Mark Gorelik
CS 123B
Profile HMM for Splice Junctions
Dr. Heller

Profile HMM for Splice Junction Classification

By Mark Gorelik

Table of contents

Problem Definition

The splice junction data set from the UC Irvine Machine Learning Repository defines the problem as “Splice junctions are points on a DNA sequence at which `superfluous' DNA is removed during the process of protein creation in higher organisms. The problem posed in this dataset is to recognize, given a sequence of DNA, the boundaries between exons and introns. This problem consists of two subtasks: recognizing exon/intron boundaries and recognizing intron/exon boundaries” (1). In this project, we explore the possibility of solving this through a profile HMM implemented in Python 3.6.

Background

Introns/Exons

When a gene is transcribed from DNA to mRNA it is not directly translated from mRNA. Instead, the transcription product, known as pre-mRNA is processed into mRNA (4). mRNA goes on to be translated into a peptide sequence. A key step of this processing is removing portions of the nucleotide sequence from the pre-mRNA. These sequences which are removed are known as introns - the removal is known as splicing and the boundaries for this removal are known as splice junctions. The remaining sequences are known as exons and are translated (1,4). As can be seen in Figure 1 the final mRNA is shorter than the initial pre-mRNA which raised the question why does a cellular system expend the energy of writing the extra bases and then snipping them out? It turns out there are two broad functions of introns. The first is alternative splicing which is where different combinations of exons are present in the final mRNA - this allows for a single gene to encode for multiple products or variants of a product(4). The second function of splicing is to generate noncoding RNA (ncRNA). The ncRNA's which be derived from introns include miRNAs, siRNAs, and long noncoding RNAs (lncRNAs) (4). miRNAs and siRNAs both are involved in gene expression by targeting and binding specific nucleotide sequences inhibiting their expression a.k.a as silencing (6). lncRNAs participate in epigenetic regulation however not all the functionality is fully elucidated (6). Understanding the boundaries of introns and exons allows for the ability to quickly define introns and exons and further classify them automatically enabling a greater an understanding epigenetic regulatory mechanisms in eukaryotes. Greater domain knowledge of siRNA and miRNA could further applications of synthetically designed siRNA and miRNA.

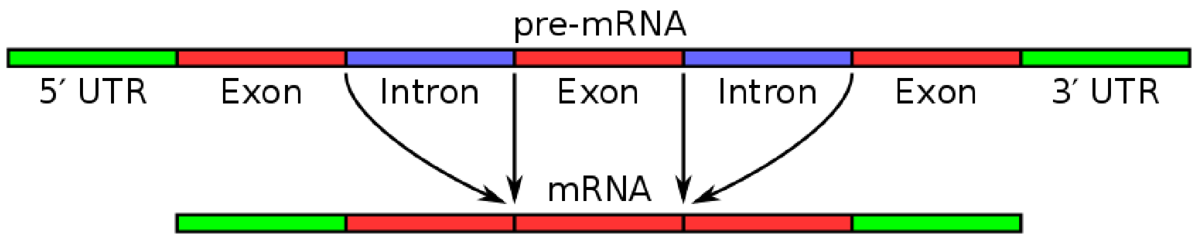


Figure 1. Example of pre-mRNA to mRNA processing with intron removal (3)

Profile HMMs

As described in lecture, profile HMMs (pHMMs) are an implementation of HMMs which are used to classify sequences of interest into various classes. The HMM is best described as a table of probabilities where each column is a character position in the sequence and each row holds an emission probability. They are commonly used for peptide and nucleotide sequence classification. While time-tested they need a good data set to make meaningful classifications that are reflective of the real world. This means pHMMs (and most if not all machine learning methods) are limited by the quality of the data they are trained on. In addition to this the training data set must be a reasonable size or else the data risks being non-representative and for very small data sets of less than 100 samples you run the risk of struggling to accurately manipulate pseudo-counts. For example, in a dataset of 100 points, a pseudo-count of 1 is .01 in terms of probability. However, in a data set of 3000, you are presented with a much greater control of the impact of the pseudo-counts. In this case, a pseudo-count of 1 is .00033 in terms of probability and provides a much more fine-grained control. You can work around this with pseudo-probabilities but that doesn’t change the limitations of a small dataset.

Data Structure

The dataset is 3005 samples. It is a TSV, the first column contains the sample class, second the sample source, third the sample sequence. The classes were used to define the HMM states. The classes present are N (Neither/None), EI (Exon-Intron Boundary), IE (Intron-Exon Boundary). There are 8 emissions: A,C,T,G,D,N,S,R. A, C, T, G are the canonical DNA bases. The other 4 characters stand for ambiguous bases. D is for G, T, or A. S is for C or G. R is for A or G. N is for A, C, T, or G.

Some important notes to draw from the data is that this data set is for classification this means that sequences presented do not go through state changes. Even though intron splicing occurs in RNA it the dataset presented is in DNA bases which means the trained pHMM would be used for DNA based investigations instead of RNA based investigations - though you could just convert RNA samples of interest to DNA sequences by swapping uracils with thymines in the string representations. The IE and EI samples come only from primate gene sequences which limit the applicability of the trained pHMM to only sequences from primates or eukaryotes with similar splice junction patterns. Finally, the N class sequences are not targeted which poses the opposite risk of over-focusing a class. I noticed in my original implementation that the N class seemed to have a high false positive rate but that was alleviated in the pHMM implementation.

Procedure

The code is in a jupyter notebook hosted at github.com/vzg100/Profile-HMM-CS123B in the file ProfileHMM.ipynb for anyone who is interested. The code was written in python 3.6 using only built-in functions. The following will be a high-level discussion of the code and the approach implemented. Actual code will be kept to a minimum for sake of the reader. The core code is 76 lines with another ~20 for testing.

Profile HMM Class

The pHMM class is the bulk of the code. It reads a file by taking a TSV (or CSV) and reading the lines into a dictionary containing each sequence mapped to its class. When doing this all the states and emissions are recorded into sets. After reading the data the user can convert the dictionary into a data structure to represent the pHMM columns and probabilities. This data structure is a list of dictionaries stored in a dictionary mapped to each state. So, for example, the state EI is mapped to a list containing a dictionary for each column position. Inside each of the columns, dictionaries are the frequency of emission occurrence at that column position for that state. A simplified example for a 1 state, 3 column, 2 emission model would like: {“EI”:[{“A”:1, “B”:1}, {“A”:2, “B”:0}, {“A”:0,”B”:2}]}. This would then be converted to a probability table by counting sum of each column and then dividing each element in the column by that sum resulting in something which looks like: {“EI”:[{“A”:.5, “B”:.5}, {“A”:1, “B”:0}, {“A”:0,”B”:1}]}.

While very dense as a data structure this was interesting to implement. It gave me the chance to play with python comprehensions creating 3-dimensional comprehension monstrosities which took what would normally be 20 lines of code and boiled it down to a single line. Aside from practicing language idioms this data structure makes it very easy to change components of the table both broadly (to every element) or narrowly (to specific elements).

Viterbi

The Viterbi algorithm implemented allows for both log odds based calculations and non-logodds. Since no state transitions are assumed in the data set the final Viterbi implementation does not include state transitions and simply multiplies (or log sums) the position probabilities and selects the largest. In the original implementation (the file is called HMM.ipynb) the Viterbi algorithm allowed for state transitions to occur in sequences.

Random Sequence Generation

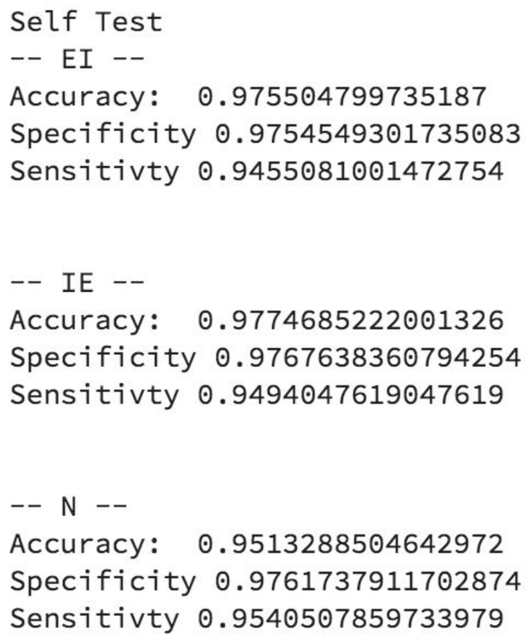
Random sequences were generated by iterating through the respective state table and generating a list of emissions with populations representative of that column and then randomly selecting an emission. After iterating through the entire column the method returns the generated seq and the seed state

Review of Original Implementation

This project went through 2 implementations. The first implementation was not a proper profile HMM but rather an HMM which made a prediction based on previous states in the same vein as the Thor HMM. Because of that the original implementation of Viterbi factored in state changes and would provide state paths for sequences and was much more effective in handling inserts and deletes. However, since the dataset didn't actually provide any examples of state transitions it wasn't feasible train and test around transitions.

Results

Self Test



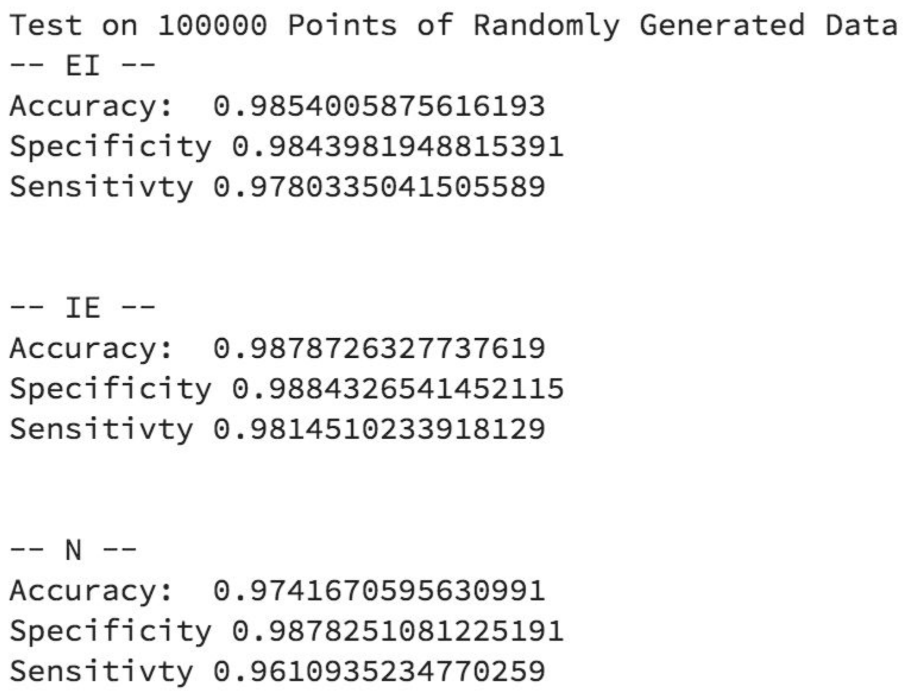
```
Self Test
-- EI --
Accuracy:  0.975504799735187
Specificity 0.9754549301735083
Sensitivty 0.9455081001472754


-- IE --
Accuracy:  0.9774685222001326
Specificity 0.9767638360794254
Sensitivty 0.9494047619047619


-- N --
Accuracy:  0.9513288504642972
Specificity 0.9761737911702874
Sensitivty 0.9540507859733979
```

Figure 2. Self Test Results

Randomly Generated Data



```
Test on 100000 Points of Randomly Generated Data
-- EI --
Accuracy:  0.9854005875616193
Specificity 0.9843981948815391
Sensitivty 0.9780335041505589


-- IE --
Accuracy:  0.9878726327737619
Specificity 0.9884326541452115
Sensitivty 0.9814510233918129


-- N --
Accuracy:  0.9741670595630991
Specificity 0.9878251081225191
Sensitivty 0.9610935234770259
```

Figure 3. Randomly Generated 100,000 Data Points Test Results

Discussion

Solution Efficacy

This implementation was a good first step. While targeted at primate splice junctions it proved to very effective across the board. Each class had a reasonably high accuracy, sensitivity, and specificity. A note on the randomly generated data test - the randomly generated data is a reflection of the data, if the data is nonrepresentative the randomly generated data is nonrepresentative and the randomly generated data test has little real-world significance.

A note on my own abilities as a programmer. My implementation of the pHMM was a really interesting exercise that I used to challenge my coding skills - it resulted in a barebones pHMM in under 80 lines of code. The first time I wrote an HMM my Viterbi implementation alone was 30 lines and while code length isn't a good metric for quality of code it is a good exercise to minimize code. That being said there is a lot of room for improvement which will be discussed in the following sections.

What could be Done Differently?

If I were to do this again I would take full advantage of Numpy, Pandas, and SKlearn to implement n-fold cross-validation and memory efficient data storage. Furthermore, I would have liked to have implemented some additional features such column specific emissions, column-specific pseudo-counts, and emission specific pseudo counts to provide a fine-tuned modeling

What are the Next Steps?

I want to continue this project but overhauling the internal data structure to be compatible with the Sklearn API to take advantage of their functions - specifically those around test/train splits and n-fold cross-validation. Those were surprisingly challenging to implement well without additional libraries. It proved to be a really healthy wake-up call on how reliant I am on the code others have written.

References

1. C.L. Blake and P.M. Murphy. The UCI machine learning repository. [https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+(Splice-junction+Gene+Sequences)]. In Irvine, CA: University of California, Department of Information and Computer Science, 1998.
2. Wheeler, T. J., Clements, J., & Finn, R. D. (2014). Skylign: A tool for creating informative, i
interactive logos representing sequence alignments and profile hidden Markov models. *BMC Bioinformatics,15*(1), 7.
doi:10.1186/1471-2105-15-7
3. Intron. (2018, April 17). Retrieved from https://en.wikipedia.org/wiki/Intron
4. Jo, B., & Choi, S. S. (2015). Introns: The Functional Benefits of Introns in Genomes. *Genomics & Informatics,13*(4),
112. doi:10.5808/gi.2015.13.4.112
5. *Rearick D, Prakash A, McSweeny A, Shepard SS, Fedorova L, Fedorov ANucleic Acids Res. 2011 Mar; 39(6):2357-66.*