|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Assignment  Cedric Hermans / MLOps@Home / 06-02-2022 |  |
|  |  |  |  |
|  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  |  | Table of contents |  | |
|  |  | [Introduction 3](#_Toc95065577)  [Method 5](#_Toc95065578)  [Conclusion and Future prospects 10](#_Toc95065579) |  | |



|  |  |  |
| --- | --- | --- |
| Introduction |  | |
| This project focuses on an AI model that can differentiate two different protein families based on the occurrence of amino acids in the protein sequences. The two proteins that will be studied in this assignment are of the non-ribosomal peptide synthase (NRPS) family and of the polyketide synthase (PKS) family.  Proteins are some of the basic building blocks of living organisms. They are in most cases biologically encoded by the DNA. The DNA first gets transcribed into RNA and finally is translated into proteins. Proteins consist out of 20 different amino acids each with specific properties. In bioinformatics, we write down the amino acids in a one-letter code. An example would be the amino acid valine that has the one-letter code V. An overview of all basic amino acids with their letter is given in Table 1.  Table 1: an overview of the 20 basic amino acids with their one-letter abbreviation   |  |  | | --- | --- | | **Amino acid** | **Single letter abbreviation** | | **Alanine** | **A** | | **Arginine** | **R** | | **Asparagine** | **N** | | **Aspartic acid** | **D** | | **Cysteine** | **C** | | **Glutamine** | **Q** | | **Glutamic acid** | **E** | | **Glycine** | **G** | | **Histidine** | **H** | | **Isoleucine** | **I** | | **Leucine** | **L** | | **Lysine** | **K** | | **Methionine** | **M** | | **Phenylalanine** | **F** | | **Proline** | **P** | | **Serine** | **S** | | **Threonine** | **T** | | **Tryptophan** | **W** | | **Tyrosine** | **Y** | | **Valine** | **V** |   Table 1: an overview of the 20 basic amino acids with their one-letter abbreviation   |  |  | | --- | --- | | **Amino acid** | **Single letter abbreviation** | | **Alanine** | **A** | | **Arginine** | **R** | | **Asparagine** | **N** | | **Aspartic acid** | **D** | | **Cysteine** | **C** | | **Glutamine** | **Q** | | **Glutamic acid** | **E** | | **Glycine** | **G** | | **Histidine** | **H** | | **Isoleucine** | **I** | | **Leucine** | **L** | | **Lysine** | **K** | | **Methionine** | **M** | | **Phenylalanine** | **F** | | **Proline** | **P** | | **Serine** | **S** | | **Threonine** | **T** | | **Tryptophan** | **W** | | **Tyrosine** | **Y** | | **Valine** | **V** |   The two protein families that will be used here are synthases. Meaning that they synthetize a specific molecule. PKS enzymes will produce polyketides. These are a large class of secondary metabolites that are mainly produced in bacteria, fungi and plants but have also been observed in a few animal lineages. Secondary metabolites are molecules that do not have an immediate role in the normal growth of the organism. They often play an important role in the defense against other organisms. A common example of secondary metabolites are toxins.  The NRPS family are a class of peptide secondary metabolites that are usually produced by microorganisms like bacteria or fungi. Normally, to produce a peptide, an mRNA molecule is needed that will be processed by a ribosome. As the name of the NRPS implies, there is no ribosome (or mRNA) required for the NRPS to produce a peptide. The non-ribosomal peptides that get formed by NRPS enzymes often have different properties than the peptides produced by a ribosome. For example, cyclization or specific modifications can occur what is normally not the case for normal peptides.  Both families often consist out of multiple domains. These are regions of the protein that are self-stabilizing and fold independently from the rest of the protein. This is often the case for parts of the protein that have a specific function in the workings of the protein. A 3D structure for one of the proteins of both families is given in Figure 1.  Due to an ongoing research project in the Bioinformatics Knowledge Center (BiKC). We are interested in the different domains that these PKS and NRPS families have and the substrates and products that each of these enzymes have. However, due to limited time this project will first set a baseline for a potential future application that can predict this based on the peptide sequence. Here the goal will be to be able to upload a protein sequence to predict if the sequence belongs to the PKS or NRPS family.  The model will be a basic dense Neural Network that was first trained on a personal laptop and afterwards transferred to Azure Machine Learning Studio to test out multiple hyperparameters so the best model can be picked. The best model will be picked and exported to the personal laptop where it will be integrated with an API (implemented with the FastAPI framework). Finally, it will be converted to a docker and stored on GitHub where it can be used to be deployed on a Kubernetes cluster. | |  |
| Figure 1: 3D structure of one protein from both protein families of interest. Left: a PKS from Streptomyces albus (PDB: 4OPF). Right: a NRPS from an unnames Streptomyces specie (PDB: 6LTA) | |  |

|  |  |  |
| --- | --- | --- |
| Method |  | |
| Data retrieval  The data was downloaded from NCBI using a custom script called download\_sequences.py. All relevant scripts/documents can be found on GitHub through the following URL: <https://github.com/CedricHermansBIT/assignment_MLOps-Home>. NCBI is one of the largest and most important databases for biological data. It provides an API which we can access using URLs. There are some modules available that should provide us with easy access, however they do not provide all the options available to us if we would use the URLs directly. For that purpose, a custom script was made which allows us to download protein sequences regarding a specific keyword. Furthermore, the sequences were filtered so only Streptomyces proteins would be returned that belong to the RefSeq section of NCBI. This means that the sequences have gone through some form of quality control. We also do not allow partial sequences. The API requires us to first search for the IDs that are linked to our search term whereafter we can use these IDs to fetch the actual sequence.  The sequences are returned in FASTA format, the most common sequence format in bioinformatics. This means we always have a header line starting with the “>” symbol that contains information about the sequence (like organism or the name of the protein). The following lines contain the actual sequence data. An example of a part of such FASTA file is shown below.  >WP\_003946496.1 SDR family NAD(P)-dependent oxidoreductase, partial [Streptomyces albidoflavus]  EGADPAPCRGRTAPSPATADTTARAGALTRAVLEVPGVGDAVVLPGPDGEPATVYVVPNRAGAADRTEQV  VSSLAPGTRVVAISGLPRTAEGGLDEGALKDLPVIDQVAAGAWRERLARLPGVREAEVVLEEVPEELERR  HVGRPRAAGGAAEPDAPSVERPASVPALSEGPALPEPSVSGWAEALLRAAGRPDGEVVHVRADGSETRRS  YASLVPEASRVLAGLRRRGLRPGDRVILQCDDTEDFVATLWGCVLGGFVAVPLTVPVSYATTSAAVSKLE  Due to the sheer number of files that were used, these are not included in the repository. There are some sample files provided to test the API later on.  Each of the sequences was processed by first removing the header lines of the FASTA and then counting the occurrence of each amino acid in the sequence. This results in a single list of 20 values that can be used as input values for the model. An example of the results of the partial FASTA from above is given below. Note that the amino acid list is for your information, and normally not really show/saved.  ['A','R','N','D','C','E','Q','G','H','I','L','K','M','F','P','S','T','W','Y','V']  [ 44, 28, 1, 15, 3, 24, 3, 29, 2, 3, 25, 2, 0, 2, 27, 16, 15, 3, 3, 35]  Model structure and training (using Azure Machine Learning Studio)  Before starting to use the Azure Machine Learning Studio, the model was first constructed and tested on a local laptop (since I actually really dislike the AMLS because it’s so slow. I wanted to punch my monitor a few times, but I managed to not actually do it). The model that was used is a simple Dense Neural Network with three hidden layers each containing twenty neurons with relu activation functions. The input layer consists out of twenty values as well, each representing the count of one of the twenty amino acids in the peptide sequence. The output consists out of two output neurons each representing one of the two protein families. The activation function used here was sigmoid. The model was evaluated using the categorical crossentropy loss function. The structure of the model is shown in Figure 2.  Using the local machine, the adam optimizer was used, but this was switched out for an SGD optimizer on the AMLS. An additional change of the model on the AMLS was the addition of different callbacks. First off, a callback for saving the best model was added. Next, early stopping was implemented and finally reducing the learning rate on a plateau was added. The files used to use the AMLS were adapted from the files that were used for one of the assignments.    Figure 2: representation of the structure of the neural network used in this assignment. The opacity of the edges represents a fictional value for the weights and is not representative of the actual used model.  As the local machine was mainly used as a proof of concept, this won’t be discussed anymore in the further parts.  Data upload  The data was uploaded the AMLS using the interface. A blob storage was used for the FASTA files. For the NRPS family, 3224 sequences were used. These were uploaded individually. For the PKS family, 16413 sequences were used. However, due to crashing and slowing down of the web browser (and frustration because of this) it was decided that for PKS, the sequences would be “cat” to a single file that was uploaded to the dataset. This meant that the data preprocessing will have to be adapted for both families. The resulting datasets are shown in Figure 3.    Figure 3: The datasets that were obtained after uploading the data.  Data processing  For the data processing on the AMLS, the notebook 01\_DataPreparing.ipynb was used. The source code is available in the GitHub repository that was mentioned earlier. The sequences were processed as mentioned earlier, and the lists that were gotten from the processes were stored in a single numpy array for each gene family (so one numpy array, containing multiple processed NRPS sequences and one for the PKS sequences). These arrays were stored using the `numpy.savetxt` function. Additionally, another array was constructed for the target of these sequences. If the sequence was an NRPS, the target was [1 0]. For PKS this was [0 1]. In hindside this could have been performed later on.  The saved numpy arrays were uploaded to the default datastore under the processed\_genes folder. A view of this processed\_genes folder is shown in Figure 4.    Figure 4: The processed\_genes folder on the Azure storage.  It was chosen to not yet split up the data into a training and test set. This was performed in the second script.  Training the network  To train the neural network using AMLS, the 02\_Training.ipynb was used. In this script a train.py script is constructed. This will be the script that is actually used when training. This is a modified version of the one received for earlier assignments. The script can accept multiple arguments from the command line, which is used to test multiple parameters in parallel. The first argument is the data-folder, which is a mount to the training data. The other parameters are hyperparameters for the neural network. In this case, the epochs, initial learning rate and batch size could be specified. The script also splits up the data into a train and test set. The actual model is defined and created by a utils.py script.  The model was trained on different settings, for epochs, the possibilities were 25, 50, 75 and 100. For the initial learning rate they were 0.3, 0.1, 0.05 and 0.01. Finally, the size of the batches was changed between 16, 32, 64 and 128. All possible combinations of these networks were tested using triple for loops, I don’t know if there is a possibility to do this using a grid search, and due to limited time, I had no time to look it up or test things. The for-loop works fine, so implementing a grid-search or not is basically bike-shedding. This resulted in a total of 64 models that were tested. Four compute cluster nodes were used for the training of the models.  To monitor the training of the models, the parameters ‘val\_los’ and ‘val\_accuracy’ were logged, which allows us to view these metrics in nice plots in the experiments part of the AMLS. Based on these plots, the best scoring model could be determined which then could be used to download this model for further usage in a local FastAPI. The ‘val\_accuracy’ plot is shown in Figure 5. This model ran over a maximum of 100 epochs, with an initial learning rate of 0.05 and a batch\_size of 16.    Figure 5: 'val\_accuracy' plot from the experiments page in AMLS. The best scoring model is tender\_milk with an accuracy score of 94.17%  The model could be downloaded through the experiments tab, selecting the specific model, and navigating to the Output+logs tab. By pressing the download all button, the model could be retrieved. This model is also stored on GitHub under assignment\_MLOps-Home/FastAPI\_and\_Docker/api/best\_model. We can reload the model on our local machine using the function tensorflow.keras.models.load\_model(“location\_of\_best\_model/name\_of\_saved\_model”). This would allow us to easily switch out the model with a different one.  FastAPI  Having this model, a FastAPI application was build. The source code of this application can be found under assignment\_MLOps-Home/FastAPI\_and\_Docker/api/app. This application loads in our AI model and gives us an API (run through uvicorn) to perform predictions of the protein family based on one (or more) unknown protein sequences.  The application will first process the input. Three endpoints were created for the input. A first one allows us to insert a single protein sequence directly (in FASTA format or raw sequence format (without the FASTA header)). Due to limitations with JSON format (and time to find an alternate solution) it is not possible to send text through the API with actual new lines. These need to be modified in \n (or removed). For this reason, a new endpoint was created that allows us to upload files that can contain just new line characters. This endpoint can however only accept a file with a single raw sequence or a single FASTA sequence.  To solve this problem, another endpoint was created. This one can only accept (multi)FASTA files. In a multiFASTA file, multiple FASTA sequences are just put together in a single file. Every “>” signifies a new sequence. Some additional preprocessing needs to be done. This endpoint returns the type of protein family as a JSON file. Each key in this JSON is the identifier from the header line of the sequence. Instead of returning the entire header, it was changed into the first part of the header, until a space is reached.  Some special additions to basic FastAPI setup that was discussed in the lectures was performed. An HTTPException was added to the multiFASTA endpoint to signify if the user uploaded a text file that is not in FASTA format (first character should always be “>”). This was also added to the documentation page.  The default string input only allows for a small textbox. For this, a custom class was created named TextIn. This gives us a large textarea box. The sequence needs to be provided in JSON-format, where the key is “sequence”. Additionally, an example sequence is provided, so this can be used directly in the docs.  For the file uploads, some things needed to be adapted, first of, we need to use a “put” request instead of “get”. To use files, a new module named “python-multipart” needed to be installed.  Finally, an optional mode parameter was added to the endpoints. This is of the Enum type and has three possible values. If no mode is provided, the default mode is the same as the “chance” mode. The different modes with example outputs are discussed below:   * Chance   + This mode returns the chance for both protein families. The first value is the chance of being an NRPS, the second is the chance of being a PKS.   + Example output: "[[0.7375801 0.25832415]]" * Specific   + This mode returns the name of the gene family. Being “NRPS” or “PKS”.   + Example output: "NRPS" * Fuzzy   + This mode returns a fuzzy output containing the name of the gene family. The output is split up into three categories depending on the certainty of the result. For each category, 4 sentences are defined that are picked at random. This is mainly for entertainment purposes.   + Example output: "Whoah, don't go too hard on me now! I think it's a NRPS"   A video about the use of the API is available through the following link: <https://howest.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=7f16ce1c-5574-4eb5-8d2b-ae340111bd65>  Docker  When the application was complete, a docker image was made for this application. The process was quite similar to what we have seen and tried in the assignment. Some small modifications were needed to also add in our AI model. A first one was to copy our best model to the docker container. A second one was to update our requirements file with the necessary modules for the AI model. In this case, tensorflow, numpy and python-multipart were added besides the fastapi module. The uvicorn server was run on port 5000. The specific docker files were added to the repository as well.  For testing purposes, docker-compose was used. When everything was working okay, the model was built with the docker build command and the “nrps-pks-predictor” tag. This was then pushed to my personal github container registry. The image is private so including a link to it would not be helpful. | |  |

Kubernetes

The final thing that was performed in this project was the deployment on a Kubernetes server. For this, the Kubernetes server of MCT was used. A namespace (“hermans-cedric-assignment”) was created, and a deployment was added called nrps-pks-predictor. This pulls the docker container that was pushed to my ghcr.io and deploys the application. The default Kubernetes deployment config was used with a specific label for the deployment. One thing that was different was the settings of the Resources tab, where it filled in the CPU limit with the suggested value. This is needed for Horizontal Pod Autoscaling which will be discussed later.

Additionally, a service was created for the deployment which allows us to access our API without having need of the specific pods they refer to. Here the selector was configured to only look at the deployment we want using the label we defined earlier. The listening port was set to 80 and the Target Port was set to 5000 (as we defined in the docker file). We can now forward this port 80 to our actual laptop using the kubectl command given below. In this case we forward it to port 8080.

kubectl port-forward services/svc-nrps-pks-predictor 8080:80 --namespace hermans-cedric-assignment

As a final addition to the project, a Horizontal Pod Autoscaler was set up. This required us, as mentioned before to set a limit to CPU usage. For the rest it is quite easy to set up. You just define a minimum and maximum of number of pods and set the Metric you want to monitor.

|  |  |  |
| --- | --- | --- |
| Conclusion and Future prospects |  | |
| As a general conclusion, this project delivered a useable docker image that contains a FastAPI application that can predict two protein families based on a protein sequence. This model is not the most accurate of models, but it does the job for this assignment as a proof-of-concept. The model suffers a bit from imbalanced data, since there were a lot more PKS sequences in the dataset than NRPS sequences.  In the future it would probably be a good idea to also search for more NRPS sequences than the ones that were used now. Besides this, the way of preprocessing the sequence should probably be changed. This is because the order of the amino acids in proteins is often more important than the actual composition itself. We tested this composition and threw away all of the other information regarding the protein sequence. The final goals of a good model would actually not be to predict which protein family a gene belongs to, but rather be a prediction of domains in the protein and possible substrates that could be processed by these domains. I myself however, would have no idea how to get started on such project. Probably some form of LSTM or GRU model should be used for the sequences. This, however is already a nice proof-of-concept.  Due to time constraints, I was not able to implement the extra part of the assignment regarding automation. Thinking about the possibilites however I see one big opportunity for this project. Since I have written a script for the automatic downloading of the PKS and NRPS sequences, it would be great to have the script running every few days to fing new sequences that are available in the NCBI database. If new sequences are found, the AI model can be retrained on the (updated) dataset and automatically be redeployed on the kuberetes server with the new model in place. The scripts regarding the azure pipeline would probably need to be modified quite drastically so that the processing of the NRPS and PKS could occur similarly, which was not the case right now.  Thank you for reading this report! | |