COSC 425 Project Proposal

Dr. Joseph Anderson Dr. Karl Maier

Abstract

The goal of this project is to enable researchers to begin leveraging the capabilities of Natural Language Processing (NLP). This is accomplished by building a Graphical User Interface (GUI) using the tkinter and PyTorch libraries for Python. In its current form, the application allows a user to import a Zotero file which contains the metadata for a set of academic publications, train a neural network to classify them based on tags within the file (providing the ability to adjust the various model parameters of the network), and then classify new or unlabeled documents using the trained network; additionally, it provides some rudimentary summative assessment and records of the training phase of this neural network.

Here we provide a technical overview of the system, and outline the next steps needed to make it more capable, robust, and accessible.

In contributing to this project you will get experience with machine learning, GUI development, and professional-grade software development practices with a source control system.

1 Organization

Currently the software, (lacking a real name – TBD), is set up to provide four main interfaces, detailed below: 1) Importing a Zotero library file to inspect the papers contained therein, and classify them based on a loaded neural network model, or classify a single paper by entering its title and abstract manually; 2) setting the parameters of a neural network and running the training process; 3) a user manual; 4) a dashboard to inspect the performance statistics and attributes of the trained model. The following sections will describe each screen with images, and known bugs/issues. The following tags will be used to describe important things to do:

- **Bug**: a blocking issue that needs to be fixed for general use.
- **Improvement**: an issue that does not hinder the use of the program, but would make the user's life easier.
- **Test:** A feature that needs more extensive testing to verify that it works correctly, or the documentation/interface updated to match its functionality.

Also in the application, you can find the "Manual" tab with some explanations and reference documentation; the content of this tab is the `manual.md` file which you can read using any text editor. The source code also has many good comments to help understand the flow and structure.

A couple of general known issues to be aware of and to fix:

1. Cross-platform sizes differ (because different operating systems have different default "styles"). This results in some buttons being too big, or not big enough. This can be fixed by setting different styles for widgets. Maybe look into the Python `ttk` widgets, a more modern solution inside the `tkinter` system?
2. There is a general lack of error-handling.
3. Console logging can be improved.

4. Need a better, more presentable `README` for the GitHub page.

5. The code should be re-organized a bit so that the application itself is at the top level, and is the main part of the repository (this may be done by the project client, instead of the SWE group).

6. We want to be able to compile the entire project into a single `.exe` or cross-platform executable that can be distributed by non-technical users.

1.1 Development Process

If this project is done by a COSC 425 group, this project will work to simulate a fully professional software engineering experience. The group will each get access to a GitHub repository of the code.

Auxiliary requirements for the project:

1. Create each issue found in this document on the GitHub repository for easier discussion and monitoring.

2. Maintain good coding and version control practices: commit early and often. Each change should be done via a pull-request to the `develop` branch and have a code review by a team member. Then, pending approval by the project client/mentor, it can be merged into the `master` branch. For a detailed guide to the standard workflow of contributing to a project on GitHub, check the guide here: `https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project`

3. Coordinate and communicate often with your team, project leader, client, to stay on track.

2 Testing

This is the part of that interface that allows a user to import their Zotero library, See Figure 1. Zotero is a popular tool that manages research papers or documents that often need to be organized or cited during a research project. The format of the Zotero library is `.rdf` and currently this software uses the Python `rdflib` to perform the import and parsing of the file from the user. After it is imported, the documents are converted to a `.csv` format that is will be usable by the neural network for classification.

The workflow of this screen is:

1. Use the "Choose File" button to select your Zotero library as an `.rdf` file. Note that for the purposes of classification, the correct labels must be in the file under the "tags" sectio n of the metadata – this is also where any new, learned, tags will be written later (See the "Classify" button).

   (a) **Improvement:** This is sometimes cut off by the text showing the name of the file, need to re-arrange the layout to fix this.

   (b) **Improvement:** This could automatically, after a successful read of the file, perform the "Convert to CSV" function, instead of taking two clicks by the user.

2. Click the "Convert to CSV" which will parse the file and store a `.csv` representation in the `.data` folder that is located in the same directory as the GUI application.

   (a) **Bug:** This should then populate the table below with all the titles and abstracts contained in the file. This happens in certain scenarios but not in others (as in Figure 1).

3. The user can then see the titles and abstracts in the table below for the imported library, and use the search feature to filter them if the user wants to find a specific one for classification. These can also be copied into the left panel via the "Send" button to see a one-shot classification and allow the user to accept or override the predicted value, and then optionally write it to the `.rdf` file.

   (a) **Improvement:** Currently, the Tkinter-table used for the searching the csv will have column (title) text bleed over to the adjacent column.

(b) **Bug:** If someone attempts to select multiple rows and uses `shift` or `ctrl`, some errors are thrown in the console. The program can still run when this happens.

4. The "Classify" button will run the prediction algorithm over the imported library and save the predicted labels as "tags" in the `.rdf` file.

   (a) **Test:** This process needs to be tested to verify that it works correctly.

   (b) **Improvement:** Ideally, the user will first select a fully classified file with correct tags already present, build and train the neural network, then return here and import a new and unclassified one to be labeled by the trained model. This flow is not quite obvious, so the interface and documentation should be updated to make it clearer.

5. On the left, a user can enter the title and abstract of a paper not in the imported file and then use the "Predict" button to run the built neural network over the entered text. They will then see the predicted labels and the confidence of the prediction. The "Confirm" button will take the prediction with highest confidence and write it to the file. The "Override" button will take the label from the dropdown at the bottom left and write that to the file instead; note that the set of labels can be customized on the "Building" tab.

   (a) **Bug:** Both of these functionalities need to be implemented.

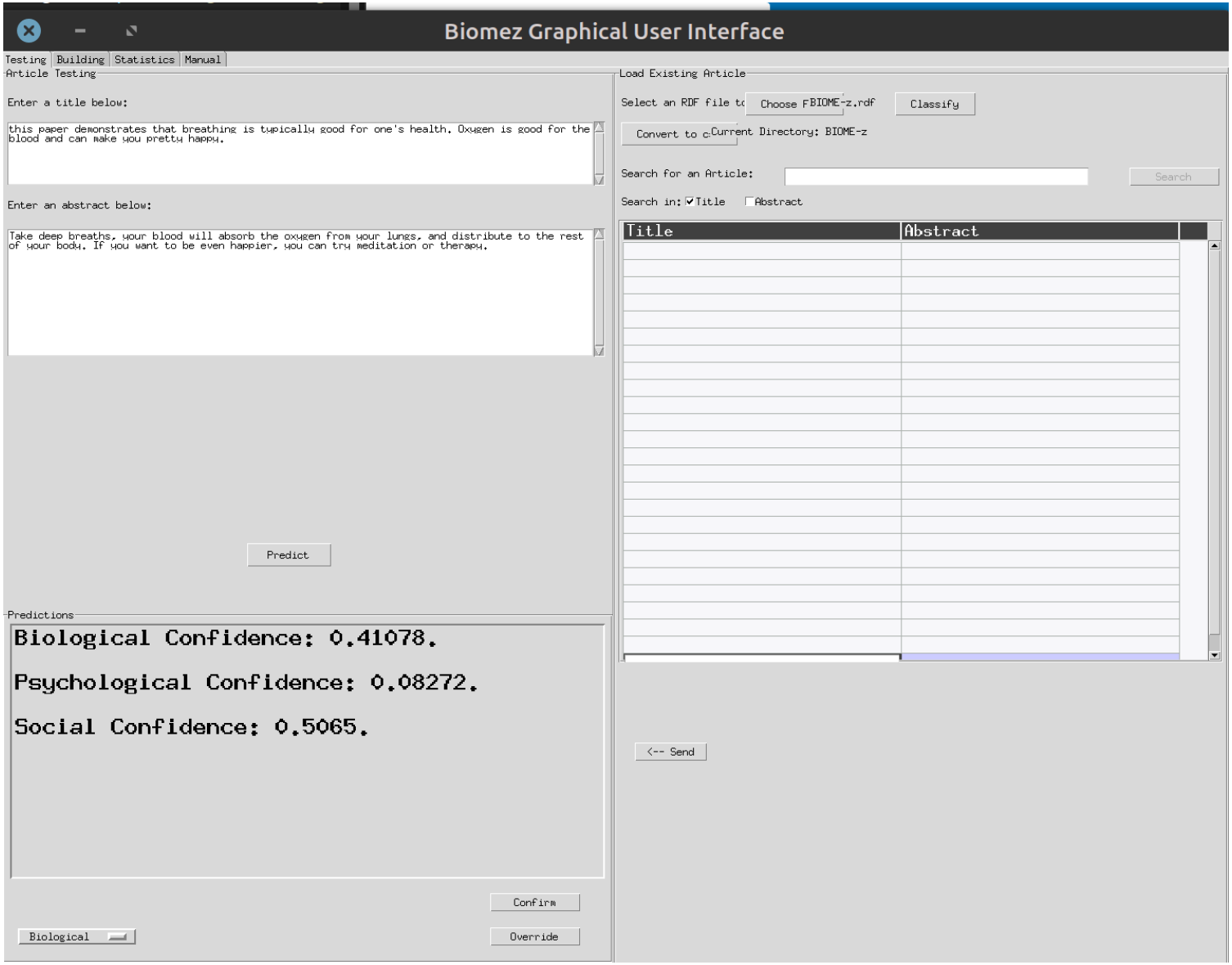

Figure 1: The Testing screen.

3 Building

On this screen, Figure 2, the user can configure the various parameters of the neural network and run the training process.

1. The "Select Model" will let a user select a folder that contains the `.csv` files needed to build the network. After importing on the Testing screen, there will be a folder within `.data` that shares the name of the `.rdf` file. The user can select this file and the program will detect the needed files to build and train the network. If the model was built before, it will load the parameters and model files that are present – any further training is optional.
   (a) **Bug:** Parse the rdf file a little better (some html still gets passed into the NN).
   (b) **Test:** The loading of pre-trained models should be tested again.
   (c) **Improvement:** The selection process is somewhat confusing, and not obvious that the user is to select one of the `.data` folders, instead of a single file. This could be made more user-friendly by adding some text/explanation, or perhaps combining all the model files into a custom blob that can be selected as a file.
   (d) **Improvement:** The name of the current model folder is not shown, this could be added easily.
2. The "Edit Labels" button can adjust the pool of possible labels that the neural network will look for in the training dataset (the `.csv` imported on the Testing screen). This should enable one to classify documents with any set of labels that the user wants, as long as the training library has the same labels present.
   (a) **Bug:** The last time this was tested, it did not work. Both the ability to add and remove labels had no effect.
3. The "Parameters" section allows some simple ways to adjust the model parameters. For the meaning/purpose of the parameters, refer to the PyTorch documentation.
   (a) **Improvement:** Currently the ranges for many of these are fixed, but they are not necessarily the best choices. It could be nice to add the ability to set custom values outside the provided range, but would need to be done carefully in order to not over-complicate the interface. Perhaps allow an "advanced" dialog that overrides the sliders?
4. The "Set new Default Parameter" button saves the current selection as the program defaults for the future.
   (a) **Improvement:** This is worded oddly and not an often-used button, so could probably be moved/explained better.
5. The "Build Neural Network" button runs the PyTorch training process from scratch with the specified parameters. This will reset any previous training and start with random weights inside the model. The progress of the training is shown by the progress bar at the bottom of the screen. Note that if a CUDA-enabled GPU is detected, it will use it to accelerate the training process – this can be monitored at the top-right of the screen, e.g. "Running on: gpu".
   (a) **Bug:** Fix the neural network building function(s), so it will build with a label that does not exist in rdf file. For ex: "Social Studies" has 0 instances in the BIOME-z data, therefore will endlessly loop when building.
   (b) **Improvement:** This currently uses very rudimentary text-classification. Could use some work to enable more advanced or state-of-the-art techniques. Even better would be the ability for the user to "mod" the program to add custom models.

6. The “Train” button will use the currently built model (not resetting the weights) and train for as many more epochs as are specified above. Because this does not start over, any changes to the other parameters should be ignored. The results of the training should be appended to the stats for display on the “Statistics” screen.

   (a) **Bug:** This currently does nothing.

7. The “Set Model Parameters” saves the parameters as a loadable file to the model folder. The main purpose of this is so that when the user loads the model later, the parameters adjust to be what they were when the model was built.

   (a) **Improvement:** This should happen automatically when the model is built.

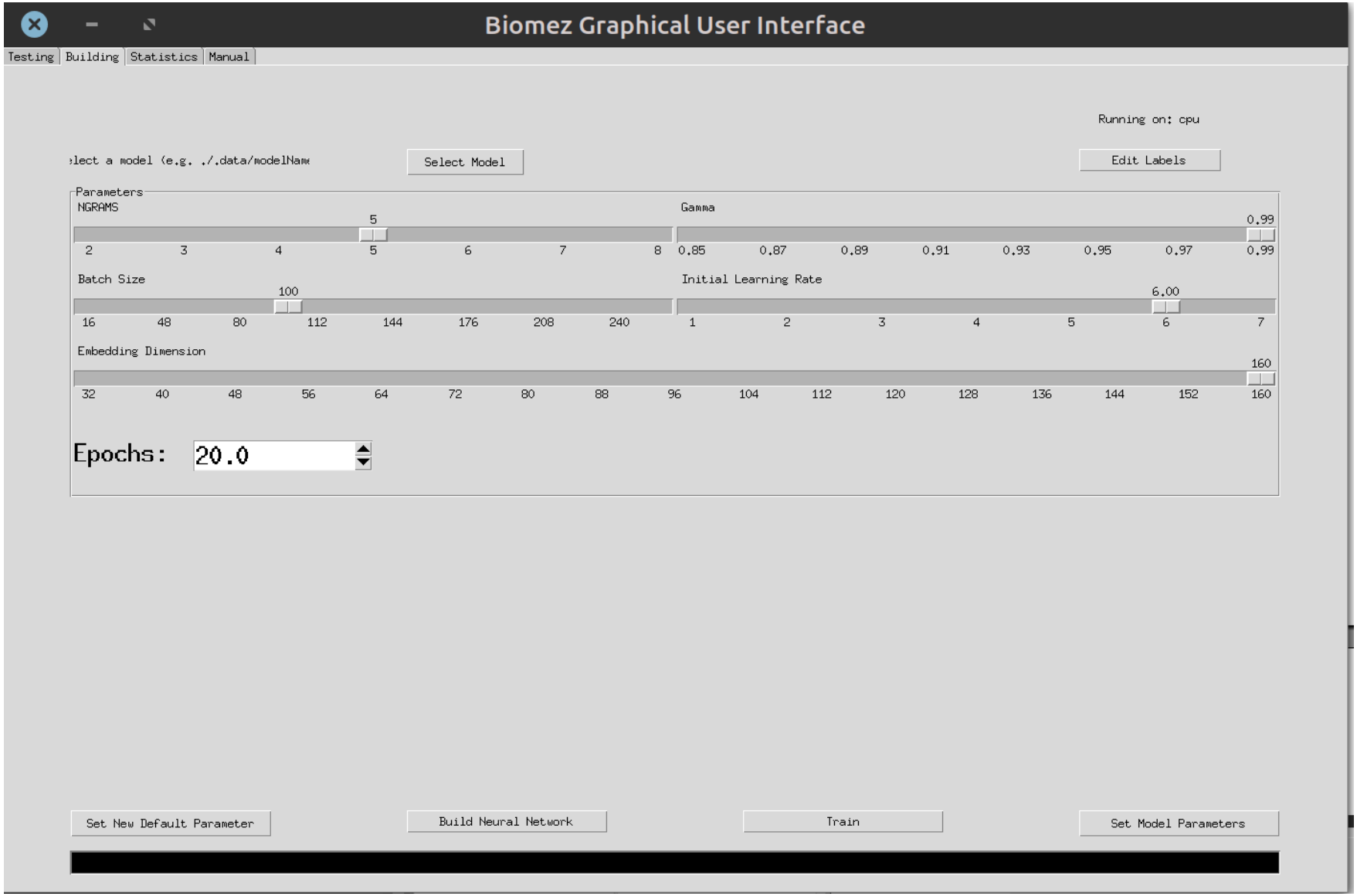

Figure 2: Screen to configure the neural network and run training.

4 Statistics

This page, Figure 3, gives the user some breakdown of how well their neural network performed over time while training on the data. See machine learning literature for a break down of training vs. validation data and loss/accuracy functions. When training, the training and testing data is selected randomly and a breakdown of label presence is given by pie charts.

The results of multiple training/builds will be saved so that the user can switch back and forth to select the parameters that behaved the best. The results data can be saved and loaded via the "Save" and "Load" buttons. The "Prev" and "Next" buttons will move between different result sets, the parameters of that run will be shown to the right.

1. **Bug:** In the `saveGraph` data function, Pie graph and General Run data are not saved to the csv folder. Then in the `loadGraph` data function, load the new csv file data into the `stats_data` class. This will allow the Run data and Pie Charts to populate when using the `load` button on the toolbar.
2. **Improvement:** Add time and date to run meta-data.
3. **Improvement:** Move run parameters (Ngrams, etc.) to the "Run #X" content panel.
4. **Improvement:** These results are aggregated between trainings, whether or not the user built the model from scratch. It would be helpful to denote which ones are simply further training an old model or which ones are starting fresh. However, this can also be left up to the user's own organizational ability!
5. **Improvement:** The label proportions in the training and testing data should be the close or the same.

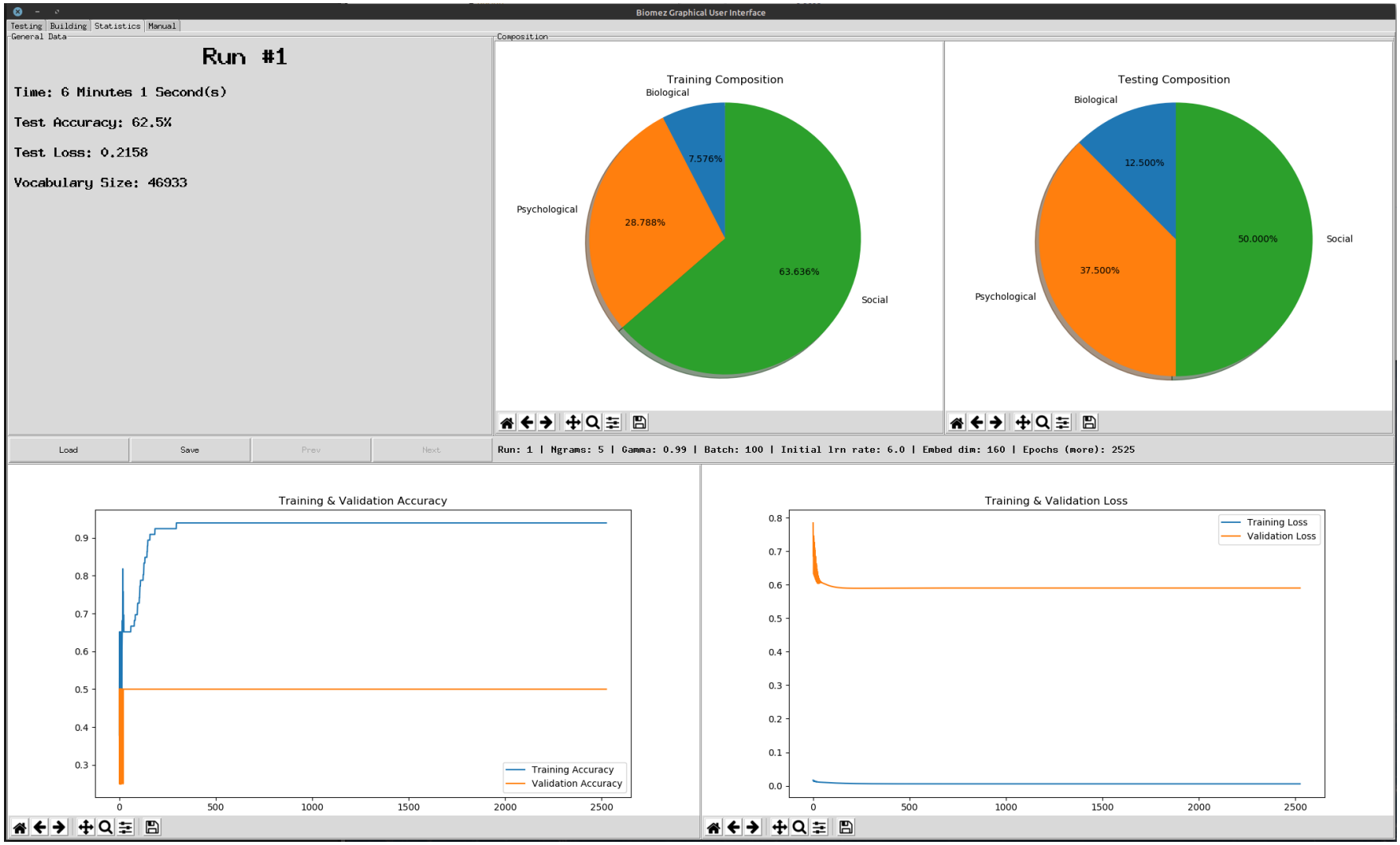

Figure 3: Screen to see the results after building a neural network.

5 Manual

This panel loads and renders a markdown file, `manual.md` into a window as html. The content can be adjusted by editing the markdown file only.

1. **Improvement/Bug:** The content of this could use a lot of updating and re-writing for clarity.
2. **Improvement:** It would be helpful to and annotated screenshots as part of this.
3. **Improvement:** The text is scrollable, but a scrollbar is not present, or any visual indication that there is more text.
4. **Improvement:** The content specific to each section could also be useful on their respective tabs of the interface – it would be good for the user to bring up the manual for the Testing screen from the Testing screen itself, possibly as a new window.

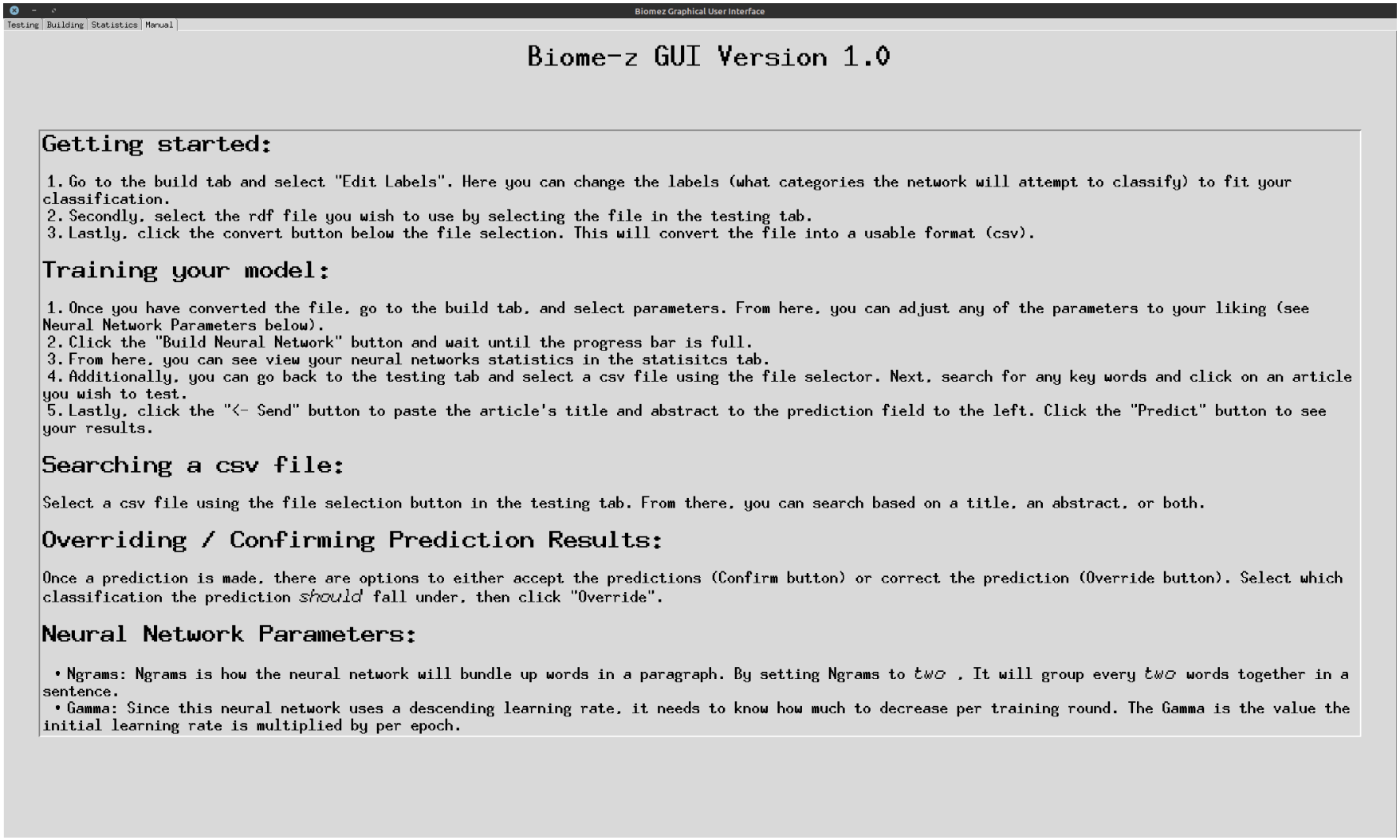

Figure 4: The User Manual