

[435] Protein Classifier Prediction/Analysis

Date of Submission:

11/29/2018

Group Name:

RamMiners

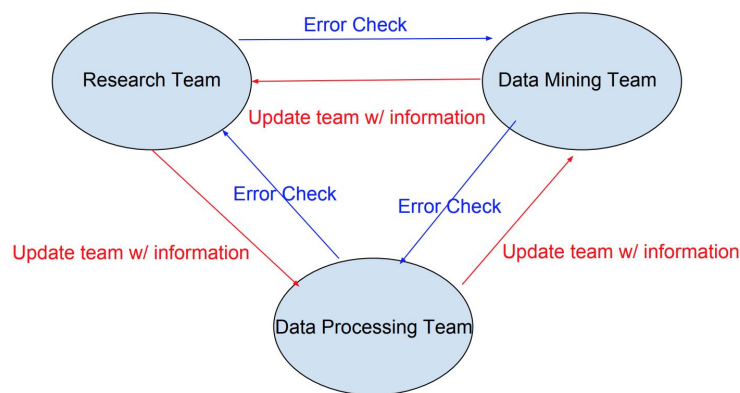


Group Members:

REDACTED

Design of Prediction System Overview:

To help create and design a prediction system for this project where we developed/evaluated/compared models for the prediction of proteins; we decided to take our five-person group and divide the workload. To be efficient, we realized we would need to use our strong suits. The ‘RamMiners’ team made sure that everyone had duties but also would check and balance each other's work. We decided to have three main divisions within our team: Research, Data Processing, and Data Mining. The Research Team was created to learn about classifying proteins, analyzing the data, and making sure that when data processing or data mining was complete that the results made sense. The Data Processing team was created so that training data could be parsed into useful features. These same techniques were used to parse the final test set as well. This was its own team because we wanted to not use 3rd party sites to parse 8795 proteins, and to make sure we knew why and what the features that were getting selected. The Data Mining team was created to make sure we had consistent models, optimize parameters, and to make sure to mine the CSV's that were created by Data Processing. These three teams all worked hand in hand together to double check each other's work. We also met together twice a week, every week since the project was posted. The team would meet and talk through the use of discord on Tuesday/Thursday after Data Science class.



Team Dynamic Model

For our first simple solution (Design 1) the feature selection we tried was the suggestion made to use the amino acid composition of the given sequences. Since we didn't want to rely on the websites provided for finding the compositions we researched how amino acid composition is calculated and created a python script that would transform the original dataset for us. Throughout the data processing effort, the overarching idea was to work with a friendly and high-level language, so that group efforts could be spent developing feature selections rather than learning a language. The language chosen was Python. Python would allow the group to focus on feature selection because of its relative ease of coding and accessibility of outside libraries. This would

help make algorithms more efficient and less time consuming to develop. Python was also chosen because of its widespread use in the data science field. Due to its use in the field, the group was able to leverage several implemented algorithms for feature selection with minimum changes. As the project proceeded the initial goal was to process the raw data into a form RapidMiner could analyze.

The original dataset had to be taken from its raw sequence and class format to a form that RapidMiner could easily take as input. A module was developed which took the simple text file in as input and built a 2D array. This array was composed of twenty features and one class. The twenty features were comprised of the twenty standard genetic coding amino acids. The class represented either DNA, DRNA, RNA, and nonDRNA. The algorithm used applied a simple exclusion rule, which discarded any amino acid abbreviations which did not exist among the standard twenty genetic coding amino acids. Once the excluded letters were discarded, a count was taken for each letter included in the sequence and this count was averaged over the total letters in the sequence. Each feature now had a percentage that represented that particular amino acid's presence in the sequence. This 2D array was processed for all 8795 pieces of data and then output to a file in CSV format. This dataset provided a good reference point for the group to measure other feature selection methods against moving forward.

After deciding on the features (20), we went into RapidMiner to see what kind of results we could get. We went with Neural Network to start out with since it's relatively simple in terms of parameters (used default) to finish our first solution. Lastly, to finish our pipeline we created an excel calculator to find the various values we needed for the final table so we could quickly check our results by just copying the confusion matrix into the calculator. Our average MCC for this Design 1 was 0.294.

The second design (**Design 2**) was based on our first design, but this time around we decided we wanted to optimize this design. With the 'Optimize Parameters' operator, from RapidMiner, we decided to test all possible 121 combinations of Neural Net.learning_rate, and Neural Net.momentum. We also set out training cycles to 1000, the Neural Net.learning_rate ended up being 0.1, the Neural Net.momentum ended up being 0.0, the Neural Net operator was normalized, error epsilon was equal to 1.0E-4, and shuffle was on because our training dataset was sorted by CLASS. After these parameters were set we reran our model, and saw an increase in averageMCC. Design 2 went up 0.002 to 0.296 for the averageMCC.

To further the complexity of the features (this ended up being our Best Design) selected we researched into what other researchers tried to use to classify nucleic acids. One issue with this is that some research relied on things that had to be measured, which limited us in our use of the found knowledge. After this research, we decided to test out using pseudo amino acid composition

of the sequences as part of the feature selection. Once again we didn't want to rely on a website for the transformation, so we created a python script that would calculate it for us, and luckily we were able to find the base algorithm for this selection. The algorithm that we used to select these features was able to be leveraged with little effort due to it being written in Python. Methods were leverage from the last module iteration to deliver the raw dataset to the algorithm and also format and write to file the data output by the algorithm. The main issues involved with leveraging this algorithm were resolving deprecated method conflicts surrounding the algorithm being written in a previous version of Python. The algorithm served well as it allowed more insight into what feature selection could do to refine our results. As the group moved forward, different methods to further separate our features were researched. This second .CSV file that we created consisted of 51 features. This design was based on Pseudo Amino Acid Features, Neural Network w/ Second Design Parameters (optimized). This .CSV file with the combination with the model we used for Design 1 and 2 created the best averageMCC value during this project. The averageMCC value was 0.307. This happened to be our best design, but we still ended up doing a lot more research and work to try to bring our values up.

Before we go into Design 3, we will talk about research and other models we tested out prior. There was a lot we tried out and in the end, we didn't find anything that seemed more valuable/reliable compared to Best Design (an example would be Design 3).

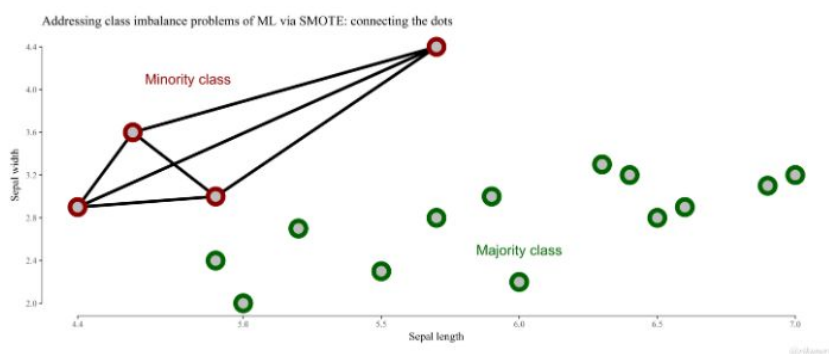
During the research the group discovered hydrophobicity as a feature selection that would possibly yield positive results. Once again the methods for input and output were leveraged from the previous module iterations. Very similar to the first iteration, the algorithm would determine the ratio of amino acids within the sequence. This time, however, three features were chosen: polar, neutral and hydrophobic. Three lists were created which contained amino acids which fell into each of these categories. An exclusion list was also leveraged from the previous iteration to exclude any letter which was not of the twenty standard genetic coding amino acids. If a particular letter in the sequence matched a particular grouping of amino acids comprised in one of the features, that feature tally was incremented. Each feature was averaged over the total amount of letters in the sequence. This process leads to the hydrophobic makeup of each sequence. As the group developed each of these algorithms and pushed them through the pipeline, it became apparent that an algorithm would be needed to join these separate data sets into one or a combination of the datasets.

The goal was to join these different datasets in different configurations to determine if there was an optimum join between the datasets. An algorithm was developed as a stand-alone module such that each successive dataset that was developed could be retroactively joined with older datasets and then compared for optimum results. Input and output were leveraged from previous iterations and a simple process of appending one dataset to another was employed. Each algorithm

that was implemented in this process was built with modularity in mind. The group wanted to ensure that each process which was employed during the development was modular and iterative. Progress could then be determined and code could be leveraged in subsequent iterations without diminishing the ability to return to a previous version or utilize a previous version in its original state. When running the hydrophobicity join dataset on RapidMiner the CSV contained 24 features and produced an average MCC value of 0.291.

Once we began working with the dataset we began to understand it more and more. We started off with thousands of amino acid sequences and that was translated into features selected by composition. Even as we continued to work and change the model we noticed one thing. Even if our model improved, it never predicted DRNA at all. Once we started to look behind the curtain and try to understand why we thought that this may be caused by how many data objects classified as DRNA were in the dataset. Since there were only 22 data objects of DRNA even if the values were accurate the classification algorithm simply did not have enough information on DRNA to be able to predict it at all, let alone correctly. We were thinking of ways to get past this problem, one of the possible avenues were just getting more values. But how would we do this with what we already had? The only way was over-sampling.

Over-sampling can be very helpful when there's many more of one class than others in a dataset. The disparity in our particular dataset was quite big with 7,859 values of nonDRNA and, again, only 22 values of DRNA. One algorithm that does this is called SMOTE (Synthetic Minority Oversampling Technique).



http://rikunert.com/SMOTE_explained

This image illustrates the way SMOTE works. Taken from the link above, it shows the way SMOTE draws lines between points of a certain class and creates new data objects along these lines. It allows us to have a more specific class that, before, didn't have any values hardly, but now allows classification to predict better if a data object should be a part of this class. There are a couple things to consider about SMOTE or just over-sampling in general. If the class that is being over-sampled has a lot of noise or isn't exactly indicative data objects that should be predicted as such, then it makes our model less accurate.

Another subject we researched was sequence similarity searching. Which is a widely used and reliable strategy for characterizing newly determined sequences. Sequence similarity searches can identify “homologous” proteins or genes by detecting excess similarity that reflects common ancestry.

One of the most used tools for this process is the Basic Local Alignment Search Tool from the National Institute of Health (BLAST). BLAST finds regions of local similarity between sequences. The program compares nucleotide or protein sequences to a landmark sequence databases and calculates the statistical significance of matches. BLAST can be used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families.

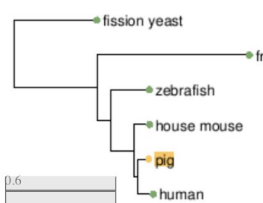
Below is an example BLAST search for a certain protein provided in training dataset.


SMARTBLAST >> Formatting Results - ZNEG8E4G011 [Home](#) [Help](#)


Summary [Report description](#)


Query: unnamed protein product Query length: 148 aa Identical to: [XP_003483058.1](#)


DOMAIN: Ribosomal L27 protein Ribosomal_L27





putative mitochondrial ribosomal ... 

mitochondrial ribosomal protein L ... 

39S ribosomal protein L27, mitoc ... 

39S ribosomal protein L27, mitoc ... 

Your query: 39S ribosomal prot ... 


39S ribosomal protein L27, mitoc ... 






[About the database](#) [See full multiple alignment](#) [Legend](#)

Descriptions

Best hits

Select: [All](#) [None](#) Selected:0

Alignments [GenPept](#) 

	Description	Max score	Total score	Query cover	E value	Ident	Accession
	39S ribosomal protein L27, mitochondrial [Homo sapiens]	268	268	100%	9e-93	87%	NP_057588.1
	39S ribosomal protein L27, mitochondrial [Mus musculus]	261	261	100%	7e-90	86%	NP_444391.1
	39S ribosomal protein L27, mitochondrial [Danio rerio]	182	182	99%	9e-59	62%	NP_001018516.1
	mitochondrial ribosomal protein L27, isoform A [Drosophila melanogaster]	88.2	88.2	89%	6e-22	40%	NP_787971.1
	putative mitochondrial ribosomal protein subunit L27 [Schizosaccharomyces pombe]	72.0	72.0	62%	2e-15	42%	NP_596173.1

BLAST, however, does not provide binding property of any given protein sequence. Thus, we had to develop our own system that carries out a similar sequence analysis to that of BLAST. As in it will directly look into the sequence itself than the features it may represent, against the provided training data, which we used as our landmark binding database.

This procedure uses two modules: unique-sequence module and sequence-similarity module. The unique sequence module is designed to go through the provided test dataset to identify all protein specific unique sequences present in each protein category. Sequence similarity module, on the other hand, will use SequenceMatcher class from python's difflib library to compare the provided sequence against all available training sequence dataset and pick the matching classification. These modules, especially when used in conjunction, produced good classification for given unknown sequences.

However, the provided training data (or better described as previous knowledge/database for this procedure) is limited compared to vast landmark database used by other tools such as BLAST. This makes classification prediction of new datasets with unknown patterns and sequences harder, as well as possibly making the produced results unreliable. In addition, the SequenceMatcher uses Ratcliff-Obershelp algorithm. This algorithm is a cubic time in the worst case and quadratic time in the expected case, which makes the entire procedure very costly time-wise. Therefore, although the produced classification prediction over the provided sequences is acceptably accurate, we have decided to proceed with other designs.

Still, it is important to remember that this may be used to provide additional and perhaps even determinable prediction when other prediction designs fail to produce classification. Therefore, it would be best to somehow incorporate these modules to handle edge cases.

```

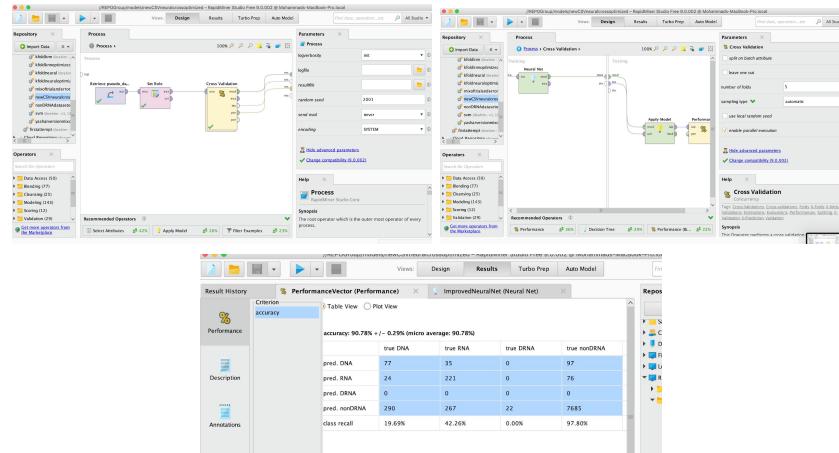
170 # The correlation function is given by the Eq. 3
171 def theta_RiRj(Ri, Rj):
172     return (
173         (H1[Rj] - H1[Ri])**2 + (H2[Rj] - H2[Ri])**2 + (M[Rj] - M[Ri])**2) / 3
174
175
176 # Sequence order effect (Eq. 2)
177 def sum_theta_val(seq_len, LVal, n):
178     sum_theta_RiRj = 0
179     i = 0
180     while i < (seq_len - LVal):
181         sum_theta_RiRj += theta_RiRj(seq[i], seq[i + n])
182         i += 1
183     return sum_theta_RiRj / (seq_len - n)

```

Excerpt of Parser Code for Design 3; deals with sequence order/correlation function.

The third design (**Design 3**) was to use our Pseudo Amino Acid Dataset that was created for our best design and use smote to oversample. The reason for doing this was to hopefully find

DRNA. When we worked with SMOTE to over-sample our data objects, we oversampled all classes that were not nonDRNA to have so all classes had the same amount of samples. Although it did increase our MCC for DRNA and DNA, it decreased the MCC of RNA and nonDRNA by a large amount leaving the average MCC lower at 0.249. Based off these results it seems the features we have selected haven't properly differentiated the classes from each other. Since this experimentation wasn't started until later into the project we didn't have time to look into other oversampling techniques.



Design 3: RapidMiner Model

Results:

Table 1. Summary of results based on the 5-fold cross-validation on the training dataset.

Outcome	Quality measure	Baseline result	Design 1	Design 2	Design 3	Best Design
DNA	<i>Sensitivity</i>	8.9	6.9	10.7	45.5	19.7
	<i>Specificity</i>	99.9	99.7	99.4	89.6	98.4
	<i>PredictiveACC</i>	95.6	95.6	95.4	87.7	94.9
	MCC	0.255	0.174	0.200	0.224	0.245
RNA	<i>Sensitivity</i>	44.1	44.0	44.6	58.9	42.3
	<i>Specificity</i>	99.0	98.8	98.6	91.4	98.8
	<i>PredictiveACC</i>	96.0	95.6	95.4	89.5	95.4
	MCC	0.549	0.536	0.524	0.371	0.518
DRNA	<i>Sensitivity</i>	0.0	0.0	0.0	18.2	0
	<i>Specificity</i>	100.0	100.0	100.0	98.0	100.0
	<i>PredictiveACC</i>	99.7	99.7	99.7	97.8	99.7
	MCC	0	0	0	0.058	0
nonDRNA	<i>Sensitivity</i>	99.1	99.0	98.4	79.6	97.8
	<i>Specificity</i>	28.7	31.8	34.2	68.8	38.1
	<i>PredictiveACC</i>	91.6	91.8	91.6	78.5	91.4
	MCC	0.443	0.467	0.460	0.342	0.466
averageMCC		0.312	0.294	0.296	0.249	0.307
accuracy		91.4	91.3	91.1	76.7	90.8

Design 1 - Amino Acid Composition Features, Neural Network w/ Default Parameters

Design 2 - Amino Acid Composition Features, Neural Network w/ Optimized Parameters

Design 3 - Join of Pseudo Amino Acid with Hydrophobic Dataset, Oversampled All Four Classes, w/ Optimized (second design) Parameters

Best Design - Pseudo Amino Acid Features, Neural Network w/ Second Design Parameters

Best Design Confusion Matrix					
	true DNA	true RNA	true DRNA	true nonDRNA	class precision
pred. DNA	77	35	0	97	36.84%
pred. RNA	24	221	0	76	68.85%
pred. DRNA	0	0	0	0	0.00%
pred. nonDRNA	290	267	22	7685	92.99%
class recall	19.69%	42.26%	0.00%	97.80%	

Prediction Summary	
Class	Percentage
DNA	1.46%
RNA	5.24%
DRNA	0.00%
nonDRNA	93.30%

When we had access to the test dataset, our team worked quickly to convert the un-classified dataset to something we could use. Once we turned the list of amino acid sequences into 51 features we gave it to RapidMiner, but unfortunately, RapidMiner could not give us the performance since this dataset is unlabeled. We thought quickly, and manually export the CSV with the predicted classes for the test dataset. With the CSV, we created a program in java that looked at the CSV and did what RapidMiner couldn't. The algorithm got the percentage metrics of the predictions our model made.

Conclusion:

The quality of the results was somewhat disappointing since a lot of work was put into research and experimentation for feature and parameter selection. Compared to the quality measure our best design doesn't really have any great reason for existing, as the quality measure is for the most part superior in all aspects. The one aspect that the best design has an advantage in is the specificity for nonDRNA that led to the MCC being higher.

Overall the goals of the team were met during the development process which allowed for progress to be realized and duplication of efforts to be minimized, all while maximizing the efficiency of each group member's contribution. It was also nice that we quickly developed a pipeline from the unprocessed DNA sequences to the average MCC calculation which allowed quicker testing of different ideas. While the research and the experimentation of parameters were definitely good, the experimentation of different methods beyond using neural network was a little lacking. More research into the value of the features selected would have also been good since the number of features we had quickly become restrictive in the amount of time it took to run the models.

Although we decided that this model was the best, it doesn't mean that it doesn't have its own drawbacks. The main bottleneck of the pseudo model is that it doesn't predict any DRNA amino acids. Out of all of our models the only model that predicted DRNA is the oversampled dataset model. The problem with that though is that it made the other classes, particularly RNA and nonDRNA worse. For that reason, we decided to sacrifice predicting DRNA for the reliability and consistency of the pseudo model.