1. About HTFome

HTFome (https://htfome.com/) is a web application (webapp) designed to give users access to data regarding Human Transcription Factors (HTF's), drugs which target HTF's, and allow users to upload gene expression data for analysis in-browser.

The webapp was developed as part of the Queen Mary University of London (QMUL) bioinformatics masters degree program. Specific application specifications can be found on the QMUL website. (https://qmplus.qmul.ac.uk/course/view.php?id=16766).

Developers of this app were A.M. Andersen, K. Antoniuk, N. Hardie and A. Sutradhar. The source code for the app can be found on Github: (https://github.com/NHardie/HTFome)

2. Software Architecture

HTFome was developed using the Django framework, and R Shiny, hosted on an Amazon Web Services (AWS) Elastic Beanstalk (EB) server. The data used in the app was gathered through Application Programming Interface (API) calls to several large, established data repositories, and is stored locally via a SQLite3 database. An app user can access this data via the browser and view one of several web pages, displaying the data as a list, or click-through to see more specific detail about an individual HTF or drug of choice.

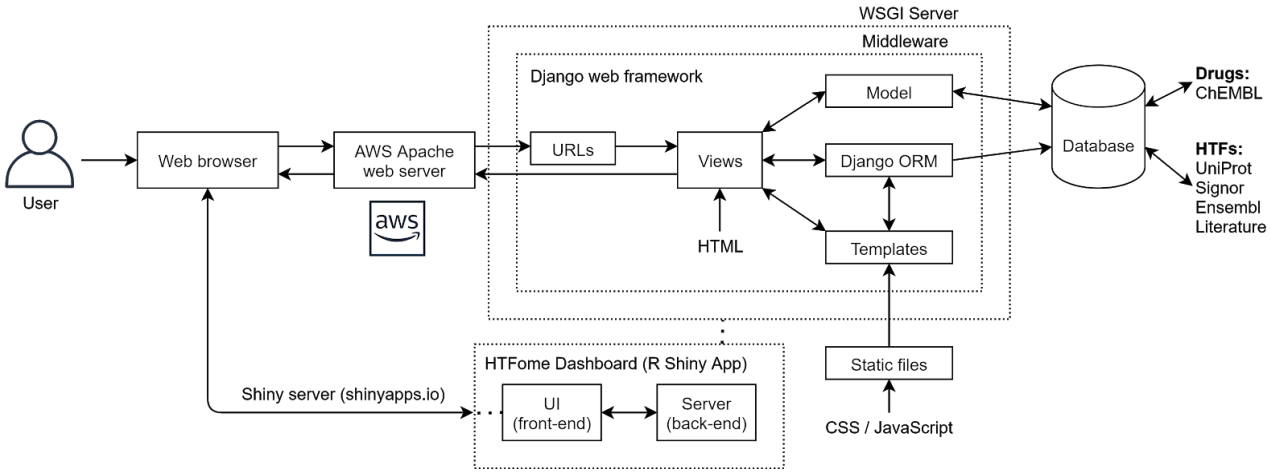

Figure 1. Schematic showing software Architecture.

2.1. Overall design philosophy

The HTFome app was designed to provide biologists with a simple method of accessing data related to HTF's and the drugs that target them. The aim was to make it easy for a biologist to enter some parameters, such as a gene symbol, and quickly be provided with useful information about the relationships that gene has to other HTF genes, and to drug compounds.
During development we were mindful that this app was never supposed to rival websites such as Ensemble, UniProt or ChEMBL in terms of data output, but instead to provide a small selection of biologically relevant data, and hyperlinks to the original data sources available.
The app is designed to be modular and scalable, to aid any future development efforts. Github was used throughout development to aid with version control and app releases.

2.2. Django web-framework:

The project itself is developed predominantly in Django, a python-based framework for creating web app's. Initially we opted to use the Flask framework, partially as we were more familiar with webapp development with this framework, and also because Flask is recognised as a micro-framework, perfect for developers who are new to web apps, although there is some compromise between built-in features and overall design flexibility.
We switched from Flask to Django for several reasons, firstly Django includes several useful features missing from Flask, which make development easier, such as a built-in administration system and a database ORM (Object-Relational-Mapper). Django is also more scalable, Django releases have forwards-compatibility in mind and within a Django project there can exist multiple apps, which allows for more modular development, as well as the ability to copy and repurpose apps later in production.
In our experience working with both frameworks, Django seemed a more logical choice for this particular project, however this is no criticism of Flask, and Django does have some drawbacks, discussed below.

2.2.1. Overall design

The general software structure is represented below, the initial idea was to have a home page that users would "land" on, a search page specifically for HTF's, a search page specifically for drugs, and access to an analysis page for GDS data.

Some slight modifications were included during development. We originally just wanted the user to be able to click on an HTF/ drug and receive more details, specifically those of biological significance, but during development we decided to also link the HTF/ drug to the original database for even more information, for example the details page for each HTF contains a link to the page for that HTF on Ensemble (for gene data) and Uniprot (for protein product data).

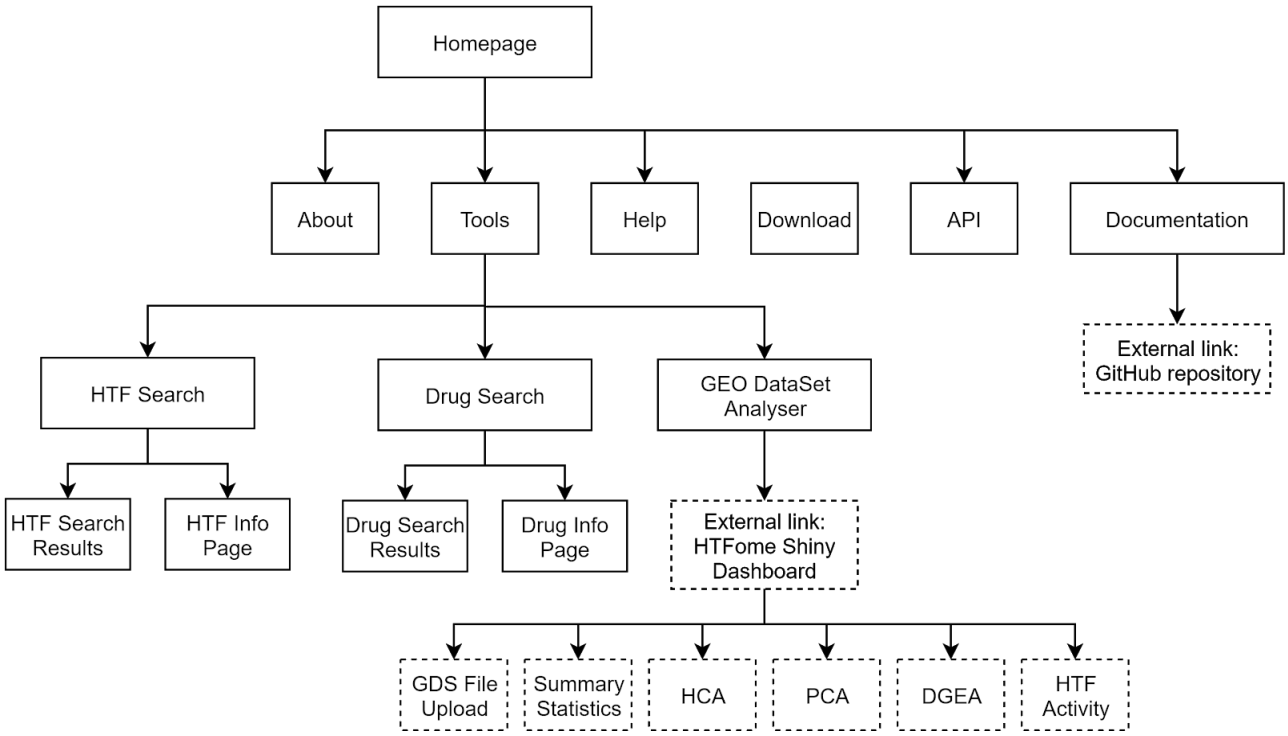

Figure 2: Site map.

We also decided to include a data upload page, accessible by an administrator only, this page allows the admin/ superuser to upload a new database to the website remotely and securely.

We determined that use of a web-hosting platform such as AWS EB would be beneficial, and we connected our Github repo to the EB environment using AWS CodePipeline, automatically deploying/ redeploying our web app when the “Main” branch of the repo was updated via a pull request/ merge. This allowed backend and frontend changes to be viewed live in mere minutes, and encouraged many small updates to the “Main” branch, rather than fewer, larger updates. Although not necessarily the best practice professionally, this helped tremendously with troubleshooting, as any changes that introduced bugs were limited in their size and scope, allowing more rapid fixes.

With the added benefit of hindsight, having a main production branch and a separate testing branch, each hosted on their own servers would have been good practice, however the limited time and experience only made this apparent late in development. AWS EB requires users to set up an environment relevant to their webapp, for example, Django apps have their own package requirements, as do Flask apps. The user determines the server they require, typically Linux, and what language they require, in this instance Python. The user then uploads their initial app along with a requirements.txt containing the packages required. This is then deployed to the AWS cloud, and can be accessed through the browser with the designated URL.

2.2.2. How to work with our django

Developers should activate a virtual environment within the directory where they wish to instantiate their project, then download the Django package, or if continuing development, should activate their virtual environment and install the required packages found in the requirements.txt.

Virtual environments allow developers to create isolated environments in which they can work on a specific project without any influence from the already installed packages on their machine. Standardisation of environments ensures all developers are using the same versions of languages and package versions, as well as avoiding any odd conflicts, or errors generated from multiple developers with separate environments working together. Simple instructions on the creation of virtual environments in Pycharm were written, and a requirements.txt was generated through the use of the command:

$ pip freeze > requirements.txt

This copies all the installed packages and their current versions to the file requirements.txt. Once a virtual environment has been created and activated, the developer can then use the command:

$ pip install -r requirements.txt
This installs all required packages of specific version, ensuring all developers are using the “same” environment. As the list of packages used grew, the requirements.txt was updated to reflect this.

Django development begins by running the command:

$ django-admin startproject PROJECT_NAME

Where PROJECT_NAME should be replaced by the desired project name. In the example of our group, this is “htf_web”. This will create the required Django files and folders, including a manage.py file, which is of great importance during development. The folder created will take the input name, i.e. /htf_web/, and will contain an __init__.py, allowing the project to be used as a package, a settings.py, containing project settings, a urls.py, which will contain website-wide urls, and two files which allow web apps written in python/ with python frameworks to communicate with compatible web servers: asgi.py and wsgi.py.

The manage.py file contains commands that can be run via the terminal, these include methods for running a development server locally, making database changes (migrations) and creating admin or superusers.

Following the startproject command, future development should continue through the creation of modular applications. This allows the developer to separate out different aspects of their website as they see fit, and also makes the reuse of code easier in the future.

To create a new app, the developer runs the command:

$ python manage.py startapp APP_NAME

Where APP_NAME should be replaced as required. The chosen name should reflect the app's function. As our app is essentially a data repo, the name “data” was chosen (after checking there were no conflicts in naming convention, i.e. “test”).
We could have seperated the app functions into smaller modules; an app for HTF data, one for drug data and one for GDS analysis, this would have the benefit of specifying the app functions, rather than having one app with a wider range. However the data used in this project are all related, so there is an argument that the apps should be combined to avoid replicating the database model code in each app. The app database models could be linked through import statements, however one of the benefits of modular development is the ability to copy an app across multiple projects, unless the developer is copying the database as well, this would sever the relationship between app and database (Here it is assumed the developer wants to copy the app exactly, however in reality the app would most-likely be chosen as a generic template for future development, and thus would be repurposed, with a new data set, changing variable names etc as necessary).
For simplicity, one app was seen as sufficient, however for future development multiple, more focussed, apps could be beneficial.

The startapp command creates a subdirectory (called APP_NAME) of the project directory. The startapp command will create several template files and a migrations subdirectory. The migrations subdirectory will contain database migrations, i.e. the way a django developer updates the database objects. The files created include: __init__.py, allowing the use of the app as a package, admin.py, allowing the creation of admin profiles for the app, apps.py which contains the app data for use in the project-level settings.py file and tests.py, a specific file for development testing. The startapp command also creates two additional files, which will be discussed in more detail: views.py and models.py.

2.2.3. Django pros / cons

In our experience working with both Flask and Django, Django seemed a more logical choice for this particular project, however this is no criticism of Flask, and Django does have some drawbacks. Although Django is older and seen as more stable, the documentation and official tutorials for Flask are easier to follow, and often troubleshooting Django will turn-up links to older releases, which may not be backwards-compatible (but most often are). The official documentation for Django is much larger, as many features that are standard in Django only exist in additional packages for Flask. Therefore, the Flask documentation mirrors the framework's philosophy, light-weight, easy to get started with, but lacking inclusion of more complex features.

One benefit of Django is the philosophy of DRY (Don't Repeat Yourself), Django is written in such a way that each required aspect should be written out fully once, then referenced later. This does however produce code which can be more confusing to budding web app developers, with separate files to store URL's, views, models and templates all linking to one another through import statements. However, this methodology is good practice and saves time in the long-run.

Use of AWS EB to deploy and host our web app also presented challenges, EB deployments can be finicky, for example, with Django there is a requirement to name the path to the /static folder, containing static files such as images, .css and .js code. While Django has a convention to keep the /static folder within the specific app directory, AWS EB struggles to find this folder and serve these files in deployment. Although there are many "fixes" found online, a number of these are contradictory, or don't work in specific cases. For our HTFome app, the solution to this issue was pulling the static directory out of the app subdirectory and placing it in the project root directory.

A similar issue was found when deploying the Flask app to AWS EB, Flask servers can be run locally for development by running the code in the file often called main.py, however, with AWS EB, this file had to be called application.py, and references to “app” in the code had to be changed to “application” for EB to run the file correctly.

2.2.4. How to develop further

Further development of HTFome will mostly include adding more web apps/ pages, greater functionality and improving upon the database design.

Adding web apps/ pages requires some knowledge of Django development, but generally follows a standard workflow:

- Create the new web app as described above
- Add the web app to the list of installed apps in /settings.py

```
/settings.py

INSTALLED_APPS = [
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'htf_web',
   'data.apps.DataConfig',
   'django_filters',
]
```

For our project we added the htf_web project and django_filters to allow use of these as packages. The main app “data” is added by the line `'data.apps.DataConfig'`. The DataConfig file is a part of the data app, so Django treats this app as installed without the need to specify it.

- Add the app to the project-level /urls.py

```
/urls.py

urlpatterns = [
   path('admin/', admin.site.urls),
   path('', include("data.urls")),
]
```

For HTFome, we added the URL's for the data app to the path following an empty string. Typically a developer will create a urls.py file within the app directory, then link the project urls.py to the app name, providing some additional modularity to the django app. If the website is very simple, then the desired URL could be provided straight to the project-level /urls.py

- Add required app URL's for web pages to the app/urls.py (data/urls.py)

```
data/urls.py

urlpatterns = [
   path('', views.home, name="home"),
   # path('path to be used as url/', views. view function name, view name)

   path('htf/', views.htf, name="htf"),
   path('htf/<str:gene_name>/', views.htf_detail, name="detail"),
   # Allows the action of clicking through to a specific HTF with the url of  #
htf/gene_name/
... ]
```

The url patterns allow the developer to select what URL should be linked to which view, and allow the addition of subsequent variables to be passed to the URL, as in our example above, a user who clicks on a specific HTF link for more detail will be sent to a unique URL containing the HTF's gene symbol.

- Write the view/s for the web page/s

A webapp's views can be described as the backend equivalent of the webpage the user sees when they access a URL. The view.py in Django will include a list of instructions for the webapp to execute when a user accesses a certain webpage. A very simple example is a home page function, which displays a HTML (Hypertext Markup Language) header as a response to a HTTP (Hypertext Transfer Protocol) request:

```
/views.py

def home(request):
   # define the function to deal with home receiving a request
   return HttpResponse("<h1>Home</h1>")
   # A request returns a HTTPResponse
```

Through the use of different Django functions, these views can be made more complex:

```
/views.py
```

```
def home(request):
   # Define function, what to do when a request comes
   return render(request, "data/home.html", {'title': 'The Human Transcription
   Factor Database'})
# return render of template located in data/templates/data/home.html, pass
# title variable to html page
```

Here the view returns an HTML page template, with a dictionary object containing a variable that's passed to the template, and can be accessed within the HTML through use of Django's DTL (Django Template Language). A similar language, Jinja2, used by Flask is originally modelled on DTL, so some experience with one informs the other. Usage and benefits of template languages are discussed below.

Views can also incorporate database objects, and functions from various packages, these require importing in the /views.py file, as is standard in Python:

```
/views.py

def htf(request):
      all_htfs = Htf.objects.all().order_by('gene_name')
      # This is the model (database table) containing the HTF's, we take the
      table of all the HTF's and order them alphabetically by gene_name

      paginator = Paginator(all_htfs, 15)
      # This is the Paginator function working on the HTF list, displaying 15
      per page

      page_number = request.GET.get('page')
      # Need to know the page_number, i.e. page 1, 2, 3...

      page_obj = paginator.get_page(page_number)
      # This releases a page object for us to use in the html template
      # page_object contains the list of HTF's for each page.

return render(request, "data/htf.html", {'page_obj': page_obj, 'title': 'HTF
Search'})
# Add a dictionary containing htf's as a page object, can now display on html
page
```

This function defines the output for the /htf/ url. The view takes a database model object, in alphabetical order by the field gene_name, then paginates these data in groups of 15. For the ability to switch between pages, a GET request is required to inform which page the user is currently on. A final function allows the view to access a specific page, given it's number.

We return the render of htf.html, and a dictionary containing the page_obj and title, for use in the HTML.

Use of HTML templates in Django relies upon the views, the views rely on the URL's. Views can be written as functions or as classes, for simplicity we chose to write our views as functions, however class views could be incorporated in later development to avoid code replication.

- Write the HTML templates for each web page

As mentioned, Django has an in-built template language, allowing a developer more control over how to present their data on the web. Generally it is good practice to separate the HTML templates by app, in our case: /data/templates/data/base.html.

This base.html is used as the standard template, setting the design of the other templates. These other templates then "extend" the base.html. Objects from the views can be utilised in the HTML through use of double curly-braces {{ }}, and simple code can be executed by use of curly-braces with percentage signs {% %}.

```
{% for htf in htfs %}
      {{ htf.gene_name }}
{% endfor %}
```

- Test

Once a view has been created, the HTML written, and the URL specified, the developer should then test their changes. In Django, this is completed by using the python command:

$ python manage.py runserver

This should be executed with the virtual environment activated.
This command will start an instance of the Django project, hosting the website locally. Developers will often be checking their code changes, and during development it's useful to turn debug mode on (True) in the /settings.py file. This aids debugging, as error messages will be shown to the developer in the browser. Debug mode should be turned off (False) during production however.

2.3. Front-end framework

The frontend development of HTFome was mostly performed using HTML, CSS (Cascading Style Sheets) and JS (JavaScript). As mentioned, the various views return to the user a HTML template, this contains the layout and information to be displayed by the browser to the user.

2.3.1. Integration with Django

We designed a base.html, then using DTL codeblocks, extended this template to other webpages, adding a uniformity of design to each page. The base.html also includes a codeblock which loads the static files for use. Django development usually includes the creation of a /static/ directory, where the developer would keep their “static” files, that is, files that once created don’t tend to change, such as .css, .js, or image files.
The base template includes areas to input content from other pages, called content blocks. Then in other HTML templates, the developer need only specify what on the page is content, and this will be inserted in the correct area on the base.html.
This method aids reproducibility and uniformity.

```
/data/htf.html

{% extends "data/base.html" %}

{% block content %}

   <main role="main" class="container">
       <h1 class="mt-4">{{ title }}</h1>
       <div class="container">
           <form action="{% url 'htf_results' %}" method="get">
               <input name="q" type="text" placeholder="Search HTFs...">
           </form>
       </div>



       {% for htf in page_obj %}
           <div class="card mb-4">
               <div class="card-body" style="margin-left:15px">
                   <h5 style="font-weight:600">Name: <a href="/htf/{{
htf.gene_name }}/">{{ htf.gene_name }}</a></h5>
                   <p class="card-text text-muted h6">Ensemble ID: {{
htf.ensemble_id }} </p>
                   <p class="card-text">Protein name: {{htf.prot_name}}</p>
               </div>
           </div>
       {% endfor %}
```

```
        <div class="pagination">
            <span class="step-links">
                {% if page_obj.has_previous %}
                    <a href="?page=1">&laquo; first</a>
                    <a href="?page={{ page_obj.previous_page_number
}}">previous</a>
                {% endif %}

                <span class="current">
                    Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}.
                </span>

                {% if page_obj.has_next %}
                    <a href="?page={{ page_obj.next_page_number }}">next</a>
                    <a href="?page={{ page_obj.paginator.num_pages }}">last
&raquo;</a>
                {% endif %}
            </span>
        </div>
    </main>
{% endblock content %}
```

Here the `{% extends "data/base.html" %}` and `{% block content %}` allow the developer to write page-specific HTML and insert it into the base template, avoiding rewriting all the HTML for the base template for each other page template.

One point to note is that the code blocks, once opened, must be closed, i.e. {% block %} requires {% endblock %}, {% if %} requires {% endif %}.

Another feature of Django is the { % include % } code block, that allows you to reuse code from another .html file. For example, a separate HTML file has been created for the navbar and footer, which gets included in base.html. This means, when base.html is extended to other pages, the navbar and footer also get included. This means, if site links need to be changed - they can be changed directly in the navbar and footer html files once (rather than in each individual html page).

2.3.2. Site Layout

Page	**Page Function**
base.html	Provides HTML base that extends to all other pages.
navbar.html	Provides navbar content that gets included to all other pages.

footer.html	Provides footer content that gets included to all other pages.
home.html	Content for Home page.
about.html	Content for About page.
htf.html	Content for HTF list.
htf_data_upload.html	Allows upload to HTF database.
htfsearch.html	Content for HTF search function.
htf_details.html	Content for detailed description of HTFs.
drug.html	Content for drug list.
drugsearch.html	Content for drug search function.
drug_data_upload.html	Allows upload to drug database.
drug_details.html	Content for detailed description for drugs.
genexp.html	Gateway to GEO DataSet Analyser.
documentation.html	Content for documentation page.
download.html	Content for data downloads page (not available yet).
help.html	Content for help page.

2.4. Database design

Our webapp required some method of storing data for the user to access through query requests. The data contained within are also relational, i.e. HTF targets and the drugs that target them, therefore a database written in SQL (Structured Query Language) seemed most appropriate for our needs. Although there are many SQL databases to choose from, we selected SQLite3.
Python version 3 comes with SQLite built-in, and this database is used in many webapps written in Django or Flask. Typically SQLite will be used in development, then in production a database such as mySQL will be hosted on it's own server for use. SQLite databases are seen as light-weight and less scalable, however their relatively small size allows them to be easily copied as a single file, useful for producing a webapp without a dedicated database server.

Although SQLite is less secure, it's easy to set up and perfect for our current needs as a small webapp/ prototype, however future development might see an increased requirement for database size and security, and so a separate SQL database hosted on it's own server should be considered, a process often integrated with popular web-hosting services, such as AWS.

2.4.1. Database Manipulation

models.py

In Django, the models.py contains the database schema for the various database objects available to the webapp, in our example we made one model to handle HTF data, and one model for drug data.

```
/models.py

class Drug(models.Model):
# Set Drug as a model

   drug_name = models.TextField(default='N/A')
   # Set drug_name as a model text field, no max limit of characters like
# with a CharField, default value to 'N/A' if no input

   trade_name = models.TextField(default='N/A')
   drug_chembl_id = models.TextField(default='N/A')
   year_approved = models.TextField(default='N/A')
```

Here we set the database object 'Drug' as a class, with the required fields as TextField and the default value of 'N/A'.
Other field types are available, such as numerical, character (accept all characters but with a maximum length), datetime etc.
ForeignKey fields can be used if one model belongs to another model, as in 'Car' and 'Car Manufacturer', OneToOne and ManyToMany fields can be used as well to map out database relationships, or these relationships can be produced manually in the views.py. Which method is best depends on how the data is gathered/ uploaded to the database, and also what is the most simple. For our needs it was easier to produce the relationships manually in the views/ HTML pages than to alter the data input.

Once models have been set, Django has a specific set of commands related to updating the database. Firstly the database being used must be referenced in the settings.py.

```
/settings.py

DATABASES = {
   'default': {
       'ENGINE': 'django.db.backends.sqlite3',
       'NAME': BASE_DIR / 'db.sqlite3',
   }
}
```

Then the user must make the SQL for the models they've specified.

$ python manage.py makemigrations APP_NAME

This command translates the simple changes made in the models.py to SQL, and also creates the database tables for any Django apps (such as admin features) which come as standard in Django.

The user must add their app to the settings.py list of installed apps, then can run:

$ python manage.py migrate

This will create the database tables for the user app models by applying the SQL changes to the database.

For HTFome, database changes are performed via two html pages, one for the HTF data, and one for the drug data. For security these are not linked on any of the webpages, and require admin permissions to access. Creation of an admin is performed via the command:

$ python manage.py createsuperuser

A new superuser can be created locally, or via SSH into the Elastic Beanstalk instance. If only a small number of changes are required, this can be performed manually through the admin page of the website: /admin/, this however is prohibitively slow and not recommended for batch updates.

The developer must then execute the commands:

$ python manage.py makemigrations
$ python manage.py migrate

To then confirm the migrations.

The developer can then start their local server, log in as admin via the admin page localhost:8000/admin, then upload the htf data localhost:8000/htf_data_upload, and drug data localhost:8000/drug_data_upload.
The data provided should take 3 to 4 minutes to upload, and the developer will see a POST request message in their terminal when complete.

/views.py

```
@login_required
# Must be logged in as admin to access this page, returns an error otherwise
def drug_data_upload(request):
   data = Drug.objects.all()
   prompt = {
       "Order": "Order of the CSV should be:"
                "drug name, drug Trade names, drug chembl id, "
                "drug approval date"
   }
   # Order of csv file format required to populate database
   if request.method=="GET":
       return render(request, "data/drug_data_upload.html", prompt)

   csv_file = request.FILES["file"]

   if not csv_file.name.endswith(".csv"):
       messages.error(request, "This is not a .csv file")

   data_set = csv_file.read().decode("UTF-8")
   # Read file, creates a data stream
   io_string = io.StringIO(data_set)
   # Can now use object in memory
   next(io_string)
   for column in csv.reader(io_string, delimiter='\t', quotechar="|"):
       if len(column) >= 4:
       # Had to include this, to avoid index out of range errors
           _, created = Drug.objects.update_or_create(
               drug_name=column[0],
               trade_name=column[1],
               drug_chembl_id=column[2],
               year_approved=column[3]
           )
       # Iterates through the original csv file, copies the data to the       #
database
   context = {}
   return render(request, "data/drug_data_upload.html", context)
```

This view simply takes an input .csv file and using the update_or_create function overwrites the current database object column by column. The column headers here should be the same as those in the models.py.
Note that if replacing the database entirely the developer will have to delete the db.sqlite3 file as well as any migrations in the app/migrations directory, then perform the steps above to create the migrations, apply the migrations and create a superuser before logging in as an admin via the /admin page, and then uploading the new drug and HTF databases via /drug_data_upload and /htf_data_upload. If using this method both the database tables will be wiped, so both must be reuploaded. This particular aspect to the web app is especially in need of further development, as it works, but not ideally.
One issue is the data input must be in the .csv format, however, multiple columns have commas within them, so the data must be arranged tab-separated, then saved with a .csv suffix, leading to files in the formal “htf_data.tsv.csv”. Using a .tsv for the upload seems to add quotation marks around the data content (On a Linux machine, windows might handle this differently), this introduces errors when using the data as hyperlinks.

2.4.2. Data Gathering

HTFome requires HTF and drug data to provide to the user, and for simplicity this data has to be freely and easily accessible by the public. We utilised the Application-Programming Interface’s (API’s) of major biological data repositories to write requests specific to the HTF and drug background data we required. Once this data was gathered, we used Python to clean/manipulate the data before it was added to the webapp database.

For the HTF data, several recognized resources were combined to produce a thorough and curated list that possessed sufficient supporting evidence. Firstly, a starting point for a comprehensive list of HTFs had to be established. A list was generated from Lambert, Samuel A. et. al 2018, as this review went over a catalog of over 1,600 HTFs and looked at how they were identified and at their functional characterization.

After establishing an initial list of HTFs, UniProt, which is a collaboration between the European Bioinformatic Institute, the Swiss Institute of Bioinformatics and the Protein Information Resource and a protein sequence and annotation database, was used to gather essential information of HTF function and subcellular location(s). This was done by utilizing UniProt’s “Retrieve ID/mapping” tool. From the HTFs list, the gene names were extracted and used as identifiers and the UniProtKB data type was used as target format. UniProt.org contains a lot of information, both revised, as well as

computationally proposed information. Only HTF results that had been reviewed were chosen for further implementation.

The data on relationships between HTFs and their targeted genes was gathered from Signor (The SIGnalling Network Open Resource), which is an open source database that contains curated signalling networks for the entire human genome and several other species.

Supporting information regarding chromosome and gene position, for all HTFs were gathered from Ensemble Biomart, which is a datasite based at European Molecular Biology Laboratory's European Bioinformatics Institute and produces, as well as maintains automatic annotation on selected eukaryotic genomes.

The drug data were gathered from ChEMBL, a database organised and maintained by the European Molecular Biology Laboratory (EMBL), containing data on small molecules with drug-like effects, including drugs used in medicine, drugs used in research, including in trials, and predicted drug compounds. ChEMBL includes a wealth of data for each compound, as well as drug targets, making it perfect for our requirements.

2.5. GEO Analysis

The specifications for the application to fulfil were (QMUL, 2021):

- Build a web application that can be used to explore background information about human transcription factors and the genes they regulate.
- Provide the user with background information about all known human transcription factors.
- Provide the user with information about drugs that target human transcription factors.
- Allow the user to upload gene expression data extracted from the (Gene Expression Omnibus) GEO database in (GEO DataSet) GDS format.

These specifications were met through the creation of a Shiny dashboard app, where the user can upload a GDS file of their choice, explore and interact with their data with various user parameters, and visualise their results graphically.

R Shiny apps have two main components: a user interface (UI) - that contains the elements that will be displayed to the user; and a server function - that contains the

back-end code to process user requests (e.g. analysis, or instructions for when a user selects a UI parameter). Shiny apps also use reactive expressions that express server logic - its function is to specify instructions on what the app should do when an input changes (i.e. all related outputs are automatically updated). An in-depth guide to Shiny and its reactive expressions can be found here: https://mastering-shiny.org/.

2.5.1. HTFome Dashboard

The code for the HTFome dashboard app can be found here: https://github.com/NHardie/HTFome/tree/gds-r-dev. It can be accessed and run from the app.R file. However, as this file is quite large, the app.R file can be further split into its components: server.R, ui.R and global.R - where code for the server functions, UI and global scripts are stored, respectively. If all three R files (server, ui and global) are installed, the app can be run from any of these files.

The HTFome Dashboard is split up into several tab items:

- Upload GDS File
- Summary Statistics
- Hierarchical Clustering Analysis
- Principal Component Analysis
- Differential Gene Expression Analysis
- Estimate of HTF activity

Each of these tabs contain a sidebar panel, which provides the user various parameters to further explore their data; a main panel where visual outputs are presented once the data has been processed, and in some cases tab items (where visual outputs can be split up and viewed in a more orderly way).

Other UI elements include: a loading spinner (to indicate to the user that their data is being processed), validation controls (to let the user know what to do, e.g. upload a file or select certain parameters).

2.5.2. Main features

Below shows the server reactives along with their associated reactive outputs for each tab item in the dashboard, a description of the reactive functions are also included.

Upload GDS file tab:

Server reactives	Reactive output	Description
validate_upload()	All outputs	Checks if file has been uploaded by user. If not, it pastes a message for user to upload file.
gds()	N/A gets passed to gds_df()	Object created when user uploads GDS file, processes file using getGEO(). Output is the GDS data structure (class GDS).
gds_df()	output$gds_preview	Converts gds() reactive to a dataframe, using function Table(gds()).
eset()	N/A gets passed to pDat() reactive.	Convert gds() reactive (data structure) to ExpressionSet using GDS2eSet() function.
pDat()	output$pDat_preview	Extract phenotype data from eSet object, using pData() function.
data_sum()	output$data_summary	extract file summary information from gds, eset and pDat objects.

Summary statistics tab:

Server reactives	Reactive output	Description
N/A	output$pheno_plot	Display phenotype plot, of pDat() reactive.

Hierarchical clustering analysis tab:

Server reactives	Reactive output	Description
gene_exp()	N/A gets passed to sort_gene_SD() reactive.	Combine gene names from gds and expression data from eSet, then remove any duplicated genes by taking the average expression of probes corresponding to same gene. Output is a matrix containing averaged gene expression data.
sort_gene_SD()	N/A gets passed to top_genes_mat() reactive.	Calculate standard deviation of genes across all samples and sort rows by highest SD. Output is a data frame containing genes and their expression data, sorted by SD. NB: Shiny doesn't seem to like data passed to apply() in matrix form, so we need to convert it to a data frame first.
top_genes_mat()	N/A gets passed to heatmap() reactive.	Extract user-defined genes with highest SD and convert to numeric matrix
heatmap()	output$heatmap	Plot heatmap (using heatmap.2)

N/A (# TODO)	output$pDat_cols_hca	Display column names user can colour samples by.

Principal component analysis tab:

Server reactives	Reactive output	Description
pca()	N/A gets passed to pca_summary() and pca_scores() reactives.	Perform PCA: get averaged gene expression data, transform it and remove any columns containing zeros and NAs first.
pca_summary()	output$pca_data	Get summary of PCA results.
scree_plot()	output$screeplot	Plot scree plot.
cum_var()	output$cum_var_plot	Plot cumulative variance.
pca_scores()	output$pca_plot	Plot PCA scores.
my_cols() NA (# TODO)	output$pDat_cols_pca	Set colours by sample type, to display column names user can colour samples by.

Differential gene expression analysis tab:

Server reactives	Reactive output	Description
FUTURE DEVELOPMENT WORK		

Estimate of HTF activity tab:

Server reactives	Reactive output	Description
validate_htf_button()		File upload validation.
viper_output()	output$viper_plot, output$viper_summary	Run VIPER analysis. Uses function viper_analysis() (defined in global.R).
sample_choice()	output$get_treatment_name, output$get_control_name	Define sample choices for HTF activity analysis. Let user select treatment variable and control variable.

2.5.3. HTF Activity

Estimating protein activity from gene expression data is a complicated task and many considerations, such as post-translational modifications and cascade effects, need to be taken into account. We have chosen to utilize the validated algorithm “Virtual Inference of Protein-activity by Enriched Regulon analysis” (VIPER), which is an R-package that is available via Bioconductor, and has been validated for protein activity estimations. VIPER is a regulatory-network based algorithm that performs a regulon enrichment analysis on gene expression signatures. The first step in the enrichment analysis is comparing two groups of samples (distinct in either phenotype or treatment) by any form of measurement in difference between groups. For our algorithm we used a simple student’s t-test. Second step is to define a nullmodel to account for the correlation structure between genes. This nullmodel is defined by a set of signatures that has been obtained by random shuffling of the test and reference samples for 1,000 repetitions. After creating the nullmodel as well as the student’ t-test on the expression signatures, a regulatory network needed to be acquired. For this we decided to use the DoRothEA package that is also available via Bioconductor. DoRothEA is a data structure that contains interactions between HTFs and their targets, which is referred to as a regulon object. The data for the DoRothEA regulons is collected and curated from different sources, such as reviewed literature, ChIP-seq peaks, TF binding site motifs and interactions inferred from gene expression.
The VIPER algorithm utilizes everything from above and computes an object of class ‘msVIPER’, which contains the gene expression signatures, the regulon and its estimated enrichment and p-values.

2.5.4. Deployment

RShiny app can be deployed either with shinyapps.io, or by installing Shiny Server on a Linux server, such as AWS. Shinyapps.io method is simpler and it allows deploying an app straight from the RStudio on a local machine. However, this method has limited configuration options and needs all of the application system dependencies to be installed on shinyapps.io servers before deploying the app. Because some of the dependencies we need are not installed on a server, it is currently not possible to deploy our app with shinyapps.io. The solution to that problem might be requesting installation of those dependencies on https://github.com/rstudio/shinyapps-package-dependencies and waiting for a response. If added, it should be possible to deploy the app with shinyapps.io

The more advanced alternative is to set up a Shiny Server on a Linux server, such as AWS. This approach (unlike shinyapps.io) gives an option to install the required dependencies on a host server and gives more configuration options. However, setting up a server is much more complicated than with shinyapps.io and requires experience with Linux virtual machines. Installing shiny package and shiny server on linux can also be problematic. Therefore, filling a request for those dependencies to be installed on shinyapps.io servers and deploying the app there might be a more viable choice than setting up a dedicated Linux server.

2.5.5. Tools: pros / limitations

The DoRothEA package has been created from curated literature and experimentally validated sources, which adds to the trustworthiness of the estimated protein activities that have been derived on the basis of these regulatory networks, as the relationships in these networks have been validated. Additionally, the quality of the resources as well as their diversity can increase the accuracy of their computationally predicted relationships, which increases the apparent reach of the dataset, as it possibly could predict the protein activities based on relationships not yet confirmed by literature. However, as the HTF-targeted gene relationships isn't built from fully validated data only, this could potentially provide false estimates for protein activities. Additionally, when looking through the data, it can be found that only around 1,300 HTFs are listed in the dataset, while other sources, such as UniProt, have upwards of 1,600 HTFs. This likely causes substantial miscalculations regarding the protein activities, as a lot of relationships are undoubtedly not accounted for.

2.5.6. Developing HTFome Dashboard further

To develop the dashboard further, smaller improvements and bug fixes could be made such as:

- Adding more validation (e.g. testing to verify GDS format for an uploaded file, testing to check for an empty file).
- Colour data by sample types (to make PCA and HCA outputs more informative).

Further larger features could be added, for example:

- Deploying the app online (currently we are unable to deploy the app to shinyapps.io as the server lacks the dependencies we require).
- A tab item within the dashboard to allow the user to view a volcano plot visualising the differential gene expression analysis.

- Making plots fully interactive by plotting using plotly or other interactive visualisation tool.
- Adding a download button so users can download their analysis as reports.

3. Overall Limitations

HTFome is limited mostly by the database. Firstly, we're not fully utilising the relational nature of the data and SQL's handling of these relationships due to only relating the data in the webpage views/ HTML. While this works well for the design of our data input, future developers would require lots more instruction regarding the correct file format for the uploading of new data to the database. Secondly, the database upload method isn't as straight-forward as desired, although it functions, uploading requires the correct file manipulation process and is somewhat arbitrary, requiring lots of steps to instantiate a new database correctly. Thirdly, the SQLite database hasn't been stress tested, either for many users, or for lots more data. It works well in development, but if the database or users grow there may be issues with speed of access.

4. Areas for future development

Future development should focus on the database, changing from uploading the database directly, to hosting an SQL database server via AWS S3 and automatic database updates should be configured. The data input and models.py should be changed to allow the data to relate to each other via foreign keys/ ManytoMany fields.

Another area to focus on developing further is combining the RShiny GEO data analysis with the Django web app, rather than linking to the GEO analysis, this should be available on htfome.com directly.

5. References

Lambert, Samuel A. et. al 2018:
Samuel A. Lambert, Arttu Jolma, Laura F. Campitelli, Pratyush K. Das, Yimeng Yin, Mihai Albu, Xiaoting Chen, Jussi Taipale, Timothy R. Hughes, Matthew T. Weirauch, The Human Transcription Factors, Cell, Volume 172, Issue 4,2018,Pages 650-665,ISSN 0092-8674,
https://doi.org/10.1016/j.cell.2018.01.029.

Uniprot:
The UniProt Consortium, UniProt: the universal protein knowledgebase in 2021, Nucleic Acids Research, Volume 49, Issue D1, 8 January 2021, Pages D480–D489, https://doi.org/10.1093/nar/gkaa1100

Signor:
Licata L, Lo Surdo P, Iannuccelli M, Palma A, Micarelli E, Perfetto L, Peluso D, Calderone A, Castagnoli L, Cesareni G. SIGNOR 2.0, the SIGnaling Network Open Resource 2.0: 2019 update. Nucleic Acids Res. 2020 Jan 8;48(D1):D504-D510. doi: 10.1093/nar/gkz949. PMID: 31665520; PMCID: PMC7145695.

Ensemble Biomart:
Yates, D. Andrew et. al, Ensembl 2020, Nucleic Acids Research, Volume 48, Issue D1, 08 January 2020, Pages D682–D688, https://doi.org/10.1093/nar/gkz966

Dorothea:
Garcia-Alonso L, Holland CH, Ibrahim MM, Turei D, Saez-Rodriguez J. “Benchmark and integration of resources for the estimation of human transcription factor activities.” Genome Research. 2019. DOI: 10.1101/gr.240663.118.

VIPER:
Alvarez, M.J., Shen, Y., Giorgi, F.M., Lachmann, A., Ding, B.B., Ye, B.H., and Califano, A. (2016) Functional
characterization of somatic mutations in cancer using network-based inference of protein activity. Nature genetics
48(8):848--47.

BiocManager:
Martin Morgan (2019). BiocManager: Access the Bioconductor Project Package Repository. R package version 1.30.10.
https://CRAN.R-project.org/package=BiocManager

Ggplot2:
Wickham H (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4, https://ggplot2.tidyverse.org.

GEOquery:
Davis S, Meltzer P (2007). “GEOquery: a bridge between the Gene Expression Omnibus (GEO) and BioConductor.” Bioinformatics, 14, 1846–1847.

Tidyverse:
Wickham et al., (2019). Welcome to the tidyverse. Journal of Open

Dplyr:
Source Software, 4(43), 1686, https://doi.org/10.21105/joss.01686

Limma:
Ritchie, M.E., Phipson, B., Wu, D., Hu, Y., Law, C.W., Shi, W., and Smyth, G.K. (2015). limma powers differential expression analyses for RNA-sequencing and microarray studies. Nucleic Acids Research 43(7), e47.

Gplots:
Gregory R. Warnes, Ben Bolker, Lodewijk Bonebakker, Robert Gentleman, Wolfgang Huber, Andy Liaw, Thomas Lumley,
Martin Maechler, Arni Magnusson, Steffen Moeller, Marc Schwartz and Bill Venables (2020). gplots: Various R
Programming Tools for Plotting Data. R package version 3.1.1.
https://CRAN.R-project.org/package=gplots

EnhancedVolcano:
Kevin Blighe, Sharmila Rana and Myles Lewis (2020). EnhancedVolcano: Publication-ready volcano plots with enhanced colouring and labeling. R package version 1.8.0. https://github.com/kevinblighe/EnhancedVolcano

Plotly:
Author: Plotly Technologies Inc. Title: Collaborative data science Publisher: Plotly Technologies Inc. Place of publication: Montréal, QC Date of publication: 2015 URL: https://plot.ly

RColorBrewer:
Erich Neuwirth (2014). RColorBrewer: ColorBrewer Palettes. R package version 1.1-2.
https://CRAN.R-project.org/package=RColorBrewer

Shiny:
Winston Chang, Joe Cheng, JJ Allaire, Carson Sievert, Barret Schloerke, Yihui Xie, Jeff Allen, Jonathan McPherson, Alan Dipert and Barbara
Borges (2021). shiny: Web Application Framework for R. R package version 1.6.0.
https://CRAN.R-project.org/package=shiny

Shinydashboard:
Winston Chang and Barbara Borges Ribeiro (2018). shinydashboard: Create Dashboards with 'Shiny'. R package version 0.7.1. https://CRAN.R-project.org/package=shinydashboard

Shinycssloaders:
Andras Sali and Dean Attali (2020). shinycssloaders: Add Loading Animations to a 'shiny' Output While It's Recalculating. R package version 1.0.0. https://CRAN.R-project.org/package=shinycssloaders