Leen Arnaout (U07624704)
BE/EC552 Final Project
6/11/2020

Project Title: crRNA Go!: an ML-based crRNA Designer for CRISPR-Cas13b RNA Virus Targeting Systems

Project Summary and Motivation

crRNA Go! is an online, machine-learning-based application that can be used to help predict the most suitable binding sites for CRISPR-Cas13b on single-stranded RNA virus, and determine the binding energy associated with the binding of Cas13b proteins to the specific binding sites. This project was conceptualized to help develop effective Cas13b mechanisms and crRNA tools that can be used to help detect viruses in clinical samples. For example, one such virus that can be detected using Cas13b is COVID-19. The advantage of using Cas13b diagnostics over regular antibiotics and PCR assays is that sometimes, the clinical samples we have that may contain the virus are not associated with any immune cells or antibodies so the cells may be infected but we cannot know this since no antibodies exist in the sample. Also, Cas13b is usually more accurate than the PCR method so the level of false positives is less.

The application is three-pronged: the first component is the machine learning algorithm. Users can input .csv files that contain the RNA sequences of the virus and associated binding energy scores (8nt (Seed) Accessibility Threshold, 16nt Accessibility Threshold, Self Folding Energy, Sequence Asymmetry, Energy Asymmetry and Free End access. The last three metrics relate to how structured vs un-folded the RNA is). The algorithm is a bidirectional LSTM. The second component is using a previously-designed machine learning model to predict the binding sites on an input RNA sequence, and the associated energies with that - this will be printed to a .csv file for future reference. The third component is a crRNA builder: crRNA is the guide RNA that is used by a Cas13b protein to target a specific site on an RNA. So, the user will input the RNA sequence for the binding site, and the website will return the sequence in the appropriate crRNA chassis that can then be sent off for ordering from a company like IDT for use in lab samples.

Summary of Code Elements

All code elements are well-commented so that a person reading it can know exactly what each code snippet does in the web-app. But, I have explained what each code snippet EXACTLY does in the HTML front-end, the Python back-end implemented using flask, and the Python script used to generate the sample data, as well as my attempt to generate test data using RNAxs (http://rna.tbi.univie.ac.at/cgi-bin/RNAxs/RNAxs.cgi)

- Front-end HTML/CSS page templates:
  The front-end pages were designed using Bootstrap, an HTML, CSS and JS library that is used by web developers to design web-apps

(https://getbootstrap.com/). The Bootstrap library has some different native elements (buttons, progress bars, page dividers etc) that a person can take the code of and implement it in their website while modifying these barebones elements, like changing their color, adding an href attribute, etc). I used this to design the five template pages that the webapp uses. All the pages have a set of shared elements that don't change: at the header of the HTML doc, we have required meta tags and the Bootstrap CSS links to support the right formatting. Then, the body comes, where everything is ordered in a set of flex columns. At the top of the page comes the title of the program, the buttons to navigate to the different parts of the webapp (train, test and chassis build). Afterwards, in another division of columns, we open a form (to be able to POST methods, which are the queries to do a certain action like train a model or test it), and the said form ends at the button that submits the query for the train, build and test pages. For the pages that display the results, we do not have a form: rather, we just display the outcome images and allow the users to download the results of their queries (be it in the form of the result .csv of testing their ML algorithm, or the .h5 of their trained algorithm). At the bottom of each HTML document, we have optional JavaScript that can be used to carry out more complex functions that make the webpage interactive, rather than just displaying static elements. The five said pages just differ in the contents of the body, as outlined here:

a. Training page: (train.html) this is the page where the user can specify the parameters for training their Machine Learning algorithm that this tool uses. It can be accessed using the url (/`train`) or the default url. On this page, we have a form (line 27: `<form action="" method="post" role="form" enctype="multipart/form-data">`) that can allow the user to upload the .csv document that specifies the training data that the user will use to train the model using the upload button on line 29. There is a blue hyperlink button on line 33 (`<a href="https://srv-file7.gofile.io/download/3k9Z8h/test_data.csv">`) that can allow the user to download a sample dataset that's 70,000 entries long. In the second step, the user can name the machine learning model they develop by entering the name in a textbox on line 34. The user can then select how many epochs they want to train their model for on line 35 (number entry text box) and control the size of the training batches using a similar number text box (line 36). Then, when the user hits the "Go" button on line 40, the `POST` method is evoked in which the uploaded and entered data is submitted to the flask backend where the python code for training the algorithm and generating the output (loss diagram and .h5 downloadable model) is generated.

b. Training results: (train_results.html) after the `POST` method from the training page is evoked, the ML model is trained and built, and the .h5 model and the loss graph are saved to the static folder inside the flask app. On the training results page, the loss graph - sent into the train_results.html page as a `{{lossgraph}}` object from flask (line 28) is displayed inside a flex column division. Then, on line 33, there is a button to download the ML model built, with the href referencing the `{{downloadmodel}}` object that was built in the flask app on flaskapp.py. This model is named using a timestamp and the name that the user specified in the training page. This .html page is displayed with results intact if it is called from the training page, at the url /`train` using the post method, this HTML template is displayed with all elements intact. If called using the `/train/results` url, then the image and download button are blank because it did not come from a post method.

c. Binding site finder page: (binding site finder.html) on this page, a form is used (line 29: `<form action="" method="post" role="form" enctype="multipart/form-data">`) to allow the user to submit their query to find out the best binding sites for Cas13b on the genome of the virus that they have entered in the text box, based on the training model (.h5 format) that they have uploaded. There is a hyperlink (line 33) for a test .h5 model, that the user can download then upload into the tool to use for their query. There is also a blue hyperlink button (line 38) for a sample RNA sequence of a virus that the user can put into the text box to test the tool upon, and this is taken from a SARS-nCov-2 sequence from China that was sequenced in December 2019. In addition, the user will then specify a binding site size for the tool to use (line 39) to process that query. Pressing the “Find Binding Sites!” button on line 44 will then `POST` the form to the flask backend that can then process the form and return the binding site finder results at the same url (/`binding`) but with the binding site results.html template.

d. Binding site finder results: (binding site results.html) After the `POST` method from the build page is evoked, the input RNA sequence is analyzed using the ML model that was loaded by the user, with sequence lengths that are determined by the user. After the query is processed, the resultant image that shows the location and directionality of the top 5 binding sites is displayed on the page in the center, being coded for by the `{{image}}` element in line 31 of the code. In addition, the resulting .csv file that displays all binding sites ordered in terms of their binding energy (best to worst) is saved with a timestamp and loaded into the variable `{{download_file}}` that is the href attribute of the download results button. Once that button is clicked, the .csv file is downloaded (line

35). This .html page is displayed with results intact if called from the building page, at the url /binding using the post method, if the post method is used. If called using the /`binding/results` url, then the image and download button are blank because this page was opened without the post method used.

e. Chassis designer page: (designer.html) this page is the simplest in the entire webapp, because the form it uses is just a simple no-action form: `<form  action="" method="post" role="form">` Inside that form, the user will input the RNA sequence they want to design a crRNA for into a text box. There is also a hyperlink button (blue text button, line 35) that the user can click to view a test sequence on another page, that they can copy and paste into the text box just to test out that feature. When the user clicks the "Design" button, a `POST` method is evoked in which the input of the text box is converted into its reverse compliment RNA sequence and placed into the crRNA chassis, and then viewed as text below the design button in big font. This result is coded as the `{{output}}` element in line 43 of designer.html, where it is blank when no method is posted, and is replaced by the result when the query is posted to the original flask executable code.

- Back-end Python Code on flaskapp.py:
  The flask app starts with importing the necessary libraries and elements (lines 1-22). On line 27, we declare the app, and we define the upload folders where the root file of the flaskapp is located on lines 28 and 29. Afterwards, we declare functions we use later on in the app in many of the sub-components of the webapp. The first one, on lines 35-46, is `DNA_to_RNA`, where a read-in from a .csv file is converted from a DNA sequence (ACTG) to an RNA sequence (ACUG), made uppercase, and stripped of all non-base pair elements like spaces and tabs. Then, from line 49 to 65, we have the function `one_hot_this`, which one-hot encodes the RNA sequences. Essentially, what this means is that it converts the RNA string into an image. Each row in the one hot array presents an index location in the string, and each column represents a base pair: A, U, G, or C. If we have an A in position 0 of the RNA sequence, we have a 1 in the first row of the array, in the first column corresponding to A, and the rest of the row is zeros. Then, if we have a G in the next position, then we have a 1 in the third column (the G column) and then zeros in the rest of the columns in that row, and so on. This function returns a one-hot encoded index. Then, for each page in the web-app, there is an `@app.route(url)` clause in which, underneath it, we have a function associated with each url and what it returns (redirect to another page, display text, view template, etc). If a button on a page is clicked and a form is submitted, in the route we also declare `(methods = ['GET','POST'])` where the method we use is post.

a. Training the ML algorithm: when the form to train the ML algorithm is posted (line 85) and a file is submitted (the training .csv dataset, line 89) the file is read and saved online (lines 90-94) then the .csv is read and loaded (line 96). The .csv file is then split into energetics scores and DNA sequences (lines 98-100) and the energetics scores (6 of them) are summed up and divided by 60,000, such that the score is now a float from 0 to 1 (scaled) in lines 102 to 111). Then, in lines 116-120, we feed in every DNA sequence, convert it to RNA and one-hot encode it, and then make an array of the one-hot encoded arrays. We then split off the x (sequences) and y (energy scores) arrays into training and testing set (75% training, 25% testing, randomly, in line 130) and convert them to tensors so that they can be fed into the ML algorithm. Then, we build the ML algorithm (lines 137 to 155). It's a sequential model made out of a bidirectional LSTM layer with 20 filters and relu activation, dropout layer of 20%, a unidirectional 10-filter LSTM w/relu application, another 20% dropout layer, then we flatten the model to create a one-neuron output (our energy score), add a dense layer w/one neuron output. Then, we compile the model with binary cross-entropy loss, an Adam optimizer with a 10E-3 learning rate and an exponential decay of 10E-05 to prevent over-fitting and memorization of data. The number of epochs and the batch size is specified by the user and read in from the form, then the model is fitted, trained and saved using the name that the user specified, along with the timestamp. Afterwards, we plot the loss graph per epoch (lines 160 to 167) and save it, then we push the loss graph and model to the output results page at train_results.html.

b. Finding the binding sites: for the method binding_2, after a `POST` method is called, the .h5 ML model is loaded (lines 186 to 192). Then, the input DNA sequence that the user wants to find the binding sites in is obtained from the form, reverse-complemented, changed into a string and an RNA sequence, and the size of the binding site is loaded (lines 196-204). Afterwards, a list of the possible binding sites and their sequences (in the specified size) is done for the forward and reverse strands, and indexed in a dataframe along with their positions and the corresponding strand (lines 206 to 254). Afterwards, each input is converted into an RNA sequence and one-hot encoded (lines 258 to 268), then fed into the machine learning algorithm to predict their associated energy scores and that is written into the same dataframe (lines 262-277). Then, the dataframe is sorted so that the sequences with the best energy scores (closer to 1) are at the top (line 281), basically "descending order". After that, this dataframe is written to a .csv file, (line 286) where the binding site with its position, strand and energy score is written. Then, the top 5 binding sites are represented using a GenomeDraw diagram (lines 289 to 312) that shows their length, location and directionality. The .csv and `GenomeDraw` image are then pushed to the results screen with the `return render_template` command (line 314).

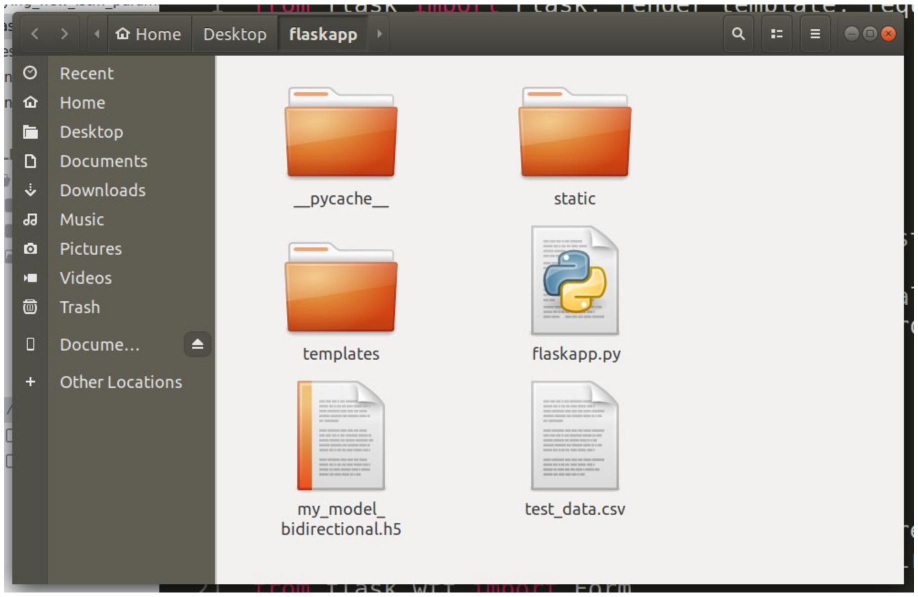
c. <u>Designing the crRNA chassis:</u> in lines 340 to 345, the user would have input the binding site they want to design a crRNA for in the text box on the /designer page, and pressed the button to `POST` this method. This binding site sequence is then converted into an RNA sequence, reverse-complemented and placed into the crRNA chassis that was designed by the Zhang lab at (https://zlab.bio/cas13). Then, this chassis is displayed in the returned `render_template` in line 347.

- <u>Python Script to generate test data:</u> on the arnaoutl/crRNA_Go_code, the file test_data_generator.py is the script that is used to generate the test data (test_data.csv) that is used to train the ML algorithms. It involves using random number generators, so the numbers generated are in a Gaussian bell-curve format. In the first two lines, we import the numpy library (used to generate random numbers) and the pandas library (used to write data to a dataframe and then to a .csv). We generate the 6 aforementioned energetics scores for 70,000 data entries (max. Value of 10,000, then will be scaled to 1 once we use that data to train the ML model), in the style of a numpy array (lines 6 to 11 of the code). Afterwards, 70,000 random DNA sequences that are each 1000 base pairs long are generated in a similar manner inside a for loop with the same length as the size of the energetics scores arrays (which allows for flexibility: if you're generating 5,000 entries instead of 70,000, the code allows that). First of all, a random array of 1000-number-long numbers (ranging from 0 to 3) is generated. Afterwards, it is converted into a string, with each digit replaced by A,C,T,G (A=0, T=1, G=2, C=3). There were issues with converting a numpy array to a string of letters, as in the conversion a lot of extra characters were added (array brackets, species, \r tabs) so those were stripped away, then appended to another array. Afterwards, the characters/DNA string array and the 6 energetics scores arrays were written to a pandas dataframe (line 35) labelled with the same labels used in the ML algorithm `('target seqn', 'Access 8nts','Access 16nts','Energy A.','Sequence A.','Self Folding','Free End')` to make sure that there are no reading issues when the .csv is read by the ML training algorithm. Then, in line 36, the dataframe is written to a .csv file that can then be uploaded to the webapp.

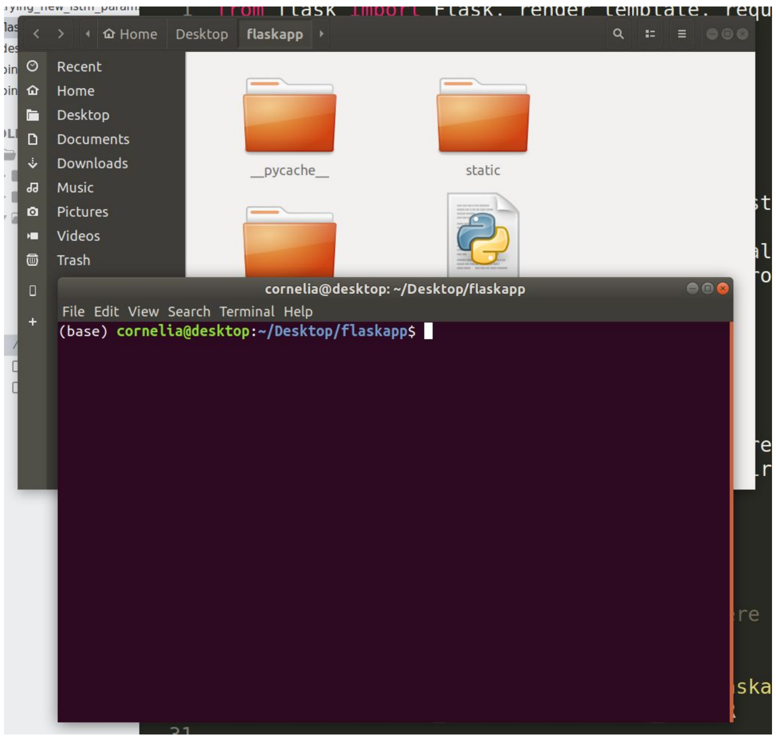
- <u>Sample data that was generated using RNAxs:</u> as mentioned earlier in this paper, the Zhang lab (leader in Cas13 research) recommended that, for designing crRNA for CRISPR-Cas13 systems, RNA folding and energetics must be considered and they recommended the RNAxs tool to be used as a guide to find the best binding sites for Cas13 proteins, the ones that require the least amount of energy input and are easiest for the Cas13 system to dock to. This is because not a lot of research has been done in this field for crRNA design, so the same principles from general RNA design have been implemented. So, I tried inputting the RNA sequences for around 120 viruses that affect humans into the RNAxs systems, and come up with sample binding sites and their associated 6 energetics scores using that system. The RNAxs website was extremely slow, only allowed me to generate binding sites that are 19 bp long at maximum, and it was a hassle copying the results from the website to a spreadsheet that I made for training. I then realized, with the help of my friend Quentin Lin (BU EE '20) that this dataset (310 sequences, 19 bp each) is too small to train my data on, and 19bp wasn't good enough to try and glean RNA characteristics and features for binding energy from. So, he recommended the generation of random data for RNA sequences that are much longer, and for a larger number of sequences, and then implementing that into an LSTM that can better identify patterns in the data.

Requirements for Running the Code

- **Co-dependencies:** to run, this project requires: Python 3.7.6, Flask 1.1.1, Werkzeug 1.0.0, TensorFlow 2.2.0, Keras 2.3.1, Pandas 1.0.1, PIL/pillow 7.1.2, numpy 1.18.1, Biopython 1.77 (this includes Bio.Seq and Bio.SeqFeature and Bio.Graphics), sklearn 0.0, time
- **To run the code and execute the web-app on flask locally, on Linux:**
  I used Ubuntu 18.04 to develop and execute this web-app.
  1. Download the flaskapp zip file from the Github repository at arnaoutleen/crRNA_flask
  2. Unzip the flaskapp zip file and place it in a directory that's easy to access:

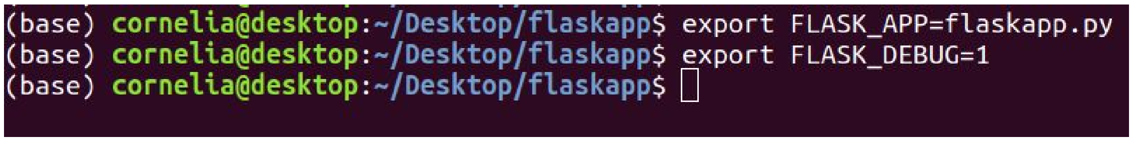


3. Open a terminal window in the root directory of the flaskapp file:



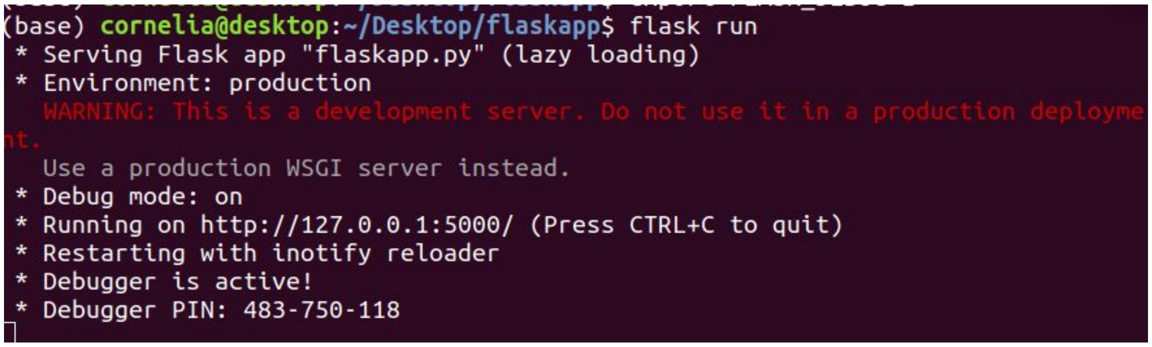
```
(base) cornelia@desktop:~/Desktop/flaskapp$
```

4. In the terminal window, set up the flask runtime environment by typing in export `FLASK_APP=flaskapp.py`
Then, set up the debugger by typing export `FLASK_DEBUG=1`, to set up the WSGI debugging server (it shows you exactly what in your code is causing the problem if you run into a problem while executing your code)



```
(base) cornelia@desktop:~/Desktop/flaskapp$ export FLASK_APP=flaskapp.py
(base) cornelia@desktop:~/Desktop/flaskapp$ export FLASK_DEBUG=1
(base) cornelia@desktop:~/Desktop/flaskapp$ 
```

5. In the terminal window, type: flask run to start the program:



```
(base) cornelia@desktop:~/Desktop/flaskapp$ flask run
 * Serving Flask app "flaskapp.py" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with inotify reloader
 * Debugger is active!
 * Debugger PIN: 483-750-118
```

The webapp will execute on localhost (`127.0.0.1:5000`) and it doesn't matter which web browser you use - given that it has CSS viewing capabilities.

- **To run the code and execute the web-app on flask locally, on Windows:**
    - You can install a Linux shell on Windows and execute the same commands that were previously mentioned above in the same manner, using a Linux Terminal.
- To run the code and execute the web-app on flask locally, on Mac:
    - A virtualenv needs to be set up, where python and a WSGI debugger are previously installed. Then, you can navigate to the root directory where your flaskapp is located, and then follow the same steps above.
- **IMPORTANT NOTE:** if you're running the app locally, you need to redefine `UPLOAD_FOLDER` in `flaskapp.py` to the root directory where you are hosting the flask app, and the static folder. This is in line 28 of flaskapp.py. In the sample code uploaded it is in: `'/home/cornelia/Desktop/flaskapp'.` Also, a WSGI debugger must be installed on your system such that you can debug the app while running it. If you do not wish to debug, you are free to not install the WSGI debugger, and the only other thing you don't need to do is that you do not need to run this line of code: `FLASK_DEBUG=1.`
- **IMPORTANT NOTE 2:** since test_data.csv is too large to download directly from the Github repository, I have instead put up a txt file called load_test_data.txt that contains the link to test_data.csv: you may use that link to download the test_data.csv file, then copy it into the root directory of where your flaskapp is.

ACKNOWLEDGEMENTS

I thank Hung-Ching (Quentin) Lin (BU EE Class of 2020) for helping me decide whether an LSTM or a CNN would be a better ML choice for this project, as well as how to design my datasets so that they are the correct size and input type (for the training set and one-hot encoding).

I also thank Max Ellsworth (BU CE Class of 2022) for helping me understand how to connect the back-end and front-end components of the webapp using flask, mainly how to push files to the static folder so that the user can then download them after clicking the “download” buttons.