1. Approach to the project

1.1) Interaction with the team

Overall it was a very positive experience working with Rudo and Sungjae. We met at least once a week to discuss progress, exchange ideas and help each other on persistent issues. Early on in the project we set up a GitHub repository and a Slack group for efficient communication and sharing of code snippets.

1.2) Overall project requirements

We read through the project requirements (http://www.bioinf.org.uk/teaching/bbk/biocomp2/project/page01.html and http://www.bioinf.org.uk/teaching/bbk/biocomp2/project/page02.html) as a group to make sure we were all on the same page with regards to the meaning of the key terms such as codon usage and restriction site.

It was clear that we needed the following functionality from the website:

- a summary table of all records in the database, to be displayed on the home page,
- the ability for users to search the database, based on some key descriptors,
- a 'gene details' page that could be accessed from the summary table and through a search, and would include information such as full dna sequence, codon usage and viable restriction sites (if requested by the user)

The website structure and hard-coded html was developed by Rudo quite early on, which was good because it gave us some direction in terms of trying to get the relevant information to various parts of the website.

1.3) Requirements for my contribution

My task could be clearly split into 3 parts:

- a parser (taking genbank text input, and returning the data of interest in a format that can be loaded into the database)
- designing and setting up a database to store the parsed information
- providing a data access tier that allowed be imported to the middle layer

A bit of planning and a top-down approach was required here. I let the database structure and access be determined by the middle-layer and website requirements. The summary table needed on the website clearly translated to an identical table in the database, which could be accessed with a simple 'select *' sql query. I discussed with Sungjae the format of the data that needed to be supplied to the middle layer, so that I could design the other database tables and data access functions accordingly.

When designing the database and the parsing script, I went with the assumption that there would be no new records added to the database (i.e. it would remain relatively small, with only 3-400 entries). This meant I could have a fairly denormalised database, which is best for fast queries and when data is bulk loaded rather than continuously added. Also, the relatively small number of records meant that storage space was not an issue.

Parsing and loading the data into the database is a one-time operation, so I realised the emphasis should be on faithful transimission of the data (maintaining integrity across records and accuracy

within records), rather than speed. It wouldn't make a difference if setting up and populating the database took 0.1 seconds or 10 seconds.

2. Performance of the development cycle

I felt the development cycle worked well within the group, with each member of the group giving regular updates on their situation. I think I could have starting development more quickly, as this would have allowed me to test the database and get feedback on the data access performance. This would have also helped Rudo and Songjae test their layers with actual data.

Unfortunately, due to unforeseen circumstances our website developer, Rudo, was unavailable for the last week of the project. This meant that we didn't achieve integration of the 3 layers.

3. The development process

My first task was reading the descriptions of the Genbank file format at http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html and http://www.insdc.org/files/feature_table.html, after which I had a better grasp of the data I was dealing with. The large number of features and feature qualifiers that were described made it essential to get a feel for the records myself. If I could rule out the existence of some rare features in the chromosme 3 records, I wouldn't have to waste time writing code for those cases. For instance, there were no records with features like: join(complement(4918..5163),complement(2691..4571)).

For parsing my first thought was to use the Biopython package in python, since it probably includes most or all of the operations I needed to carry out on the raw genbank data. However, I soon found out that it was not as simple as it seemed, and required trawling through a lot of documentation. This stumbling block, and the realisation that it might help me get a better feel for the data if I dealt with it more directly, meant I opted for a different route: a combinatoin of regular expression patterns and python's built-in functions for manipulation of strings, lists, etc.

I could get to work on the parser straight away, after having decided on the minimum list of data that was needed in the whole database for each record. This first stage of this was creating a set of regular expression patterns that were sufficiently selective as well as efficient. To this end, I relied heavily on the websites http://regexr.com/ and https://regex101.com/ for real-time building and testing of patterns. The latter displays the number of 'steps' taken and the time in ms for a particular expression to search a given string. In addition, simply using Ctrl-F to search the genbank text file helped me notice important things. For example, '//' should not be taken as the delimiter, because it is also found in URLs in the COMMENT or JOURNAL sections of some records. Therefore, '\n//\n' is used instead.

Before any parsing could take place, the genbank file for chromosome 3 required processing to remove unwanted records. I used regular expressions to match records that have the attribute 'pseudo' (and so have no coding sequence) and records with joins to remote records. I could deal with splice later on in the parser by only extracting the information from the first 'CDS' feature of the record. However, I couldn't work out how to go about omitting mutations and variations from the data (e.g. the gene CCR5 show up as around 20 different records). In some cases the words 'mutant' or 'variant' are explicitly mentioned in the product name, or the field 'FEATURES variation /note=""' indicates variation e.g. an SNP. But many don't, so I wasn't sure which records to delete. In the end I simply kept them all.

The next stage was to implement the parser. At first, I made a single long script for each table in the database. Later on I realised that a functional programming approach would be much better, so I created a module of parsing functions – 1 for each field. These functions could be imported into the parer script, which uses them to write simple data files with fields separated by '|' and each record on a new line. These data files could easily be bulk-loaded into the mysql database.

Before data could be loaded, the tables had to be created. In order to determine the appropriate data type (e.g. varchar or mediumtext) for each column, I ran a few short scripts that gave me the maximum length of each field.

Once the database was set up, I could run some sql select queries to further check the data and look for any anomalies that may have slipped through the parser.

I could then move onto data access. The only method I had been taught was using mysql.connector, which is not supported by python 3 .5 onwards. The hope server for Birkbeck users does not allow the installation of new packages. Pymysql and MySQLdb were not available on hope – the only option was mysql.connector. Luckily, the hope server was running python 3.4.5 at the time, which supports mysql.connector.

4. Code testing

Most of the database tier development did not require input from the other layers, so I was able to do it myself, by running short scripts (e.g. one of the parser_module functions) and checking if the outcome is as expected (unit testing). The unit testing I did wasn't systematic and foolproof – I didn't make separate functions that compare expected output with actual output, rather I did it by eye.

I didn't use the python debugger (pdb), as I felt I was able to gain enough information through manual detective work. Most of the time the errors returned after running a script were clear enough. However, the pdb would probably save me a lot of time in the long run if I learn how to use it properly. It may have also helped me to get the parser and data access tier to work without errors.

5. Known issues

- The website isn't integrated with the database and middle layer
- There is no API for search functionality
- There is an error in the data access tier. When retrieving results by accession, the following type of error occurs:
  - print('coding seq for record AB005803: %s' % get_coding_seq(AB005803))
  - NameError: name 'AB005803' is not defined
- There is no shell script to set up the database – it has to be done manually
- Mutations and variations of the same gene are included in the database

6. What worked and what didn't - problems and solutions

A few of the main challenges that the parser successuflly overcame were:

- dealing with the heterogeneity in the CDS positional information (they could be a single range or a join of multiple ranges, forward strand or reverse complement, complete or

partial; this gives $2^3 = 8$ different types). Not only did several re patterns need to be written, but also the reverse complement numberings had to be adjusted
- ensuring the coding sequence for the partial records was in the right reading frame – otherwise the codon usage calculations would be completely wrong
- dealing with ambiguous nucleotides when creating the reverse complement
- changing the coding region positions for reverse complements (in the parse_full_cds_coordinates() function).

7. Alternative strategies

A few alternative strategies for my part of the project:
- Since there were errors in the data access, I could not judge its performance. If it was too slow, a more denormalised database structure, and the use of complex queries and/or views, would have been necessary.
- Object-oriented programming rather than functional programming. I didn't really look into it, so I'm not sure exactly what the advantages or disadvantages would have been. Perhaps it would have helped get rid of the redundant/overlapping parts of my functions. With so many separate parsing functions there are potentially a lot of places where the data integrity could be compromised.
- Using the biopython package

8. Personal insights

Having only picked up python and sql programming a few months ago, this project was certainly unchartered territory and provided a steep learning curve.

I learned that you really need to get to know your data and you can't make too many assumptions about it. Comprehensive testing of expected vs. actual outcomes, and then looking back at then raw data to figure out what went wrong, is very necessary.

One of the main insights was that writing code always takes several times longer than anticipated, because debugging is most of the work. A few times I found myself writing long chunks of code or completely changing the structure of the code without testing. This meant that when I got around to testing, layers upon layers of errors had built up, and required a long time to disentangle.

I hugely improved my python programming abilities, through countless cycles of error messages, searching stackoverflow, and playing around with test scripts.

Considering all of the dead-ends, de-bugging and re-writing of code, it would probably have been quicker to make a paper copy of the database. However, the next time tackling a similar project would be much quicker.

Overall, I gained experience in working on a 3-layer piece of software. Previously I was only vaguely aware of what terms like 'front-end', 'back-end' and 'middle-layer' meant in practice.