Reflective essay on the biocomputing 2 group project – Ang Li

Approach to the project

We took a standard initial approach to the project - reviewing the specification document together at the initial meeting, and allocated roles according to individual preferences of areas of development. We discussed the likely achievable objectives, with a plan to review progress. Following the feedback session, we met again to discuss the feedback from the lecturers, which highlighted that our dummy API was under-developed and of insufficient focus on the details.

I have subsequent had a meeting each with the other members of the team individually, in which we fleshed out what information would be passed between the layers. During development, we discussed changes and clarification via email.

From our discussion and my own review of the notes and advice for the business layer, I surmised that my role was to perform minimal processing of the database layer contents, except for where calculation steps were involved to generate new data for the front end.

Performance of the development cycle

My contribution to the development cycle was for facilitation of testing of the interaction between the layers, I felt that the example of getAllEntries() as a potential check within the dummy API for a simple passthrough to the DB layer did not require modification. However, I also defined a simple return string for db_input(), from output from the business layer could be queried.

The development process

My personal coding process was a short cycle of intensive iterations of code over the course of a week prior to submission. Prior to this, I had reviewed the genbank file and documentation, and was thus aware of what the likely outputs were from the database layer. Once I had started coding, I realised that my dummy API approach to taking in and returning data through 2 different functions was redundant, without providing additional benefit. I also categorised functions into those that needed to be called by the front end, and those for business layer use only. The latter were spun out into the various modules. However, since we did not reach a conclusion on the database outputs, the final "live" business layer contains assumptions as to the nature of information coming from the database layer. (Which includes the lack of additional processing for certain key fields.)

Code testing

During my development cycle, I conducted testing of my code using sample sequences copied from the genbank file, or contrived test sequences to determine whether a particular aspect of the function (i.e. determining that the restriction enzyme function would detect a cut site within the exon start/stop sites.). This was carried out locally using the Spyder IDE using a WinPython installation. This was an adequate substitute testing that the processing steps were functional and performed largely as expected. However, there was no substitute for an example sequence to test the interaction between the database layer and business layer. There is also uncertainty as to the scalability of the code, and whether attemping the higher throughput operations will cause issues with speed. There is also the possibility that

exceptions such as empty values in critical fields, end up causing termination of code, or cause errors within the output.

Known issues

Whilst the ambition was to allows personalisation of the restriction enzyme input into custom sequences. Time restraints meant that the business layer only has a limited and prespecified number of restriction enzymes. The implementation reflect this, and there is no flexibility to choose 2 of the 3 enzymes, rather than "all".

From working with example sequences from the genbank file, it has become apparent that there are sequences that causes errors due to incorrect alignment/incorrect translation/sequence information. This makes the information on chromosomal codon use and coding sequence codon use difficult to interpret. From examining the sequences directly and running code targeted to review whether the exon identification process was error free. I have not identified any clear point of error in the calc_exons.py script that may cause such a mismatch error. It remains possible, that this error may exist within the genbank file itself. I have attempted to mitigate this by only returning partial alignments where there is likely to be mismatch. An area for improvement would be to include information and error text to pass to the front end, so that the uncertainties are explained to the user when this occurs.

For the business layer codon_useage, there is a weakness in the code, which assumes that there is perfect alignment, which may influence the accuracy of the function. One possible solution to this would be to include sequences where there is full alignment only, through preprocessing/error checking within the protein alignment module.

Alternative strategies

An alternative strategy that we discussed at the initial meeting but agreed not to implement, was the use of the database layer as a cache. There was potential to generate the codon useage and coding sequence information at website setup, with a strategy of using an additional function to calculate this for new additions to the database.

This approach may have helped to identify sequences which do not align correctly between the coding DNA and translation entries. Additionally, from browsing the genbank file, some entries referenced the entirely of the chromosome sequence from the experiment. Although we did not performance test the running of the business layer on such an entry, there is undoubtedly a performance benefit from caching exon/coding sequences from these entries in particular.

A further potential alternative to simply overwriting the text file output of runAllcodon_use.py, would be to incorporate an aspect to the script to automatically back up the existing file before the rest of the code is run. Therefore, if any error or interruption occurs during the running of this code, the previously cached data is not lost.

What worked and what didn't : problems and solutions and personal insights

Overall, I feel that we failed to collaborate effectively in this project. Although it is important to emphasise that was due to any one individual action or mistake. On reflection, one of the issues that we encountered was that we did not identify a clear project leader from our initial meetings. This, alongside real-life events has likely lead to letting the assignment drift and staying within our own silos, leading to last minute coding and delivery of a rushed end

product. As such, we did not follow the development cycle closely, although the various pieces to do so were present. Accordingly, this was a missed opportunity to support each other during the project. Personally, I feel that I have not been effective in considering the needs of others when choosing to rapidly develop my code towards the end of the assignment period.

Whilst I feel that there is a degree of success in the functional aspecta of the business layer - as shown from the test data in bl_testing_api.py. There are many areas which I feel could be significantly improved. For example, when designing the output to the front-end, the degree of complexity that exists when putting everything into 1 dictionary has left other team members with the unenviable task to detangling the correct pieces of information to fulfil their requirement. This could have been improved by returning several different iterable objects separately, which helps with human review of the output. Additionally, when reviewing the written code, it is clear to me that whilst there are some error codes, the implementation could be improved by use of error flags, to help identify the source of the error and debugging steps.

Completing this project has given me some confidence in my ability to solve problems using python. This project has shown me that whilst I have skills of implementing scripts and functions that are operable, the code that I write is not pythonic in writing "beautiful" code. Although I have implemented simple regex commands, I feel that some aspects (such as converting a DNA forward strand to the complementary strand), could have been improved with the use of regex rather than iterating through a string. Although I attempted this approach – this was ultimately unsuccessful due to difficulties in transforming more than one character (i.e. all A to Ts AND all T to A) in a single step. This is clearly an area of development for myself to focus on going forwards. Another opportunity for my personal development would be to greater familiarise myself with list comprehensions, which may simplify some of the longer scripts that I have written.

I have also learnt important lessions regarding design of the API. Whilst I have approached the design of the business layer from a biological perspective – firstly completing coding for exon identification, before coding for codon useage and finally protein alignment. This approach may have been too naïve, assuming that all entries would not contain errors that would misalign exons and amino acids. If I were to repeat this project, I would incorporate the amino acid sequence into the initial step as an error checker. Perhaps giving options to divide outputs into fully aligned, partially aligned and non-aligned, so that end users may be more aware for the potential for errors and biases in partially aligned and non-aligned entries.

Code documentation

The business layer code is structed in the format of the bl_api.py file containing most of the functions required. There are separate modules that the business layer API subsequently calls as needed, which are split into individual scripts.

The list of required scripts are: calc_exons.py, codon_useage.py, prot_gene_alignment.py, rest_enzyme_activity.py, runallcodon_use.py. These should be present in the same subfolder as bl_api.py.

There are 5 points of interaction between the frontend and the business layer. These are (in order of intention of running:

- runAllcodon_use.runAllcodon_use()
- bl_api.rest_enzyme_list()
- bl_api.getallentries()
- bl_api.frontend_input()
- bl_api.getAllcodon_use()

It is anticipated that at first setup, the front end interface should execute the runAllcodon_use. runAllcodon_use(). This will query the database layer and produce a text file document the codon use within the database. The output file is produced as a text file in the format:

> Codon, Amino acid, raw count, percentage use – with example: ATT I 0 0.0
>
> Each field is separated by a space: " " and each codon is separated by new line "\n"

This operation is expected to take a significant period of time to complete. When the database is updated, this operation should be repeated, with the script automatically overwrite the existing file.

Once this text file is present, api.getAllcodon_use() can be used to fetch the file. Whilst a direct link to the file is possible, maintaining backup files before overwrite may be desirable, and this function can be updated to direct to either the live version, or latest backup copy as appropriate

bl_api.rest_enzyme_list()

This function returns a list of available restriction enzymes for which the calculation of activity sites is possible. This function returns the names as strings within a list, including an option for "all".

bl_api.getallentries()

This function directly invokes the database API function to obtain a list of the all the entries within t

This returns an iterable object with the format *(From db_API)*: [('accession', 'gene', 'protein_id', 'sequence', 'aa_seq', 'chromosomal location')]

bl_api.frontend_input()

This function takes input from the frontend as a 3 or 4 item iterable object (tuple or list), using the information to perform a search and carry out necessary processing to present data back to the front end.

The input should be of format: (searchfield, data_type, rest_enzyme_flag, rest_enzyme_name)

- searchfield - string object
- data_type - string object of options "gene_id", "gen_acc", "prot_prod" or "chro_loc" - corresponding to the type of search data
- rest_enzyme_flag - boolean - indicating whether restriction enzyme search is required.
- rest_enzyme_name - string object (Optional)

The function returns a dictionary of {numerical index:tuple}. The specifications for the returned dictionary is as follows:

Structure for key:value pair:

{index:([Gene name, Accession, Protein product name, Chromosomal location],[raw DNA sequence, Exon locations], alignment, codon_data, renzyme_output)}

index	Int - 0->x (number of entries returned, this is also the dictionary key)
[Gene name , Accession, Protein product name, Chromosomal location]	4 item list of string objects
[raw DNA sequence, Exon locations]	2 item list, of string object and list of tuples
alignment	list of tuples with of (Amino acid, codon) pairing, with both being strings. An error message is returned if alignment fails
codon_data	dictionary of codon useage in format {codon:(number_of_occurances, amino_acid_letter, codon_use_perc)} • codon - str - three letter codon • number_of_occurances - int • amino_acid_letter - str - single letter amino acid code • codon_use_perc - float - percentage use of the codon within the sequence
renzyme_output	Returns a tuple of 2 dictionaries: renzyme_output[0] - {enzyme_name :[(start, stop), midcut]} • enzyme_name - str • (start,stop) - int (if activity in overall sequence), formatted in format of the range function (i.e. the true end site is stop-1) o If no activity found for the restriction enzyme, the module returns the tuple ("No","activity") o midcut - int (0 - no mid exon cut, 1 - mid exon cut, 2 - no

	activity on DNA strand) - specific to the (start,stop) site renzyme_output[1] - {enzyme_name:overall_midcut} o overall_midcut – int o if value == 0 - activity present on DNA strand but suitable to isolate exons, if value ==1 :enzyme not suitable to isolate exons)

For information - of the additional modules:

calc_exons.py

Is used to calculate

codon_useage.py

Is used to generate a dictionary of codon useage of a given DNA sequence

prot_gene_alignment.py

Is used to generate an alignment from a protein sequence and a genomic sequence.

rest_enzyme_activity.py

Is used by the business layer to calculate restriction enzyme activity.