Bioinformatics & Beyond Final Project: Detecting Mutations Using Text Classification

By Noah Segal-Gould

Background: Mutations happen frequently as a necessary part of evolution. Techniques for identification of mutations make it possible for biologists and bioinformaticians to efficiently compare genomic data. Using the Python programming language, our class sought to detect mutations of nucleotide sequences using string manipulation techniques. These techniques perform well, but the act of identifying which mutation occured between a sequence of nucleotides and a known mutated one presents an interesting research opportunity for text classification. Using a supervised machine learning technique for classification of text such as Naive Bayes, it should be possible to identify mutations based on those present among the classes of known generated mutations.

Methods: In class, we trained the K-Nearest Neighbors classification algorithm on numerical data associated with categories of warmth to predict the warmth of unseen numerical data. Classification problems typically perform in this fashion: preexisting data is already associated with labels, which together are used to train a classifier in order to predict the best labels to fit other data. For this project, I initially intended to use K-Nearest Neighbors for classifying mutations but instead decided to use the Naive Bayes classification algorithm for its ease of use with text and also its simplicity. Naive Bayes computes the probability of particular classes given each particular feature or word in a text document using Bayes Rule. It is naive because it naively assumes the conditional independence between these features. Since this project uses short nucleotide sequence strings associated with other short nucleotide sequence strings, this assumption should be violated frequently. Even so, I implemented methods in a class to generate all possible insertion, deletion, and point mutations to be used as training data for the Naive Bayes classifier. I used the Scikit Learn machine learning library's implementation of Naive Bayes as well as its `CountVectorizer` for conversion of text data to numerical data based on token counts. The `MutationsClassifier` class generates training data and instantiates a Scikit Learn `pipeline` for easy management of both the classifier and vectorizer in its constructor. The `generate_training_data` method uses the earlier `all_point_mutations`, `all_insertion_mutations`, and `all_deletion_mutations` methods to generate all possible mutations of those types for short nucleotide sequences of lengths 3 to 6 using the `generate_all_sequences_with_length` method. I would have liked to expand this to work on much larger sequences but my implementation is very time consuming. The generated training data follows the format of string and label pairs for which each string contains a space separating the two nucleotide sequences (e.g. `("ACTG ACTA", "point_mutation")`). The `classify` method trains the classifier on this training data and produces predictions given some sequence-to-mutation pair string. Finally, the `cross_validation` method uses K-Folds Cross-Validation to split the training data into 10 testing and training sections, randomly selecting 10% of the entire dataset as testing data and using the remaining 90% to train the classifier in each fold. By splitting apart the dataset in this fashion, I accumulate a confusion matrix of correct and incorrect classifications as a kind of statistical measurement on the performance of the classifier. I use K-Folds in place of a single split into testing and training data sections because every time the confusion matrix is accumulated in each fold, the classifier intentionally excludes 10% of the data so when it is used specifically for testing it is data the classifier has never encountered before. Thus, creating a confusion matrix of categorical scores using K-Folds Cross-Validation makes it possible to visualize the performance of the classifier using all the data available even though each fold excludes 10%. This confusion matrix is plotted using `matplotlib`.

Results: In total, this procedure generated 92,736 point mutations, 98,176 insertion mutations, and 24,544 deletion mutations. The `all_point_mutations`, `all_insertion_mutations`, and `all_deletion_mutations` methods generate lists of sequences. Here are examples of these lists:

```
> MutationsClassifier().all_point_mutations("CGATAG"))
> ['TGATAG', 'CGGTAG', 'CGAAAG', 'CGATTG', 'AGATAG', 'CGACAG', 'CTATAG',
'CGATCG', 'CGATAA', 'CGTTAG', 'CCATAG', 'CGATGG', 'CAATAG', 'GGATAG', 'CGATAC',
'CGAGAG', 'CGCTAG', 'CGATAT']

> MutationsClassifier().all_insertion_mutations("CGATAG"))
> ['CCGATAG', 'CAGATAG', 'GCGATAG', 'CGAATAG', 'CGTATAG', 'CGACTAG', 'CGATAAG',
'CGATACG', 'CGATCAG', 'ACGATAG', 'CGATTAG', 'CGATAGG', 'CGCATAG', 'CGGATAG',
'CTGATAG', 'CGAGTAG', 'TCGATAG', 'CGATGAG', 'CGATATG']

> MutationsClassifier().all_deletion_mutations("CGATAG"))
> ['GATAG', 'CATAG', 'CGATG', 'CGTAG', 'CGAAG', 'CGATA']
```

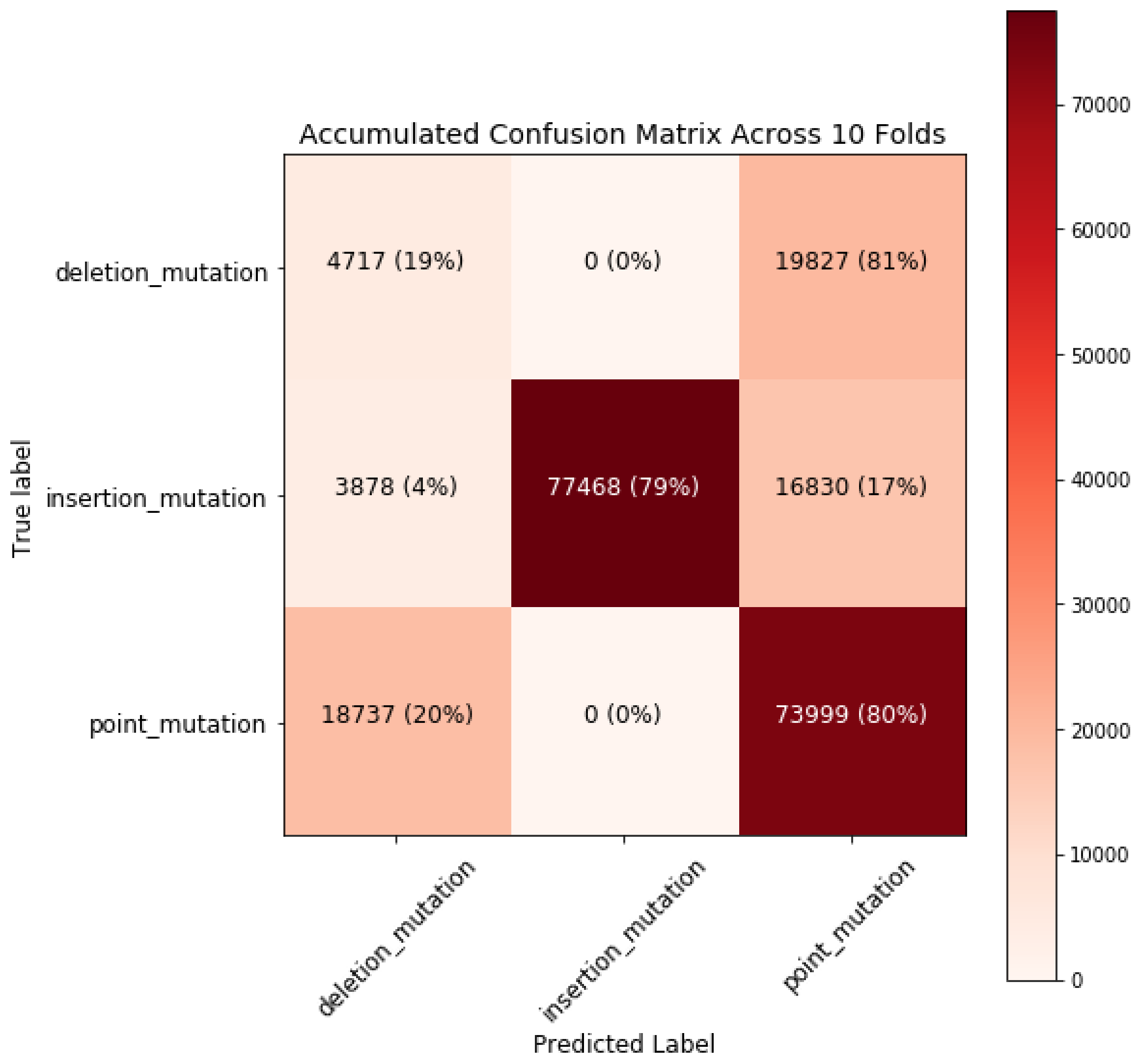
Given the following list of test sequences:

```
tests = ["GGGA TGGA",         # Should be a point mutation
         "GGAT TGAT",         # Should be a point mutation
         "CATTAC CATCAC",     # Should be a point mutation
         "GGA GG",            # Should be a deletion mutation
         "CGAT CGA",          # Should be a deletion mutation
         "CGATAG CGTAG",      # Should be a deletion mutation
         "CGTCAA CGTCAAA",    # Should be an insertion mutation
         "GAT GGAT",          # Should be an insertion mutation
         "CCCC CCTCC"]        # Should be an insertion mutation
```

The predicted classifications are frequently incorrect, overclassifying pairs as deletion mutations:

```
{'GGGA TGGA': 'deletion_mutation',
 'GGAT TGAT': 'deletion_mutation',
 'CATTAC CATCAC': 'point_mutation',
 'GGA GG': 'deletion_mutation',
 'CGAT CGA': 'deletion_mutation',
 'CGATAG CGTAG': 'point_mutation',
 'CGTCAA CGTCAAA': 'insertion_mutation',
 'GAT GGAT': 'deletion_mutation',
 'CCCC CCTCC': 'deletion_mutation'}
```

Finally, the accumulated confusion matrix across all 10 folds:



As shown above, actual deletion mutations are incorrectly classified as predicted point mutations 81% of the time, whereas actual deletion mutations are correctly classified as predicted deletion mutations only 19% of the time.

Conclusion: This Naive Bayes approach for classifying mutations using numerical counts of sequence tokens as vectors performs only somewhat well. The conditional independence assumption should be violated consistently in this approach, but insertion mutations and point mutations are still classified correctly respectively 79% and 80% of the time. Meanwhile, deletion mutations are very poorly classified in comparison, which I theorize is because there are around 70,000 fewer sequences in that dataset than either of the other two. In future work, I would like to apply similar text classification techniques for detection of silent, nonsense, and missense mutations as well and also use significantly larger nucleotide sequences.

Acknowledgements: I worked alone on this project and used code from `sklearn` to complete it.