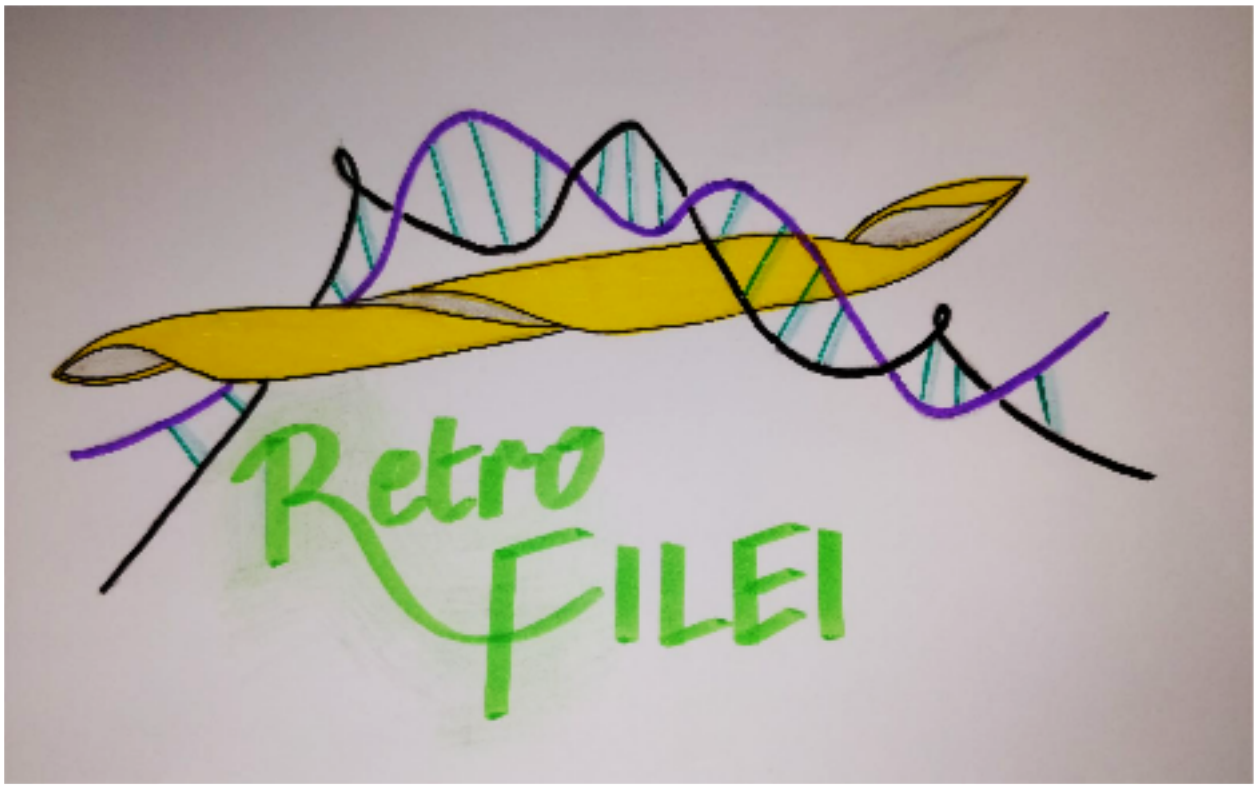
BIO 727P Group Project



Contents

Documentation

The web framework for Retro-Filei was created in Python 3.5 using Django (version 2.0.1), a high-level Python development environment. Django uses a somewhat rigid file structure, the components of which will be explained in this documentation.

Design Philosophy

We had a bottom up approach to the building of the Retro-Filei website with the aim of providing the relevant information in the most user friendly way. We planned for the main functionality that we wanted to achieve and equally assigned roles to team members. We decided to concentrate on instituting working code before considering the aesthetic interface of the website. Once basic functionality was achieved we chose to advance the functionality in the most scientifically appropriate way.

The set up for the website gives the user clear access the the relevant information and easily allows for expansion to other retrotransposon families. We wanted the user to be able to selectively visualise information from our databases in a succinct and accessible way. The ability for users to be able to contribute their own information will allow the website to grow, with visualizations updating in response to new information.

Installing Retro-Filei

An installation guide and the necessary files for replicating the Retro-Filei web framework is included alongside this documentation in the Retro-Filei folder. Due to file size restrictions on GitHub only a sample (25%) of the data we used is available for you to use. The functionality of the website will remain intact, however the visualisations and other information will not provide as accurate a reflection of the distribution of retrotransposons across the human genome due to the small size of the dataset.

Virtual environment

During development the web framework was initiated and written within a virtual environment (using the Python package 'virtualenv'. Using a virtual environment provides a 'clean' python environment divorced from any Python Packages already installed on the machine being used. The virtual environment set up in the folder `/Retro-Filei`. The Django project and apps exist within the `/src` directory within the `/Retro-Filei` directory. A requirements file, `requirements.txt` was also generated here via `pip freeze`, which lists all of the python packages used in the virtual environment (including version numbers).

Django vs Flask

A decision had to be made early on as to the choice of web framework environment we would use. There are costs and benefits to either approach. Flask is a minimalist

environment designed for ease of use and accessibility. The file structure is not predetermined, and any functionality the user wants to implement has to be added in manually. The user must also implement any connection to external databases.

Django on the other hand has lots of inbuilt functionality. This poses an obstacle: there is a steep learning curve; it is not immediately intuitive as to how to use it. The file structure is incredibly strict; functions will not work if they are not in the correct places. This is also a positive aspect of Django - once you've built one Django framework, it is relatively easy to understand someone else's or port yours to another project.

Django directory structure

A Django project (`/group_project)` and a Django App (`/retrotransposons)` were set up in the `/src` directory. This directory also contains the html templates for the websites (in `/templates`), the SQLite3 database used by the website and the script manage.py (the script responsible for configuring the system path, running the local server and providing an interactive shell for interfacing with the SQLite3 database). Two other directories, `/media_files` and `/static_files` serve as places to store documents that are uploaded by users. The directory tree was as follows:

```
src
├── manage.py
├── db.SQLite3
├── group_project
│   ├── __pycache__
│   └── settings
│       └── __pycache__
├── media_files
│   └── documents
├── retrotransposons
│   ├── media
│   │   └── documents
│   ├── migrations
│   │   └── __pycache__
│   ├── __pycache__
│   └── static
│       └── images
├── static_files
└── templates
    ├── retrotransposons
    └── snippets
```

SQLite3 database

By default Django uses the SQLite3 language to produce a database. The option also exists to use either a MySQL or postgreSQL externally hosted database. We chose to use the default SQLite3 simply for ease of use, due to the fact that it would be created locally and would not require any SQL code to work and would not need to remotely accessed. This type of database is not suitable for an actual production environment; only one process is

allowed to make changes to the database at a time. It is however acceptable for use in a development environment such as ours, and the database could be transferred to a MySQL or postgreSQL database at a later date.

The Django project (*group_project*)

Django is structured in terms of projects and apps. A project can contain many different apps, and an app is usable in any project (in a sort of plug and play system). Our project folder, `/group_project`, contains two items of importance; urls.py and `/settings`. This structure can be seen in the directory tree:

```
Retro-Filei
├── __init__.py
├── __pycache__
│   ├── __init__.cpython-35.pyc
│   ├── urls.cpython-35.pyc
│   └── wsgi.cpython-35.pyc
├── settings
│   ├── base.py
│   ├── __init__.py
│   └── __pycache__
├── urls.py
```

Settings (base.py)

The script `base.py` in the `/settings` folder is mostly pre-generated on initiation of the Django project, but a few additions were made. Any downloaded Django apps (such as django-tables2) had to be added to the `INSTALLED_APPS` list, along with our own `retrotransposons` app. This neatly displays the pluggable aspect of Django apps.

The `TEMPLATES` dictionary was altered to have its `'DIRS'` key point to the `/templates` directory in `/src`. Additionally static and media root paths were set up to accommodate stored files for use in the website, such as images and text files.

Urls.py

The script `urls.py` sets up (appropriately) the urls the website will use. A list of functions is imported from the `views.py` script (in the `retrotransposons` app). These functions are used to provide context for urls, which are specified in the `urlpatterns` list. The `urlpatterns` list specifies an extension to look for and the function to be initiated upon executing that url. For example, `127.0.0.1:8000/ORF0_protein_logo` will execute the `ORF0_protein_logo()` function, taking request (generally GET or POST) as its argument and returning an html template with the necessary variables to populate that page. When an html page requires no context, the class method `TemplateView.as_view()` is used to direct straight to these generic html templates.

The *retrotransposons* app

The retrotransposons app folder contains the main functionality of the web framework, housed within a number of different python scripts (admin.py, apps.py, choices.py, filters.py, forms.py, models.py, tables.py, tests.py and views.py). This structure can be seen in the directory tree:

```
Retrotransposons
├── admin.py
├── apps.py
├── choices.py
├── filters.py
├── forms.py
├── __init__.py
├── media
│   └── documents
├── migrations
│   ├── __init__.py
│   └── __pycache_
├── static
├── tables.py
├── tests.py
└── views.py
├── static_files
└── templates
    ├── about.html
    ├── base.html
    ├── contact.html
    ├── home.html
    ├── retrotransposons
    │     ├── alignment.html
    │     ├── HERV_info.html
    │     ├── HERV_phylogeny.html
    │     ├── karyotype.html
    │     ├── LINE1_info.html
    │     ├── LINE1_phylogeny.html
    │     ├── mz_results.html
    │     ├── new_entry.html
    │     ├── ORF_sequences.html
    │     ├── overview.html
    │     ├── phylogeny.html
    │     ├── sequences.html
    │     ├── success.html
    │     ├── table.html
    │     ├── upload_mzID_file.html
    │     └── upload_mzTab_file.html
    └── snippets
        ├── css.html
        └── js.html
```

To understand the context of most of these python files, we must first explore `models.py`.

Models.py

Each object of the class Model corresponds to a table in the SQLite3 database. The fields specified become column names in the table. The model field type (IntegerField, CharField etc) indicate the type of input that the field will accept, with arguments passed to these fields indicating whether for example the field can be blank, has a maximum length or has a default value.

Four models were set up for this project: one to contain LINE-1 retrotransposons, another to contain HERV retrotransposons, one to contain the expression Atlas, and finally a model to enable uploaded documents. Originally it was intended to store all retrotransposon data within a single model. The decision was made to make two separate models for LINE-1 and HERV so that the two families of retrotransposons were more readily accessible. It would have been possible to do it in one model by using a field to differentiate the two, but this may have complicated querysets and functions downstream as the majority of analysis is within each retrotransposon family.

The fields for the LINE-1 and HERV models corresponded to the schema used in UCSC Table Browser, where the data used to populate these models was taken from. We chose to do this so that we would have the ability to access any of this data – whether or not we would actually ended up using it. Fields we deemed unnecessary going forward were marked `null=True` and `blank=True`, so that any future entries to the database could contain information in these fields, but aren't required to.

The fields for the expression Atlas model are admittedly arbitrary and would be subject to change or addition in future; this concept was only briefly worked on due to time constraints. Changing a model once it has been set up is, fortunately, very easy.

Once a model is created (or changed) it must be migrated. Migration essentially commits any changes made in models.py (changes cannot be made to the database by saving models.py).

Changes are evaluated and then committed using the following code:

```
$ python3 manage.py makemigrations
$ python3 manage.py migrate
```

Each migration creates a numbered migration python script in the `/migrations` folder inside the app. It is possible to revert to previous migrations using these scripts.

Once these models had been set up they would next be populated with data. The raw data downloaded from UCSC Table Browser took the form of a .csv file. This data was converted using the python package Pandas into a dataframe, each row of which could be iterated over to extract each column value, which were then used to create entries in the database using the Django model API (these scripts are included in the 'upload_scripts' folder).

Populating the models was done using manage.py shell, an interactive shell where changes can be made:

```
$ python3 manage.py shell
```

After importing the appropriate model, column names and corresponding row data are passed as arguments into (for the HERV model):

```
from retrotransposons.models import HERV
HERV.objects.create()
```

The data stored on these models is utilised by the functions that live in the other python scripts, namely views.py. This data is accessed through the Django queryset API.

QuerySet API

Django provides a convenient interface for interacting with the SQLite3 database, without the need for actual sql code. The queryset API allows for the construction of complex database queries by combining different methods to refine the results. An example queryset on our HERV model could be:

```
HERV.objects.filter(genoName = 'chr2').exclude(GAG = 'None')
```

This queryset would contain all entries in the HERV model on chromosome 3, excluding those that do not have a GAG amino acid sequence. This queryset can be assigned to a variable, which can be further filtered or iterated over:

```
# assign to a variable
qs = HERV.objects.filter(genoName = 'chr2').exclude(GAG = 'None')

# further filter
qs_HERVK-int = qs.filter(repName = 'HERVK-int')

# print each GAG sequence in the query set
for obj in qs:
     print(obj.GAG)
```

Data can also be output as tuples or dicts instead of querysets objects, a feature utilised by many of the functions in views.py.

Views.py

Views.py hosts all the functions of the webframe. Here database information is inputted into functions and the output is used and displayed by the html templates. There are many functions defined in this script, each of which will be given an overview here (comments in the code itself explain functionality).

Each function is as a rule defined twice, once for the LINE-1 model and once for the HERV model. This is admittedly inefficient as most of the code is thus written twice. A future improvement would be to consolidate these duplicated functions together, passing the model used as arguments into the function.

Tables

The functions `simple_HERV_table` and `simple_LINE1_table` are used for configuring the table web pages for their respective models. Using the `simple_HERV_Filter` or `simple_LINE1_Filter` classes imported from `filters.py` an initial queryset (all) is established. This filter is used to configure a table (`simple_LINE1_Table` or `simple_HERV_Table)` imported from `tables.py`. Code required to export the generated tables as .csv files is also set up using the Table_Export() function from django_tables2.

The table, filter and family (identifying the model used) variables are then sent to the `table.html` template. Djangobootstrap3 is used to generate the html for the filter form, while django_tables2 generates the html for the table using a bootstrap template.

A limitation encountered when rendering the table to the webpage was the inability of django_tables2 to wrap text fields. Originally we had included DNA and amino acid sequences in the table, but the long strings of DNA would simply extend outside of the page boundaries. This caused us to limit the number of fields that are displayed on this page and make seperate pages for the DNA and amino acid sequences. Otherwise djang_tables2 is very efficient at constructing very complex tables with almost no effort.

Getting django_filters to interact with django_tables2 also proved to be a challenge, the documentation for both specifying this functionality but remaining somewhat vague as to how to implement it. A solution was dependent upon setting up very specific variables names.

New Entries

Users are able to contribute new data to the LINE-1, HERV and Atlas databases. The Entry_Form() (imported from forms.py) contains fields that correspond to the fields of the appropriate model. If the user has filled in the required fields (those not assigned blank = True and null = True in the model, assigned required = True in the form) the form is deemed valid. The form fields are in and a new entry is added to the appropriate model using the information entered by the user in each field. Upon successfully creating a new entry the user is redirected to the success.html template which informs the user of the unique ID of their new entry.

File Uploads

Users can upload both mzID (mzIdentML) and mzTab files, inspecting them for peptide sequences that match sequences present in our database. A file extension validator is put in place to detect the file type the user has uploaded, and reject files if they do not end in the correct extension (.mzid or .mztab). The functions for parsing these files produce similar outputs (a list of Amino Acid sequences), but use quite different methods for extracting this information.

The function `mzTab_parser()` takes in as its argument an mzTab file. Any lines in that file starting with the string 'PSM' contain peptide spectrum matches, which contain the amino acid sequences which need to be extracted. Each line in this file is split by whitespace to produce a list of words in that line. If 'PSM' is at index 0 of this list, index 1 will contain a peptide sequence (which is then extracted and added to a list). The list of peptide sequences that has duplicates removed using the `set()` function.

The mzTab_parser() function was coded manually as the PDF for the mzTab file format was very clear and more example files were available which made manual coding very simple. The mzTab file format is pretty simple anyway which also allowed it to be easy to code a parser. The MSnbase package in R allows parsing of mzTab files however the output is more complicated to export a list from. Implementation of this package would also be more complicated than just using some simple python code. Manually coding this function allows for far easier integration into the website. Using the MSnbase package in R may have resulted in the lag experienced when running the mzID package via rpy2 as the MSnbase package would also have to be ran via rpy2.

The `mzID_parser()` function uses the R package mzID (see the section on rpy2 later) to produce a dataframe of the information in the uploaded mzID file. This R dataframe is then converted to a pandas dataframe using the pandas2ri module function `pandas2ri.ri2py()`. Decoy sequences are then removed from the dataframe, and each entry in the 'pepseq' column (containing the peptide sequence) is added to a list. Duplicate peptides sequences in this list are then removed using the `set()` function.

The R package mzID was used as it created a dataframe from an mzID file which contained all the relevant information. The package was very simple to use as the documentation was clear. The actual documentation for the mzIdentML file format was rather more complicated and the examples available were massive files which were difficult to work with. The mzID R package was noticeably slower when run via rpy2 compared to within R itself.

A python package, pyteomics.mzid, was another possibility however this proved difficult to extract the relevant information from as the documentation was very vague. It was possible to eventually get it working successfully, however the R package had already been implemented into the website so changes were inconvenient at this stage. Another possibility may have been manually coding a parser, this is more complicated to do than using a package that has already been created, however it would have been far easier to implement into the website. Using the python package or creating a parser would be options for the future in order to speed up the process of parsing mzID files.

Both parser functions output a list of peptide sequences, which are used to generate query sets against the LINE-1 and HERV models. For each peptide sequence a queryset is generated to see if it matches any peptide sequences present in the database. If there are any matches, the queryset is added to a list. This list of matches it sent to the `mz_results.html` template. Each queryset in this list is then iterated over to display the salient fields.

One limitation of the current code is that it does not map a specific peptide sequence to a particular open reading frame (in cases where a match contains more than one open reading frame). It would be possible to do this by running three different query sets for each peptide sequence and building three separate lists, one for each open reading frame.

Another issue affecting both mzTab and mzID file parsers is one of performance. A typical mz file may contain over 3000 sequences. On an average 4 core machine it takes around a second to process 3 or 4 sequences. This equates to wait times of upwards of 15-20 minutes to process a single file. This is obviously an issue, a solution to which is not immediately apparent.

Protein Logos

Protein logos were used to show the variation between sequences within a repeat. Seven functions comprise the protein logo alignment; six are the same except for the open reading frame they use (ORF0, ORF1, ORF2, GAG, POL and ENV), while the other actually generates the alignment images for use in the `alignment.html` template. In hindsight it would have been better to implement a form that would input an argument into a single function to determine which open reading frame to use.

The function `ORF0_protein_logo()` (and respective copies) each take in a repeat name supplied by a form (`ORF0_repname_Form()` etc, imported from `forms.py`). This repeat name is used to generate a queryset against the appropriate model, returning a list of tuples comprised of (repeat name, genomic start position, genomic end position, strand and open reading frame sequence) from objects that contain the desired open reading frame sequence. This list is used to write a fasta file containing this information. The file path to this fasta file is then passed as an argument for the `protein_logo()` function.

Via rpy2, the protein_logo() function calls the R package Biostrings' function `readAAStringSet()` to read in the sequences in the fasta file. An alignment is performed on these sequences using the R package `msa`. These packages use the S4 class data format, which pandasr2i is unable to convert to a pandas dataframe, so the rpy2 indexing system is used to make a python list of these sequences. This list of sequences is written to a text file (that the user can download).

In order to fit the protein logo on the webpage, it is necessary to split it into smaller sequences (fragments). The python class TextWrapper(), imported from the textwrap module, breaks each sequence into chunks of a specified length. These fragments are then zipped into lists containing all fragments from the same position in the original complete sequence (essentially creating multiple mini alignments). The height for the .jpg image output is assigned relative to the numbers of items in this list of zipped fragments (otherwise it will be squashed and unreadable). The R package ggseqlogo finally generates the protein logo image that is sent to to the `alignment.html` template.

Originally, MSA was considered as a package that would easily be able to complete alignments, protein logos and dendrograms. We soon realised that the requirements of MSA for generating a protein logo mean that it overcomplicated the implementation within the website. The package ggseqlogo shortcuts and simplifies the ggplot package's ability to produce protein logos with more versatility of MSA, and is able to take take the MSA aligned sequences as input.

A bug was encountered with ggseqlogo that only seems to occur while using it within views.py: Any extra graphical options implemented on the ggseqlogo image (such as ggplot2 theme options, labels etc) would not render. This behaviour is unexpected since when the function as a whole is executed as its own python script outside of the Django webframe, all of these extra options are rendered correctly. This behaviour is unfortunate as we would

have liked to have removed the numbers beneath each line of the protein logo image (ggseqlogo treats each line as a new alignment and thus starts indexing the positions again from 1 on each line).

Dendrograms

Two types of dendrograms were generated, cladograms and phylogenetic trees. Cladograms show how related the different repeats are to each other, but do not show the genetic distance between them. The branch distance in a phylogenetic tree is proportional to the evolutionary distance between the repeats.

The msa package allows dendrogram creation as well as multiple sequence alignment, the seqinr package is required in order to calculate a matrix of pairwise distance from aligned sequences, and the ape package is required to perform the neighbour joining tree estimation which can then be plotted as a tree. Actually plotting of the tree is done by the plot() function of R, this is a lot more difficult to customize as it is not specifically for dendrograms. A basic tree was generated using msa, however other packages were explored to improve the visuals.

The cladograms and phylogenetic trees were created using the ggtree package from bioconductor which is an extension of the ggplot2 package in R. The ape package is also required for ggtree. The ggtree package required a Newick format file or a nexus file as input to generate a tree. For each ORF a FASTA file was created that contained the consensus sequence for each repeat. This FASTA file was used for a multiple sequence alignment with Clustal Omega, from which a neighbour-joining tree and a Newick format file were generated. This Newick format file was imported into ggtree in R to create a cladogram and phylogenetic tree. The ggtree package has a lot of options for customization of the tree which proved to be very useful in creating visually appealing cladograms and phylogenetic trees with more relevant genetic distance information.

The cladograms and phylogenetic trees were implemented on the website as images as this is a lot quicker to load for the users. The dendrograms do not change as long as the database remains the same, so it is not beneficial to make them into a function as the output will always remain the same. Generating Newick format files is not too difficult using the ape package however implementation into the website would become more complicated as files would need to be temporarily stored. The process would involve running a multiple sequence alignment for each repeat for each ORF in the database to generate a consensus sequence for each ORF. These consensus sequences from the multiple sequence alignment would need to be saved as a FASTA file. A multiple sequence alignment will need to be done to the FASTA file of consensus sequences in order to create a Newick format file which can then finally be used to generate a dendrogram with ggtree. Limiting the number of functions used from different packages is also beneficial for implementation into the website.

It is also possible to generate these dendrograms using code. This is more complex and requires more packages. Firstly the FASTA file would be used for a multiple sequence alignment using the msa package in R. A newick format file could then be extracted from the multiple sequence alignment using the ape package. This file could then be imported into ggtree to create the cladogram and phylogenetic trees. This is rather long winded and requires far more processing power than just loading images on a website. If more

retrotransposon families are added in the future it will be necessary to code for dendrogram generation as manually creating pictures will take too long.

Karyotypes

The `karyotype()` function produces an image of the distribution of retrotransposons throughout the genome as points on each chromosome. The argument is supplied to the `karyotype()` function by the `LINE1_karyotype()` function or `HERV_karyotype()` function. A repeat name chosen by the user from a dropdown list is used to generate (via a queryset) a pandas dataframe containing the columns: repeat name, genomic start position, genomic end position, strand and chromosome.

In the `karyotype()` function Pandas is used to separate this dataframe down into individual dataframes for each chromosome. Using pandas2ri, these dataframes are converted into R dataframes and stored in a list. The R package karyoploteR is used to generate an image of each human chromosome.

The list of R dataframes is then iterated over, each dataframe is used as a source to generate the start and end positions of the repNames for karyoploteR via another R package; GenomicRanges. This package consists of a function called `makeGRangesFromDataFrame()` where a dataframe object is implemented and for each chromosome, the start and end position is given. KaryoploteR has another function called `kpPlotRegions()` this function produces rectangles across the genome based on the what is given in the GRanges object, i.e this actually shows the plot of each repName and its location on the genome. Each rectangle is equal, though some may appear longer than others, this is due to overlapping repeats. This function was used as opposed to the functions `kpPlotDensity()` and `kpPlotCoverage()`, because it showed the best representation of the data. The generated image is stored and used in the `karyotype.html` template.

There were many ways of visualising the data such as bar charts (see below), scatter diagrams as well as ideograms. The ideogram/Karyoplot was most preferable because it directly maps the location of the repNames to the chromosomes.

A variety of R packages could be used to visualise the data. Initially ggbio was the most favourable package to use when plotting the karyotype graph. After some research, it was found to have only worked with the ‘hg19’ human genome is incompatible with the database because it came from updated hg38 genome. When plotting the karyotype with ggbio only one chromosome could be plotted at a time this could prove to be very time consuming.

KaryoploteR was a good package to show the distribution as it overcome the limitations of ggbio. One limitation that was found was plotting large quantities of data, all the repNames could not be plotted at the same time otherwise the plot would be a lot more clustered. This is why bar graphs were introduced (see below).

Sequences

The DNA sequence functions, `LINE1_sequences()` and `HERV_sequences()` are quite simple functions. They take in a repeat name (from a list of choices imported from choices.py) or ID number from the user via the `sequences_form()`. This input is used to generate a queryset matching that repname/ID.

This queryset is iterated over to compose a fasta file of the sequences, with the start and end position, the strand, repeat name in the description line. This same queryset is then iterated over within the html template `sequences.html` to populate the page with the same information included in the fasta file (which the user is also to download).

The peptide sequence functions (ORF0_sequences etc) work in exactly the same way, returning the Amino Acid sequences for each open reading frame as their output.

Information graphics (barcharts and stack plots)

To summarise the given data in the database, two types of bar charts were generated using the Matplotlib library in python3. This library is widely used in graphically visualising data in python. Matplotlib contains a package which is referred to as pyplot and it is used to generate and customise figures..

The barcharts were formed using a Django Queryset API output of a list containing tuples of the repeat name and its chromosomal location for all repeats. One bar chart visualises the frequency of the repeats per chromosome and the other, the frequency of the repeat names. The Django Queryset API input is converted into dictionaries of frequency (using the x axis component as the keys) and plotted using the function plt.bar(). To generate both the graphs the same procedure was followed but using the appropriate dictionary. Figures had to be individually adjusted to present the data more clearly.

The stacked bar plots were generated using three Django Queryset API outputs of a list containing tuples of the repeat name and its chromosomal location only for repeats containing a particular ORF. A frequency dictionary was created for each ORF list and these were combined into a 'super dictionary'. The super dictionary was then implemented into the function plt.bar(). The size of the figures were adjusted so that they could be seen clearly on the page.

Matplotlib was used for these graphs instead of R because it was more feasible in applying the code to Django. This was noticed while doing the Karyotype plot graph. For the karyotype to be implemented into Django, it was not as straightforward as a lot more packages would have needed to be used. Using R would have required the conversion of R script to python.

Interfacing R with Python – rpy2

Several of the functions described previously make use of R packages (using the R language). Since everything in Django is written in Python, it was necessary to provide a bridge between these two languages. The python package rpy2 was used as that bridge.

Rpy2 converts R functions into python objects, with methods that emulate R functionality. R packages were installed and converted to python objects using the following code:

```
# to install the R package ggseqlogo:
From rpy2.robjects.packages import importr

# import the R base functions:
base = importr('base')
```

```
# set the source repository to download from:
base.source("http://www.bioconductor.org/biocLite.R"

# import the BiocInstaller R package
Biocinstaller = importr['BiocInstaller')

# to install the R package Biostrings:
biocinstaller.biocLite(Biostrings)

# once installed, the R package can be imported into a script:
Biostrings= importr(Biostrings)
```

To call an R function from a particular package once it has been imported it is simply a case of first calling the package that function comes from, and using the desired R function as a method on that package. For example, following from above:

```
# to use the readAAStringSet() function from Biostrings:
example = Biostrings.readAAStringSet()
```

The output of R functions from rpy2 retains the data structures they would have if they were running in R itself (vectors, dataframes etc). Accessing columns, rows or index positions in these data structures can be done either through python style indexing (for vectors only) or by using the .rx() and rx2() methods (equivalent to R's [ ] and [[ ]] notation). To accommodate differences in the way the two languages work, all '.' characters in R code must be changed to '_' characters in rpy2 code. Images of the graphical output of functions are generated in the same way as R using the `rpy2.robjects.r.jpeg()` and `rpy2.robjects.r('dev.off()')` functions.

The package pandas2ri provides a convenient method for transforming pandas dataframes to R dataframes and vice versa. This greatly aids the simplicity of interacting with dataframes while using rpy2.

Under the assumption that the use of rpy2 would be a simple process for implementing R within python the use of R packages was preferred. The variety of R packages specifically designed for similar biological uses was a major factor in the decision to use R for most of the plots. Python packages also exist for similar uses, however most of these were more complicated to work with. Previous experience with plotting in R also allowed quicker and easier understanding of the new packages used for plotting. Upon the implementation of rpy2, it became clear that it was more complicated and time consuming than expected, however the majority of plots had already been created using R code and packages. Had this been identified sooner a greater emphasis would have been placed on choosing python packages in order to simplify the process of incorporating functions into Django.

Non standard python scripts

The following python scripts are not automatically generated upon creation of a Django app; they have been created to house specific class types separate from views.py, mainly to keep the code organised, easier to read and accessible to other scripts.

Tables.py

Tables.py contains all of the definitions of the Table class. This is a non standard class, provided by the app django_tables2. Each definition of the class Table contains a number of statements that correspond to fields in a model in models.py. These statements are also the names given to the columns in the table (unless the verbose_name argument is given).

The class meta specifies the model that the table will use, the html template used to render the table (a few are provided by djangi_tables2), and the fields from the model to use (which will be automatically generated if not overwritten by the statements with the same name).

Instances of the Table class are passed to an html template via {% render_table table %}, which uses the template specified to generate html code for the table. This lets the page generate a dynamic table from a large query set without having to create the vast amounts of html code that would need to be needed if it were done manually.

Forms.py

Forms.py contains all of the definitions of the Form class. This class is one of the default django classes provided. Each definition of the form class contains one or more statements that correspond to fields that the user will provide input for. Field types include CharField (for text), IntegerField (for numbers), ChoiceField (where the user chooses an option from a dropdown menu) and FileField (for uploading files).

Each field can take the argument `required = True` or `required = False`; a required field must be filled in for the form to be valid.The widgets the form will use in each field can also be specified, allowing some customisation.

The model the form will refer to is specified by the class meta, along with the field names. Form instances are passed to an html template via {% form %}. Django_bootstrap3 can be used in combination with forms to render the form with bootstrap CSS, using {% bootstrap_form form %}.

Functions in views.py can take POST requests from forms. If the form is valid, the information from each form field is gleaned using the form ‘cleaned_data’ attribute. Each form field is accessible via its name. For example:

```
Form = ORF0_repname_form()

Repeat_name = form.cleaned_data[‘repname’]
```

It is possible to overwrite the default cleaned_data attribute, perhaps to arrange a validation error if two fields have the same value, or to set up mutually exclusive fields.

Filters.py

Filters.py contains all of the definitions of the FilterSet class. This is a non standard class, provided by the app django_filters. Syntax wise, FilterSet definitions are almost identical to the Form class (they are even rendered in html in exactly the same way). The difference lies in the field statements, which use django_filters.NumberFilter (or ChoiceFilter, CharFilter etc). These fields take similar arguments to form fields; in addition to these fields can take the 'lookup_expr' argument. This argument ties in to what FilterSets actually do.

Unlike forms, which can have various uses, filters have one purpose: to modify a queryset. Each field in a FilterSet instance modifies a queryset from a starting queryset supplied when the object is created. For example:

```
f = simple_HERV_Filter(queryset=HERV.objects.all())
```

The class definition simple_HERV_Filter has a repeat name (repName) ChoiceFilter. The user can choose a repeat name, say 'HERVH-int', for this field. The queryset is them modified:

```
queryset = HERV.objects.filter(repName = 'HERVH-int')
```

Each filter field the user fills in then further modifies the queryset, allowing the user to dynamically change the data that is returned to them. The main reason why we chose to use django_filters is its interaction with django_tables2. By combining a the Table class with the FilterSet class it is possible to update the information returned by a table quickly and easily by just changing the entries in the filter form fields.

Choices.py

Choices.py is a collection of lists and querysets that are used by various functions in views.py. Putting them here is mainly to aid the readability of the code and make them accessible to different scripts. The script contains the function `choice_generator()`. This function creates lists for use in ChoiceFields(). Choice fields take their options as tuples; the first value being the label added to the dropdown list the user picks from, the second value is the one that the choice field returns via the cleaned_data attribute. The choice generator function simply takes each item in a list (returned from a values_list queryset), duplicates it, and places the pair of values into a tuple, which is then appended to a list. A 'Select All' option is also included in these choice lists, which allows a user to choose all the options available. These lists are additionally alphabetically sorted to make finding individual repeat names in the drop-down list easier.

These choice lists are newly generated every time the website is initiated. This allows any new data added to be included in these choice lists. During development we used pre-generated lists to increase the performance of the website.

Site Map

The Retro-Filei website contains the following pages:

(Base.html)

- Home
- Overview
- Contact
- About
- Atlas
    - Table
    - Add to Database
- LINE-1 / HERV
    - Information
    - Phylogeny
    - Table
    - DNA sequences
    - Open Reading Frames
    - Distribution
    - Alignment
    - Find Peptide Sequence
    - Inspect mzTAB
    - Inspect mzID
    - Add to Database

Django template tags

Django includes a method for applying logic to html templates, using syntax similar to python. Template tags can take variables returned from python functions (usually from views.py) using double curly braces notation; {{ variable }}. These variable tags can be iterated over, and have boolean logic applied to them to add or remove html code from the template. The {% for %} {% endfor %} template tag can iterate over iterable objects (lists, querysets etc) and generate html code for each item in the iterable. The {% if %} {% endif %} tags accept logical arguments to conditionally include or exclude html code situated between the tags.

Other template tags can used to automatically generate html code. The {% url %} template will generate a html tag. The {% static %} tag pulls files from the framework's static root folder. Bootstrap CSS is added to pages using the {% load bootstrap3 %} and {% bootstrap_css %}. The list of available template tags is extensive, the complete list is detailed in the Django documentation (https://docs.djangoproject.com/en/1.7/topics/templates/)

Base.html

This template codes for the navigation bar and website header text that is used on every page. Other pages uses the block content tags {% block content %} and {% endblock %} and {% extends base.html %} to include the html code from this template on those pages.

Home (home.html)

A simple html page that welcomes the user with a few buttons directing the user to the different aspects of the website.

Overview (overview.html)

This page provides general scientific information about retrotransposons.

Contact (contact.html)

Lists the email for each group member and a link the the group's GitHub repository.

About (about.html)

Lists links to the python and R packages used to produce the website and a link to UCSC Table Browser where the data used to populate the database was taken from.

LINE-1 / HERV:

Information (LINE1_info.html / HERV_info.html)

Each information page displays chart graphics produced by the stacked_plot_chr(), stacked_plot_repname(), bar_plot_chr() and bar_plot_repname() functions, along with text information about each retrotransposon family.

Phylogeny (LINE1_phylogeny.html / HERV_phylogeny.html)

Displays the phylogenetic trees and cladograms for each retrotransposon family. These images were pre generated, the page simply uses a static image file.

Table (table.html)

Takes context from the functions simple_HERV_table(), simple_LINE1_table or Atlas_table(). Each function specifies the table to render in the template tag {% render_table table 'django_tables2/bootstrap.html' %}, the filter variable for {% bootstrap_form filter.form %} and the {{ family }} variable to customise the page dependent on the information being displayed.

The filter form fields are contained within a drop-down well so that so that they can be revealed or hidden to prevent to the page being too busy. The rendered table is built from a bootstrap template, and can be sorted by clicking on the column names.

DNA sequences (sequences.html)

Takes context from the LINE1_sequences() and HERV_sequences() functions. A form rendered by {% bootstrap_form form %} allows the user to pick a repeat name or ID number to pull from the database all entries that match that input. Fields from each item in the queryset are extracted using a {% for %} template tag. Each item (result) is contained within a 'panel' division to make it easier to separate them.

The user can also download these sequences as a fasta file - the download button uses the {{ repname }} variable to add the repname chosen to the file name.

Open Reading Frames (ORF_sequences.html)

This page works in exactly the same way as sequences.html, but it returns the open reading frame fields from the objects in the {{ info }} queryset provided by the ORF0/ORF1/ORF2/GAG/POL/ENV_sequences() function. The page if differentiated logically between HERV and LINE1 using the {{ family }} variable, which determines which fields are chosen from the queryset.

Again the user can choose to download the results as a fasta file in the same way the the DNA sequences.

Distribution (karyotype.html)

On this page the user is able to choose a repeat name from a dropdown list, which is sent to the karyotype() function to produce to return an image displaying the positions of the repeats on the chromosomes of the human genome. The logical variable {{ image }} is turned to true once the karyotype() function has running, informing the page via an {% if %} template tag to pull the image from the static directory (using a {% static %} template tag).

Alignment (alignment.html)

Similar to the karyotype page, this page allows the user to choose an open reading frame from either LINE-1 or HERV (determined by the {{ family }} variable) and pick a repeat name from a dropdown list of those that contain that contain that open reading frame. Again the {{ image }} tag is turned to True once the protein_logo() function has created the image; it is fetched from the static directory and displayed on the page.

Find Peptide sequences (AA_search.html)

The user is able to enter an amino acid sequence into the form field, which sends the input to the LINE1_AA_search() or HERV_AA_search() functions. A queryset it returned to AA_search.html containing all the repeats with peptide sequences containing that input sequence. The number of matches to sent to the {{ matches }} template tag and the queryset to the {{ info }} tag, which is then iterated over to return the pertinent fields.

Inspect mzTAB (upload_mzTAB_file.html, mz_results.html)

The page contains a file upload form where the user can upload an mzTAB file to see if it contains any peptide sequences that match sequences in each database. A file validator is implemented to prevent the wrong type of file being uploaded. The user is redirected to the mz_results.html template once the file has been successfully uploaded and analysed using the mztab_parser() function.

The {{ hits }} variable contains a list of any sequences in the uploaded file that have been successfully allocated one or more database matches. These are displayed as a bullet point list.

The {{ output }} variable is a zipped list containing querysets and the peptide sequence from the file that matched to them. Using a nested {% for %} {% endfor %} each queryset that matched that peptide sequence is iterated over to extract the useful fields.

As mentioned previously, the wait time on this page is unfortunately extremely long (upwards of 20+ minutes). This is rather obnoxious for the end user; sometimes the web browser assumes the page may have crashed due to the extreme load times. It would have been nice to have some sort of loading bar to show the progress of the function.

Inspect mzID (upload_mzID_file.html, mz_results.html)

This page functions in exactly the same way as the Inspect mzTAB page, except the file validator looks for the .mzID extension rather than mzTAB. The uploaded file is sent to the mzID_parser() function. Wait times are even longer on this page because the mzID_parser() function takes more time to read the document. The user is again redirected to the mz_results.html template, which is given the same context variable as before.

In hindsight a dropdown choice menu to pick the file type to upload would have been preferable to making two separate pages for the file types.

Add to Database (new_entry.html, success.html)

On this page the user is presented with form for adding a new entry to the appropriate database. A number of the fields are marked ‘required’; the user cannot click submit until these fields are filled in. These required fields are the ones with the most useful information, and are the ones returned or used by the website’s functions. The page will flag unfilled required fields if the user attempts to click submit.

The user may leave the open reading frame fields blank - the user may know they have a retrotransposon but may not know whether it contains any open reading frame. These fields are replaced with ‘None’ in the database if they are left blank.

Upon successfully submitting the form, the new entry is created and the user redirected to the success.html template which displays the ID number of their entry.

Unfortunately there is no way to ‘police’ this feature at the moment; that is to say that any user can upload new entries with completely nonsense information, and they can do it as many times as they want! Since the website builds choice lists dynamically and feeds these lists to the websites functions, weird things may start happening. Due to the nature of the information required it would be quite difficult to prevent this sort of thing happening, outside of having some sort of moderator to review entries.

The option for entering multiple new entries or reading in bulk new entries does not exist. This could be implemented if the user provided a csv file with the appropriate column headings; code was already written to do this for populating the database in this way.

Future Development

Although we were able to incorporate a lot of the desired functionality into the website framework, there are many aspects that we recognise as needing improvement. Within the framework itself there is much unnecessary repetition of functions and this requires streamlining to prevent this.

Alongside this, our website has a very basic user interface and implementing more complex css and improving general aesthetics would be a good idea for future development. This would make navigating the website a more pleasant experience for the user. As well as having a basic user interface we would like to have been able to make plots more interactive to increase the scientific data that can be shown to the user.

Implementing the pyteomics module for python for mzID parsing would be another improvement as the current mzID package in R parses very slowly when run via rpy2. The pyteomics module would not need to be run via rpy2 as it is a python module which should improve the speed of mzID parsing. Another option would be to manually write an mzID parser in python which should also run faster.

Another future improvement would be expanding the atlas to take out the matches from uploaded mzID and mzTab files and putting this information into our atlas database. This will be a very useful tool for users as it will show matches found in real samples and will be important for linking retrotransposon expression to cancer and disease.

As a test website our website only shows functionality for two retrotransposon families. We would like it to be possible for the website to be able to incorporate other retrotransposon families. Currently the database hosted within the framework has separate databases for each of the families, ideally this would be a single database that could be built upon or to directly connect to the external UCSC public access MySQL database.

We believe we have a strong foundation from which to build these future developments on. Most of these are relatively simple to implement and we believe these should be easy to incorporate given more time. We are proud of what we have been able to achieve within the given time frame.