant.
- **Email:** Email of applicant.
- **Phone_number:** Phone number of applicant. Format of phone number is checked by a regex validator.
- **Education:** This represents the summarised version of the applicant's education section in their CV. The CV parser uses the *education_section* field in the CVUpload reference and extracts a summary of the applicant's education history.
- **Work_Experience:** This represents the summarised version of the applicant's work experience section in their CV. The CV parser uses the *work_experience_section* field in the CVUpload reference and extracts a summary of the applicant's work history.
- **Skills:** This represents the summarised version of the applicant's skills section in their CV. The CV parser uses the *skills_section* field in the CVUpload reference and extracts a summary of the applicant's skillset.

The next fields specified in this model are used for the prediction algorithm. As explained in previous sections, the dataset used for this algorithm contains attributes such as *satisfaction_level*, *time_spent_at_company*, *average_salary* etc. The classification models needs the applicant's information in regards to these attributes in order to make a prediction. The applicant must answer several questions when they are completing the application form for a job posting. The answers to these questions are saved to this model and used later for our prediction model. The fields are:

- **Satisfaction_level:** How happy the applicant are/were in their current/previously departed role on a scale of 1 to 10.
- **Number_of_projects:** The number of projects they have worked on in their current or previous role.
- **Average_salary:** The average annual salary the applicant made (low, medium or high).
- **Work_accident:** Whether they've had a work accident before.
- **Promotion_in_last_5_years:** Whether they've had a promotion in the last 5 years in any of their previous jobs.
- **Time_spent_at_company:** How long they have been at their current/previously departed employer for in years.

This information is fed into the machine learning algorithm and the output is also stored in this model.

- **Performance:** Predicted work ethic outputted by the machine learning prediction algorithm.
- **Match_ratio:** How much they match the job they applied for. Output result from requirement matching algorithm.

MatchedRequirement: This model is used to keep track of the requirements an applicant meets. It contains the following fields:

- **Application:** Foreign key to input in UserApplication table. It is the user application that has applied for the job that has the requirement that is stored in this model.
- **Requirement:** Foreign key to input in Requirement table. It is the requirement being checked against the user application stored in this model.
- **Matched:** This field stores a boolean depending on whether or not the applicant matches the job requirement or not.

Front-End Design

The front-end of my application is designed using the standard view and template system that Django provides. I also use Bootstrap to make the template look more presentable in a browser. I wanted to design my front-end properly. I designed my application view based off of Don Norman's Principle of Interaction Design.

Visibility: *"The more visible functions are, the more likely users will be able to know what to do next. In Contrast, when functions are "out of sight," it makes them more difficult to find and know how to use."*

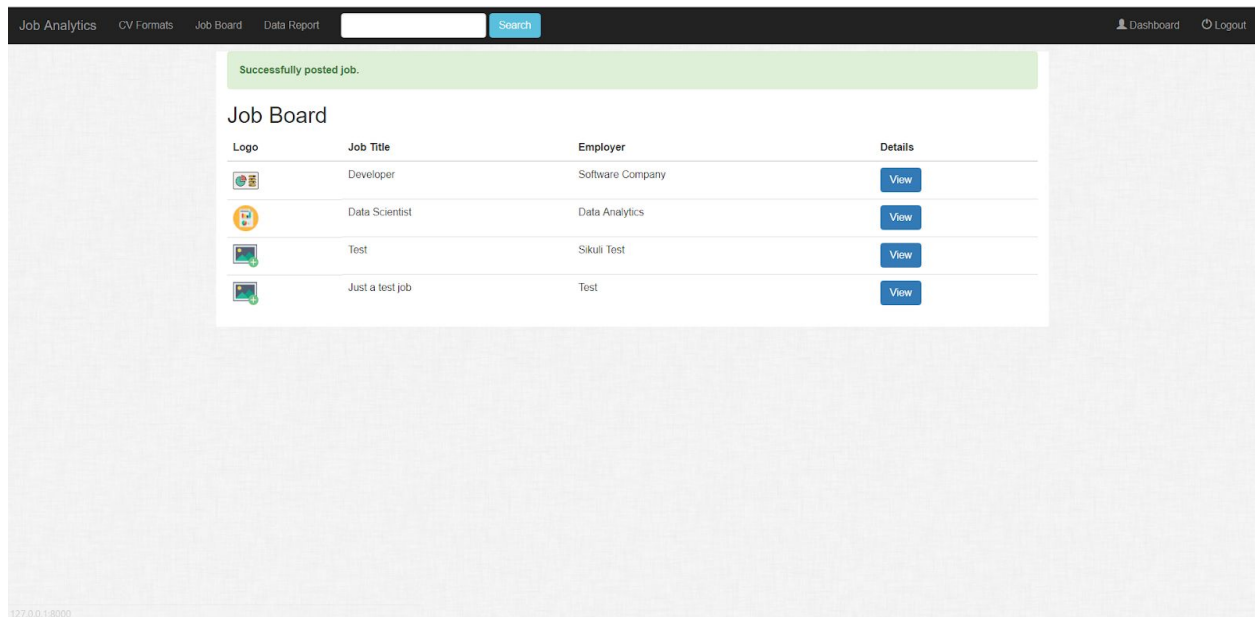
I wanted to avoid using dropdown menus, modals, slide-in panels etc, as much as possible. I prioritized what needed to be seen on the interface by the user so they can understand how to interact with and operate my application.



I tried to make the interface as simplistic as possible as to not cause confusion as to what element does what.

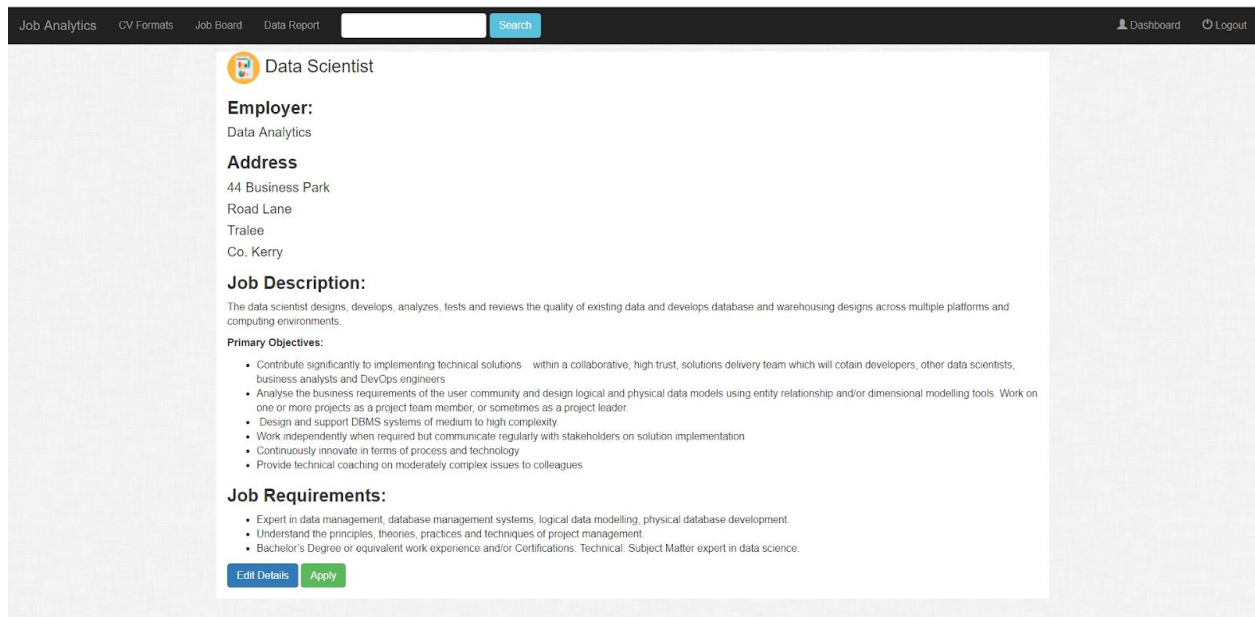
Feedback: *“Feedback is about sending back information about what action has been done and what has been accomplished, allowing the person to continue with the activity. Various kinds of feedback are available for interaction design-audio, tactile, verbal, and combinations of these.”*

I utilised the Django messaging framework as much as I could to display messages when a task was completed correctly or incorrectly. I also added checks for deletion of objects to let the user know what they were doing before they did it.

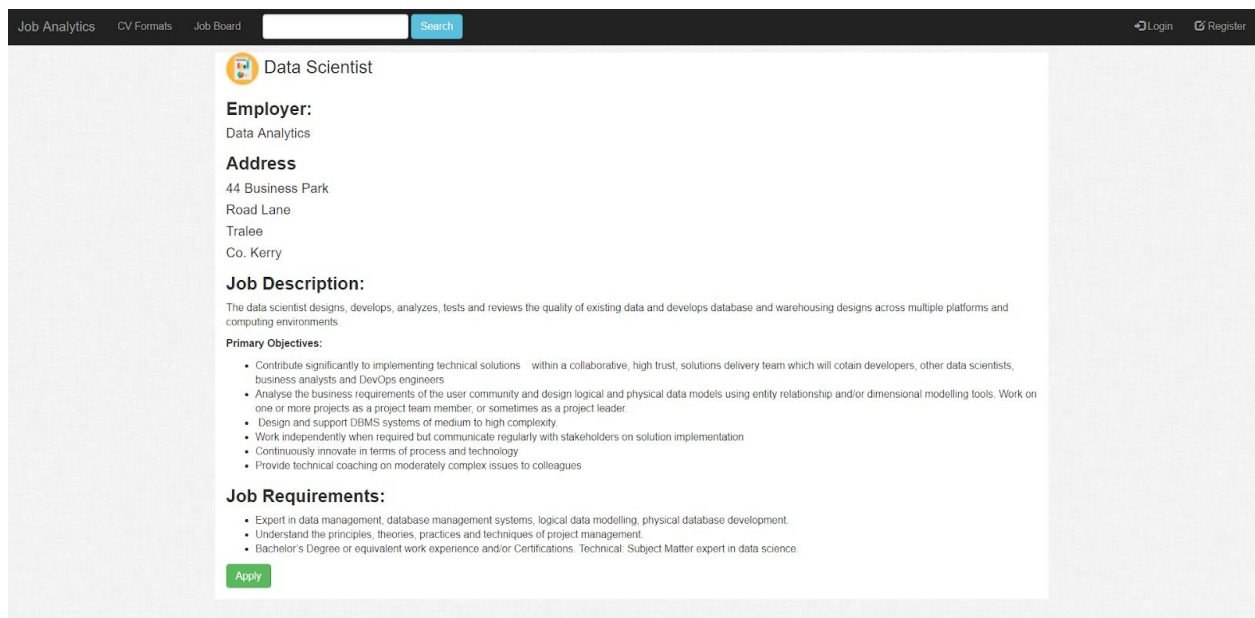


Constraints: *“The design concept of constraining refers to determining ways of restricting the kind of user interaction that can take place at a given moment. There are various ways this can be achieved.”*

There are numerous examples of constraining interaction to users in my application. Only users who posted jobs can delete and edit them. Only employers can view the data report specified in the navigation bar when a user is logged in. URLs are also restricted only to users who have permission to view them. If a user attempts to access a restricted URL and they don't have the necessary permissions, they are redirected to another page.



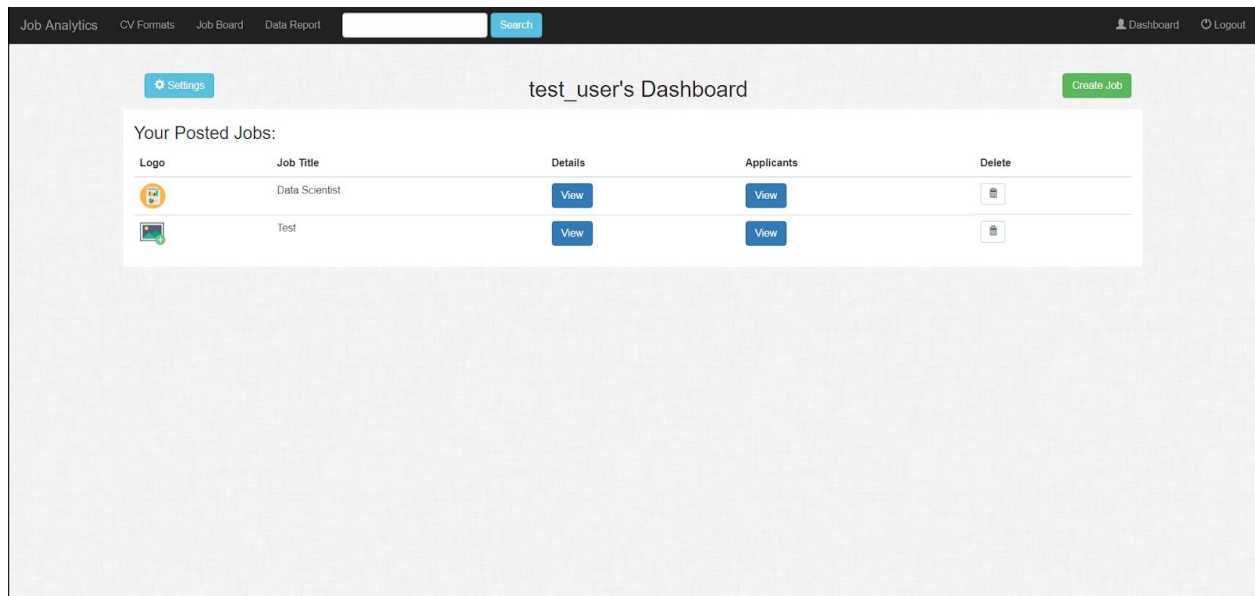
This is the job information view. The user who posted this job is logged in so therefore they can see the “Edit Details” button.



This is the same view, but this time there is no user logged in. They cannot see the “Edit Details” button. Also notice how the navigation bar changed. The “Data Report” tab is now missing and the right hand side tabs have changed to “Login” and “Register” from “Dashboard” and “Logout”.

Mapping: “This refers to the relationship between controls and their effects in the world. Nearly all artifacts need some kind of mapping between controls and effects, whether it is a flashlight, car, power plant, or cockpit. An example of a good mapping between control and effect is the up and down arrows used to represent the up and down movement of the cursor, respectively, on a computer keyboard.”

All buttons and elements are labelled or contain a self explanatory glyphicon.




Notice the trash icon. Even though this is the universal symbol for deleting something, the column which it is in is still labelled “Delete” for further mapping.

Consistency: “This refers to designing interfaces to have similar operations and use similar elements for achieving similar tasks. In particular, a consistent interface is one that follows rules, such as using the same operation to select all objects. For example, a consistent operation is using the same input action to highlight any graphical object at the interface, such as always clicking the left mouse button. Inconsistent interfaces, on the other hand, allow exceptions to a rule.”

I designed the interface using the Bootstrap framework which allows for consistency.

Implementation vs Initial Design

Compared to my initial design specified in my functional specification, there are vast differences compared to my implementation. These differences include different approaches in user implementation, components used, design patterns etc. I will compare the main differences



between my implementation and initial design and explain why I choose to go with a different path to my original design.

User Implementation

In my initial design of the application, I had planned to make both applicant employer users. However, i only implemented employer users into my design. I came to the decision not to have applicant users when I started doing more research on similar job posting websites and applications to mine. While some of the most well known job posting sites do have applicant user functionality, the majority of them do not require a user to be created to apply for a job. Making applicants create a user account to apply for job is a bit excessive. Doing this would be assuming that the applicant will apply for more than one job and that all the jobs they're interested in are posted on this application. In reality, job applicants use multiple job posting sites and will use the one with the more appealing jobs to them. Therefore, I decided not implement applicant users. I did implement employer users as a user needs to be created to post a job and keep track of the applicants for that job.

Frontend Framework

In my initial design, I had intended to construct my frontend using the ReactJs framework. I opted instead to use the template-view system provided by Django. My reasoning behind this is that although ReactJs is a very powerful and useful framework, configuring it with a Django backend is complex and time consuming. Spending time making sure everything is configured correctly would take time away from developing the application's main features such as the CV parser or machine learning algorithm. Therefore, I opted to not use ReactJs and focus on a more simpler approach to my frontend. Django's Jinja templates offer great reusability and are simple to use, so they were a great alternative.

Database

In my initial design, I had planned on using Google Firebase for my application's database. I opted instead to use PostgreSQL. After learning more and more about the Django framework, I realised that PostgreSQL would be a much better option compared to Firebase. Firebase is relatively new and although there are tutorials on how to use it with Django, there isn't many. PostgreSQL has support for using python as procedural language and is so simple to integrate with Django. Therefore, I decided to use PostgreSQL instead of Google Firebase.

Features left out

One or two features planned in my initial design were left out of my implementation. One was the feature to allow an applicant to upload their CV and see what job they match. This functionality

was developed and integrated with the application at one point. However, it was not implemented in an elegant manner and had high time complexity. I couldn't figure out a more efficient way of implementing this feature, so I decided to omit it from the application. Another feature omitted was the feature to allow employer user to specify questions to be asked when an applicant applies for one of their job posts. This was omitted due to other features taking priority.

Components

CV Parser

The CV parser was implemented using rule based parsing. The curriculum vitae had to be uploaded in a specific format (which is checked by the application at upload time). Using libraries in the Natural Language Toolkit (NLTK) and python regular expressions, the application extracts important information from the curriculum vitae. The format for the CVs is shown below:

Mandy Good
23 Black Dragon Lane, Canterbury, Kent
Mobile: 0871234567 mandy978@gmail.com

Profile
A Business Administration graduate from the University of Kent. I have skills and knowledge essential for managing key areas of an organisation and the problem solving skills needed in finance. I am looking for a graduate trainee post in marketing where I can use my strong influencing skills.

Education
2004 - 2007 University of Kent
BA (Hons) Business Administration 2:1
1998 - 2004 St Brigid's Secondary School
Leaving Certificate 350 points

Work Experience
2006 - 2007 Store Assistant, Iceland Supermarket
Worked in a busy team sometimes under pressure. Provided a quality service to customers.
2005 - 2006 Barmaid, The Plough Pub
Was often left in sole charge of the bar and learned the valuable art of dealing sensitively but firmly with drunken individuals near closing time. Worked as part of a team as well. This involved planning, organisation, co-ordination and commitment e.g. ensuring sales targets were met, a fair distribution of task amongst the team and effective communication with other staff members.

Skills

- **Computing:** ECDL qualifications in MS Word, Access, Powerpoint. and Excel. Can write basic web pages.
- **Languages:** Good conversational French.
- **Driving:** Full current clean driving licence.

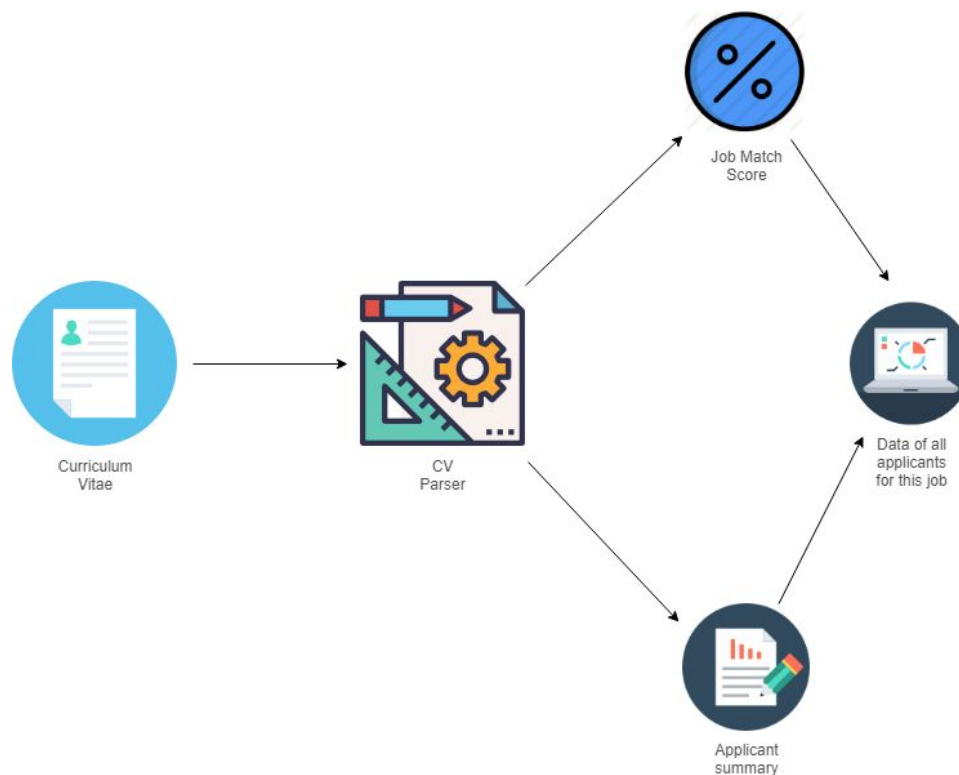
Any curriculum vitae uploaded must have the following headings:

- Profile

- Education
- Work Experience
- Skills

The application would then extract important information from these sections and display it in a summary for the job poster to view. The information extracted includes:

- A brief summary of their education section. This just includes the names of any educational institutes attended along with the dates of attendance.
- A brief summary of their work experience section. This includes the title of their position, the name of their employer and the dates they worked in that role.
- A list of their skills. This is extracted from their skills section. The section is tokenized and each token if verified by a relational dataset that contains 28,935 unique skills that appear in the likes of job postings and user applications.



This summary section for an applicant also contains their predicted work ethic as well what requirements they match, as well as the ones that they don't.

Requirements

- Must know python ✓
- Must have 2 years experience as an Associate Engineer ✓
- Must have obtained at least 435 points in their Leaving Certificate ✓
- Must have a degree in Enterprise Computing ✓
- Must know Javascript ✗
- Must have 3 years experience of Perl ✗

Predicted Work Ethic

Satisfactory Worker

Education

2014 – 2018 Dublin City University (DCU)

2008 – 2014 Ratoath College

Work Experience

2013 – 2018 Intern, SAP

2009 – 2013 Java Developer, Workday

Skills

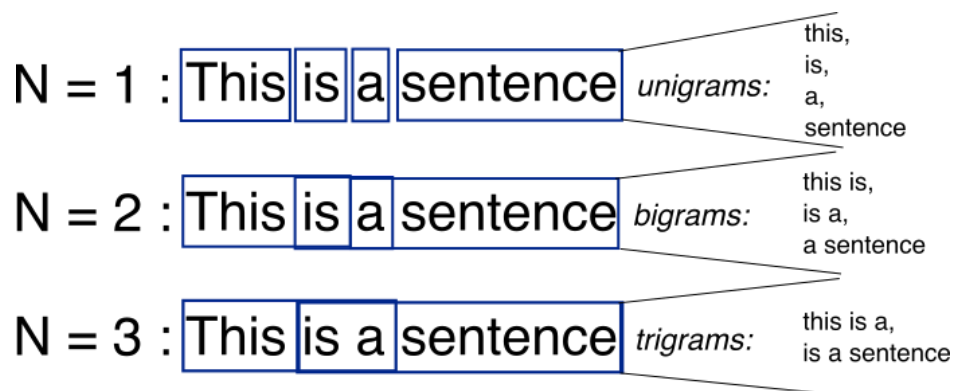
- python
- sql
- teamwork
- java
- django

Requirement Matching Algorithm

The application uses a word error rate evaluation metric to check if an applicant's curriculum vitae matches the requirements specified. This uses the levenshtein distance which checks the minimum number of editing operations to transform an machine translation output to a reference translation. In this case, our reference translation is a job requirement and the machine translation outputs are the sentences in the curriculum vitae. This is also called fuzzy matching.

$$\text{WER} = \frac{\text{insertions} + \text{deletions} + \text{substitutions}}{\text{reference length}}$$

Depending on the requirement type specified (skilled, education or experience), the requirement matching algorithm splits the corresponding CV section into n-grams. An n-gram is a contiguous sequence of n items from a given sample of text.



Using a python library called “FuzzyWuzzy”, each token is compared to the requirement keyword specified by the job poster. The library has 4 different ways of comparing string similarity using techniques such as:

- **String Similarity:** This uses the levenshtein distance explained previously. It measures how much two strings are similar based on the number of edits it takes to make them identical. The function it uses to achieve this is called “ratio”.

```
fuzz.ratio("NEW YORK METS", "NEW YORK MEATS") ⇒ 96
```

- **Partial String Similarity:** This uses a heuristic called “best partial”. It’s for when two strings are of noticeably different lengths. This measures by checking how many edits are needed to create a best matching substring which is the length of the shorter of the two

strings. The function it uses to achieve this is called “partial_ratio”.

```
fuzz.partial_ratio("YANKEES", "NEW YORK YANKEES") ⇒ 100  
fuzz.partial_ratio("NEW YORK METS", "NEW YORK YANKEES") ⇒ 69
```

- **Token Sort:** This approach involves tokenizing the strings, sorting them alphabetically, joining them back into a string and then measuring how similar the strings are. The function it uses to achieve this is called “token_sort_ratio”.

```
"new york mets vs atlanta braves" ↔ "atlanta braves mets new vs york"
```

```
fuzz.token_sort_ratio("New York Mets vs Atlanta Braves", "Atlanta Braves vs New York Mets") ⇒ 100
```

- **Token Set:** This approach tokenizes both strings but instead of sorting them it splits the tokens into two groups: intersection (set of common substring) and remainder (not common substrings). It then constructs a string based on these sets and then compares similarity. The function it uses to achieve this is called “token_set_ratio”.

```
t1 = "angels mariners vs"  
t2 = "anaheim angeles angels los mariners of seattle vs"
```

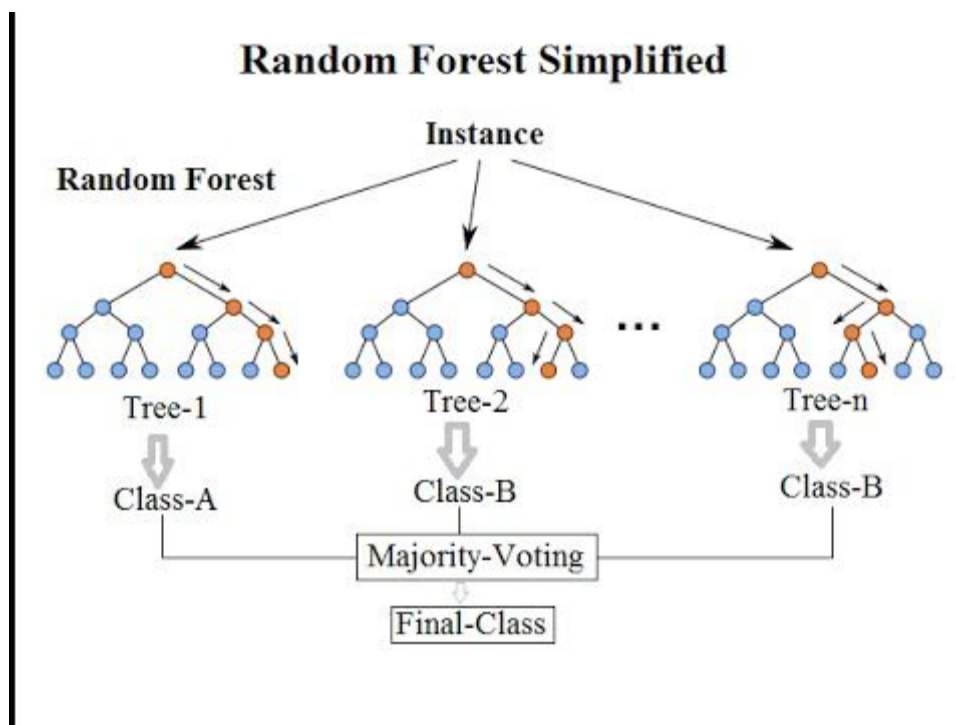
```
t0 = [SORTED_INTERSECTION]  
t1 = [SORTED_INTERSECTION] + [SORTED_REST_OF_STRING1]  
t2 = [SORTED_INTERSECTION] + [SORTED_REST_OF_STRING2]
```

```
fuzz.token_set_ratio("mariners vs angels", "los angeles angels of anaheim at seattle mariners") ⇒ 90
```

To cater for as much text variations as possible in curriculum vitae, I decided to use all 4 of these approaches and take the average of their results added together. This way, the algorithm will be able to check for more variations of requirements being matched. For example, let's say an employer posts a job and one of the requirements states that an applicant “must be familiar with using data analytics techniques”. An applicant applies for the role who previously worked as a data scientist. This applicant would of course be a match for the specified requirement but that all depends on whether the content in their CV produces a good similarity result. They might specify countless data skills but might not mention the phrase “data analytics techniques”, which could result in a poor matching score. However, combining the functions the fuzzywuzzy library has to offer can reduce the risk of missing out on qualified applicants.

Random Forest Classifier

To get a better understanding of how machine learning works, I constructed my own algorithm from scratch for learning purposes. I decided not to use this algorithm though, as it's common industry standard to use renowned Python libraries to implement prediction models. Despite this, constructing a model from scratch really helped me learn what was going on in the algorithm. For my prediction algorithm, I decided to go with a random forest classifier using the python library Scikit-learn. Random Forest is a classification method for classification, regression and other tasks, that operate by constructing a multitude of decision trees.



I constructed this algorithm using the Scikit-Learn library. Below is the classification report for the algorithm.

```
--Random forest--
```

	precision	recall	f1-score	support
Excellent Worker	0.71	0.78	0.74	1589
Poor Worker	0.50	0.22	0.31	27
Satisfactory Worker	0.71	0.64	0.67	1384
avg / total	0.71	0.71	0.71	3000


I had to do some preparation on the dataset before I started building the classification model. Preparing the dataset was very easy. It contained no null values. The majority of our data was numeric except for *sales* and *salary* columns. I converted the values in the *salary* column to numeric values by assigning them categorical codes. As for the sales (*department*) column, I removed it as it has a limited range of job sectors. In the context of the application I was building, I didn't think it was a good idea to limit job posting to this small range. Another thing I had to decide was how I was going to classify employee work ethic.

As mentioned in my research section, the dataset contains a column labelled *last_evaluation* which contains float values ranging from 0 to 1 based on an employee's performance. Predicting this attribute is more suited to a regression model than a classification model. However, I decided to convert this column to categorical data and construct a classification model. This is because when it comes to differentiating between certain evaluation values, there is little difference to consider. For example, is there much of a difference between an employee who has an evaluation rate of 70% and an employee who has an evaluation rate of 80% in terms of work ethic? Not really. Therefore I decided to split the *last_evaluation* column into classes based on their value range.

Class Label	Evaluation Range
Excellent Worker	100 - 70
Satisfactory Worker	70 - 40
Poor Worker	40 - 0

It was difficult to label these classes as the data is so dispersed as there are very few values below (0.5). Despite this I think these class labels are appropriate for what I am trying to predict. In my experience in a working environment, I've learnt that you only really recognise two kinds of employees:

- The excellent workers who stand out and go beyond their work duties specified in their role.
- The poor workers who stand out because of some or multiple negative attributes they have (e.g. lazy, bad punctuality etc).



Then there are also the workers who I have labelled, for the lack of a better term, satisfactory. These are the workers who are just below excellent but not enough to really stand out and catch your eye. They meet an acceptable standard of work ethic. After I prepared data and then proceeded to build the prediction model. I split the data into training and test subsets. The training subset contains 80% of our data and test subset contains 20%. If you wish to see the results from the testing the algorithm, please refer to my testing documentation.

Problems and Resolutions

Below are list of problems I ran into while developing this application along with their resolutions.

Requirement format

Specifying how to allow users to define job requirements proved to be a challenge. Requirements can be defined in a wide variety of formats and accounting for them all would be difficult. Allowing users to specify as many requirements as they needed in a job post also proved challenging as I struggled to figure out how to accomplish this using Django model forms.

Resolution

The resolution for the requirement format was to put a constraint on the kind of format users can specify. Not an ideal resolution at all, but one that was effective in helping me accomplish my application's functionality as well resolving other problems. As for specifying the number of requirements, a small library called `jquery.formset` allowed me to implement dynamic form fields for requirements.

CV Format and parser

With the vast amount of CV formats, deciding on how to handle this proved difficult. Constructing a CV parser to handle all formats would be incredibly difficult. I had take into account stuff like how the parser would match the CV to job requirements, split sections in the CV and summarise the CV.

Resolution

Only one CV format was allowed to be uploaded. This was checked at upload time by checking the section headings in the CV. If the CV didn't match the format it was rejected and the user was asked to try again after reviewing the format.

Future Plans and Retrospective

Looking back on this project, I am proud of what I have created. However, if I were to do this project all over again or if I were to continue to work on this project, this is what I would do differently:

Improve CV Parser

The CV Parser implemented in this application is rule-based. It's simple and is effective at doing its job, but can only allow handle one CV and requirement format. If i was given more time to work on this project, I would have done a lot more research in to the constructing of either a machine learning recognition tool or a neural network. These would cater for more CV variations and would have more accurate results. One of the main reasons why I didn't attempt to make one in this project is because to construct one of these things you need a vast amount of different variations of CVs and they must be rated by a professional reviewer as either good or bad.

Analyze Requirements more

Given more time, I would have improved the way my application analyzes requirements. I would have liked to recognise the skill level specified in each requirement more i.e. there's a difference in "Must know C++" compared to "Must be proficient in C++". I also would have made specifying requirements easier for users. At the moment, users must specify job requirements in very specific detail and must specify what keywords are to be searched for in CVs. I would have liked if my application analyzed the requirements more and detected what keywords to search for using some natural language processing techniques, instead of the user having to specify keywords.

Add more features to web application

Given more time, I would have added more features to my application to make it feel more like a proper web application. Features such as:

- Notifications
- Password reset
- Allow the user to specify more information on creation
- Better job searching
- More statistics of applicants

References of software used:

- **Django Web Framework:** <https://www.djangoproject.com/>
- **FuzzyWuzzy: Fuzzy String Matching in Python:**
<http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/>
- **Chart.js:** <https://www.chartjs.org/>
- **Bootstrap:** <https://getbootstrap.com/>
- **PostgreSQL:** <https://www.postgresql.org/>
- **Natural Language Processing Toolkit:** <https://www.nltk.org/>
- **Django-Dynamic-Formset:** <https://github.com/elo80ka/django-dynamic-formset>
- **Pythonanywhere:** <https://www.pythonanywhere.com/>
- **TinyMCE:** <https://www.tinymce.com/>
- **Scikit-learn:** <http://scikit-learn.org/stable/index.html>
- **Tablesorter:** <https://mottie.github.io/tablesorter/docs/>