BookCatalogue: IS Project

Carson Foster

Initial Goals

- Maintain a database of books with information and also user-defined tags ✔️
- Allow the user to create, update, and delete books ✔️
- Allow the user to search for books based on stored information or tags ✔️
- Allow the user to search based on complex queries with boolean expressions ✔️
- Allow the user to submit a book's summary or "blurb" to have a neural net predict the book's genre ✔️
- Have the neural net be at least 80% accurate ❌
- Provide a GUI for the above tasks ✔️
- Provide a graphical installer for the packaged application ❌

Additional and Removed Functionality

Additional Functionality

- Alerts: the program can show pop-ups for information about errors
- Logging: the program logs its errors to a log file
- “Nodes”: the search GUI allows for the construction of queries through a drag-and-drop mechanism (think Scratch)

Removed Functionality

- Accuracy of the AI: the AI is only ~60% accurate on training data
- Graphical Installer: as of right now, there is no installer for the project (but this might change later)

Software Used

- Visual Studio Code: IDE with configurable build system
- MySQL: database management system
- Powershell & Batch/CMD: scripting for build process
- Maven: the package management and build system for Java
- Java 11: the language driving the project
- Other Java libraries: provide additional functionality
  - Hibernate and JavaX: integration with the MySQL Database
  - JavaFX: GUI library
  - deeplearning4j: AI library
  - (originally) easy-bert: word embeddings, replaced with a package from deeplearning4j
- Datasets
  - BlurbGenreCollection-EN: training, validation, and testing datasets for the AI
  - glove.6b.100d: word embeddings dataset

Problems

There were a lot

Problems: Part 1

- NetBeans (and no other IDEs I could find) support projects that are both modular (use JPMS) and use Maven, so I ended up building my code manually with Maven and command-line scripts
- I couldn’t successfully **require** Hibernate in my module-info, as Hibernate isn’t modular. I learned about automatic modules and was able to **require** it with the automatic module name from the Hibernate jar filename.
- Java couldn’t find the Hibernate module at runtime, so I had to create a Powershell script to parse the locations of the dependencies of my module and create a module path to pass to the java command.
- Hibernate wouldn’t run, which I eventually found out was due to a bug in Java 9: two Oracle libraries necessary to interact with the database both **export** the same module, which causes an error. I upgraded to Java 11, which fixed the issue.
- The query class used a pseudo-infix style (e.g. `queryObject.hasTag(“cool”).and().isAuthorLast(“Rowling”)` to search for a book that is both marked as cool and was written by someone with the last name Rowling). This doesn’t naturally lend itself to computer comprehensibility and wouldn’t actually end up working. I fixed this by switching to a pseudo-prefix style (e.g. `queryObject.and().hasTag(“cool”).isAuthorLast(“Rowling”).endAnd()`)

Problems: Part 2

- Hibernate is supposed to generate a “static metamodel” class for each class/table, but it didn’t; this fix was pretty simple: I wrote the static metamodel class myself.
- The query didn’t perform as expected: a query for books with tag A and not tag B returned books with tag A (and some still with tag B), and a query for books with both tag A and tag B returned no books (even though there are books with both tags). I eventually figured out that I had written my Hibernate query naively: it was checking for all books with a tag equal to A and not equal to B (which is all books with tag A) or all books with a tag equal to A and equal to B (no books). I managed to create a “contains(X)” query though: count the number of tags equal to X on a certain book, if the count >= 1, that book contains tag X. By checking for containment rather than equality, I was able to solve this problem.
- The methods I wrote to remove books that matched a query didn’t work; again, the fix was pretty simple: I had forgotten to wrap the deletion in a transaction and commit the transaction to the DB, so I added that code.
- Pretty much the only major problems I ran into in the development of the GUI were aligning issues: buttons wouldn’t be the right size, things wouldn’t take up the whole width they had, etc. These issues were solved by GridPanes, extra containers to regulate containers of buttons, and setting properties.

Problems: Part 3

- After putting the dependencies in the POM and requiring them, Java still cannot find one of the dependencies of deeplearning4j, called nd4j-native-platform. This is because DL4J isn't modular, so the names of the jars are being converted into module names (e.g. nd4j.native.platform); this doesn't work for this package, since **native** is a Java keyword. I tried a few things to fix this, but none of them ended up working, so my fix was for the AI section of the project to not be modular.
- I had a general problem with the DL4J documentation, which was very poorly written. Unfortunately, there was no fix for this problem.
- After I had created the data processing layer/step and read more (bad) documentation, I realized that the schema that I have the data in isn't compatible with easy-bert and DL4J. I ended up using a completely different system for processing the data, using a few natural language processing classes from DL4J and downloading another pre-built word embedding model.
- DL4J wouldn't load the word embedding model. After some painstaking research, I discovered that this was, in fact, a bug in DL4J. However, there was a workaround: if you temporarily disable the buggy part, load the model, and re-save it in the binary format (instead of the default text format), you can re-enable the buggy part (which works with a binary format, and ensures that the program is more efficient), and work with the binary model with no problem

Problems: Part 4

- After some research, I discovered that the structure of a text classification neural net is much more complex than I had previously thought. I found an academic paper that provided a description of a neural network that they claimed performed well, and translated the description into Java with DL4J libraries.
- The class I had written to load a blurb and its label (the genre) was functioning incorrectly, which I discovered was due to the fact that the data file was in the format UTF-8, while the class was reading it in as plain ASCII, causing problems. I changed the class to read in the file with UTF-8 format, which fixed the error.
- The neural net was only ~50% accurate, so I sent an email to the people who provided the dataset (since there was an issue with it), and tweaked various hyperparameters. I was only able to get the AI to be 60% accurate, though.
- The AI section isn't modularized, while the rest of the project is, which causes problems. The solution to this was to compile and package the AI section separately into a jar, and then **require** it as an automatic module.

Tasks that were unexpectedly easy

Don’t make me laugh… It’s nothing. Nothing was unexpectedly easy

Lessons Learned

In general, I learned a lot about:

- Maven and manually building Java applications
- Relational databases and SQL
- Using Hibernate to link Java code and a database
- Aligning GUI elements correctly
- deeplearning4j and its data input methods
  - (and its NLP classes, to an extent)

A few specific lessons I learned were:

- It's never as simple as it should be. Plan accordingly.
- Neural nets are very complicated, and hard to debug/improve.
- Maven is very handy at handling dependencies; its plugins are also very useful.
- Modular Java adds just as many problems as it solves.