Using Codon and Amino Acid Frequencies to Predict Taxonomic Group and DNA Type

Introduction

The DNA of organisms is structured at a low level into codons, three-base sequences that encode amino acids. The four nucleotides (adenine, cytosine, guanine, and thymine) produce a total of $4^3 = 64$ codons, which encode a total of 20 amino acids, and a stop codon. There is a great deal of redundancy here, with each amino acid being represented by between one and six codons.

It has been established that the taxonomic identity and type of DNA can be predicted with high accuracy from a simple frequency analysis of codons in a sample of DNA, using a variety of machine learning algorithms.[1] In the research establishing this, Random Forest and Extreme Gradient Boosting algorithms were assessed as preferable overall. The ability to rapidly draw inferences from only the frequencies of codons in a sample can make assessment of new sequences comparatively computationally simple, with applications throughout the world of biological research. This project was intended, first, to replicate the findings of Dr. Khomtchouk in the paper referenced, and to see if this principle extends to the specific amino acids that the codons encode.

To accomplish this, I obtained the data used in Dr. Khomtchouk's study, from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Codon+usage) – a dataset with 13,028 observations with 69 attributes. In addition to the relative frequencies of all 64 codons, some attributes of the organism – its name, taxonomic group, DNAtype, etc – were also provided. After some basic data wrangling (consisting mostly of resolving type issues) I used the codon frequencies in preprocessing to produce relative frequencies of the amino acids encoded.

Both the codon- and amino-level data were used to train machine learning models of the same types as employed in the referenced study: k-Nearest Neighbors, Random Forest, Extreme Gradient Boosting, Multi-Layer Perceptron, and Naive Bayes. All of these models (with the exception of Naive Bayes) required some degree of hyperparameter tuning; for k-Nearest Neighbors and Random Forest, I attempted this myself. For the others, hyperopt was used for hyperparameter optimization. All five model types were used to predict both DNA type and taxonomic group (referred to in the source data set and, unfortunately, throughout my work somewhat incorrectly as 'kingdom') based on both the original codon frequencies, as well as the amino acid frequencies that were developed from them. It was my expectation that the amino acid frequencies could be only marginally less effective for prediction, and perhaps useful in some circumstances in practice.

After fitting many models, I found that the best method across all four model categories (predicting DNA type and taxonomic group based on codon and amino acid frequencies) was the Multi-Layer Perceptron artificial neural net; best hyperparameters for each model were determined using hyperopt. This produced the best ROC-AUC score in each model category, and the best micro-F1 score in two of the four. An appealing alternative is the k-Nearest Neighbors model, which had the best micro-F1 in the other two model categories, and requires two orders of magnitude less runtime. It was also verified that models using amino acid frequencies as predictors performed nearly as well as those using codon frequencies. Several aspects of the outcome indicate a need for more thorough follow-up study.

Data Wrangling and Preprocessing

The data from the UCI repository were generally clean and well curated. In the entire 13028 x 69 dataset, there were no missing values at all. I did have a bit of trouble placing these data in a pandas dataframe; there was a persistent problem with data types. Categories that were described by abbreviations and numbers were translated to more readable names. One-hot encoding was applied for the sake of the libraries that require it.

I decided to employ a three way split to the data, first creating a test set comprised of 15% of observations to be used to test all models, and then, for each model, producing a training and validation set, again at an 85/15 ratio.

One of the most salient features of these data was noticed by a simple enumeration of value counts: the data set is extremely imbalanced. With respect to the ‘kingdom’ outcome variable, the initial set produced:

```
bacteria         2920
virus            2832
plant            2523
vertebrate       2077
invertebrate     1345
mammal            572
bacteriophage     220
rodent            215
primate           180
archaea           126
plasmid            18
Name: Kingdom, dtype: int64
```

And with respect to the ‘DNAtype’ variable, the value counts were:

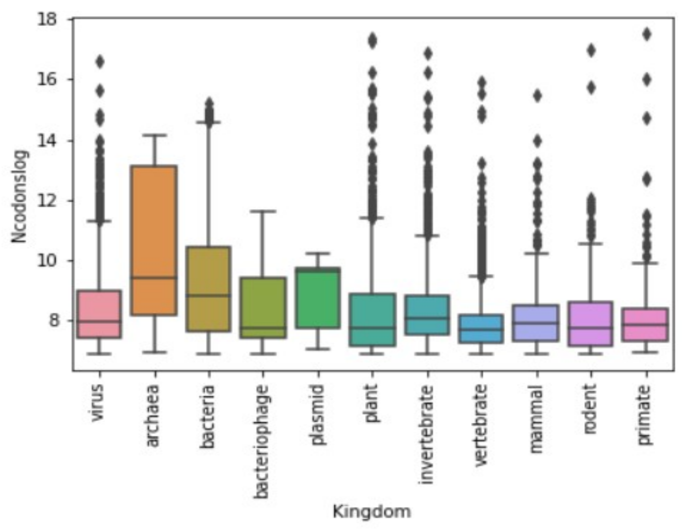
```
genomic                   9267
mitochondrial             2899
chloroplast                816
plastid                     31
kinetoplast                  5
cyanelle                     2
nucleomorph                  2
apicoplast                   2
secondary_endosymbiont       1
chromoplast                  1
Name: DNAtype, dtype: int64
```

One concern with such unbalanced data is that the comparatively trivial number of cases in some categories can exert significant leverage on the model as a whole. In the case of “kingdom” I noted that the distribution tapers off somewhat more slowly, and that identifying many of the more sparse groups could be of substantial interest; I decided to omit the 18 observations of plasmids. The “DNAtype” variable, however, had a far more unbalanced distribution, and I decided to omit plastids and every category with a lower count. In total 62 rows were dropped, leaving 12,966 observations. There was a single observation for which the categorization was somewhat anomalous; this observation, too, was discarded.

It could be of interest to subsequently fit models that include these categories to determine just how much effect these observations would have.

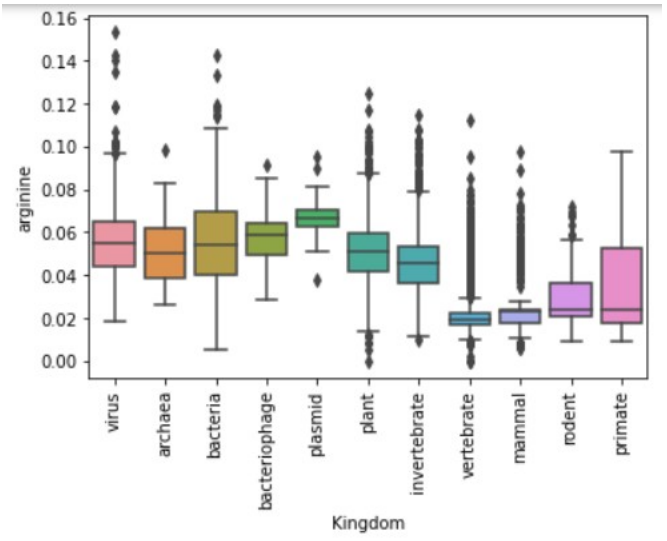
Exploratory Data Analysis

I began my graphical EDA with an exploration of perhaps the most simple and accessible predictor provided: the number of codons for each observation. Reflected in boxplots by taxonomic category – which required a log transformation to be particularly informative – the number of codons yields:



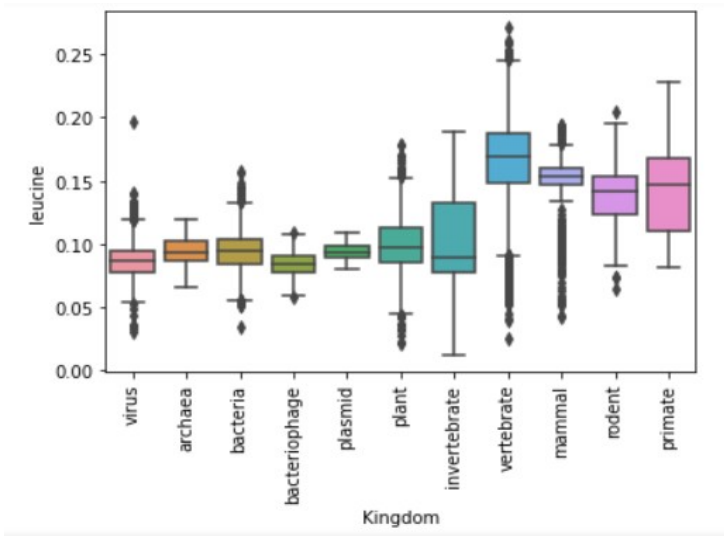
This demonstrates a reasonable degree of variation among the ‘kingdom’ groups with respect to this basic feature – something I was looking to validate. I proceeded to examine the distributions for various amino acids across these categories as well. That these showed significant variation among the categories – with respect to the center, variation, and skewness of the distributions – for all amino acids gave me a reasonable assurance that these would be of some predictive value.

For instance, the boxplots for arginine:



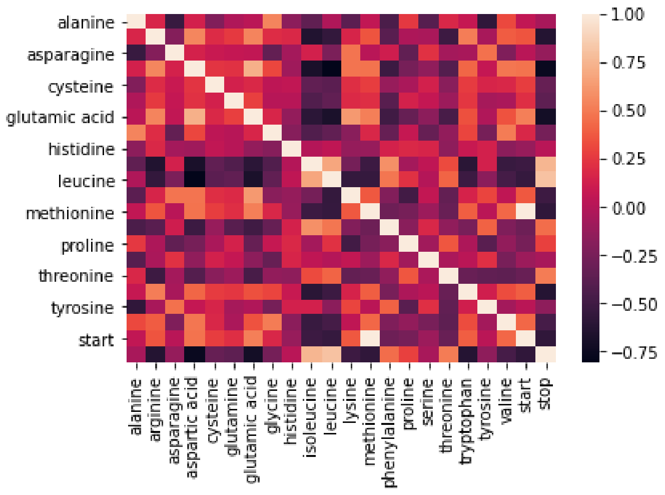
It’s interesting to note here that the median (the middle bar in each ‘box’) is very similar in the vertebrate, mammal, rodent, and primate groups, but the dispersions are quite different. A similar pattern is seen with several other amino acids in this dataset.

For another example, the boxplots for leucine:



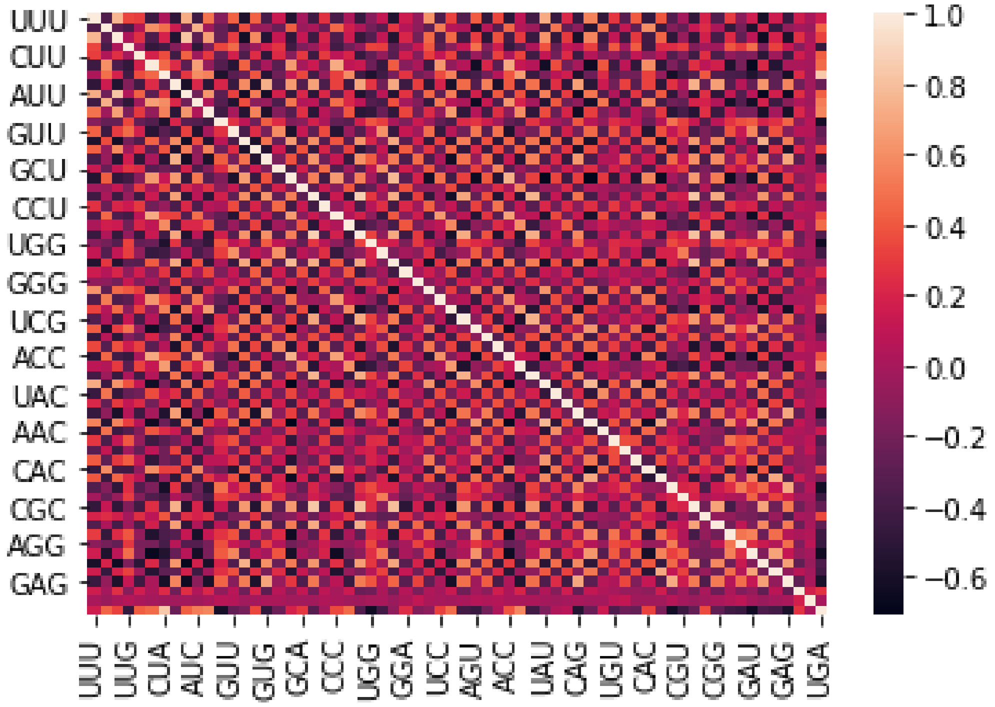
As in the preceding example, the distributions demonstrate significant differences among the categories to be predicted. It's interesting to note – and perhaps meaningful – that we can see patterns among taxonomic groups that are related. For instance, 'bacteriophage' is a subset of 'virus', and we can see that the median bars for the two groups in the boxplot above are almost exactly equal, yet the 'virus' group has somewhat more variation, and some more extreme outliers on both sides of the distribution.

I thought it might also be useful to visualize the cross-correlation matrices for the codon and amino acid data. For the aminos, the matrix is somewhat easier to read:



What can be gleaned from this? One prominent feature is the high correlation between methionine and the start codon, which is unsurprising, as the start codon codes for methionine. Another interesting point is the moderately strong correlation between isoleucine and leucine, which are (as the names suggest) isomers. Both of them have a strong correlation with the stop codon – does this suggest that they are more common among shorter genes?

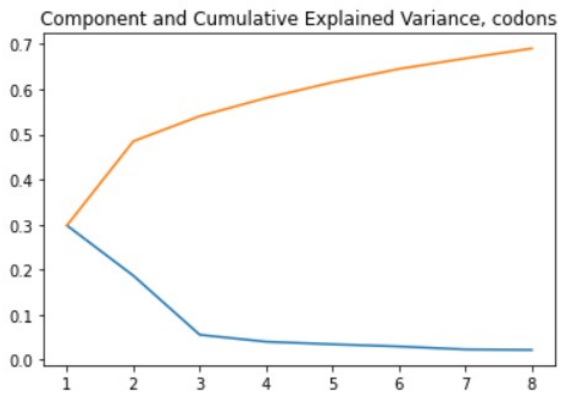
The same plot produced for the codons is difficult; there are simply too many features being compared:

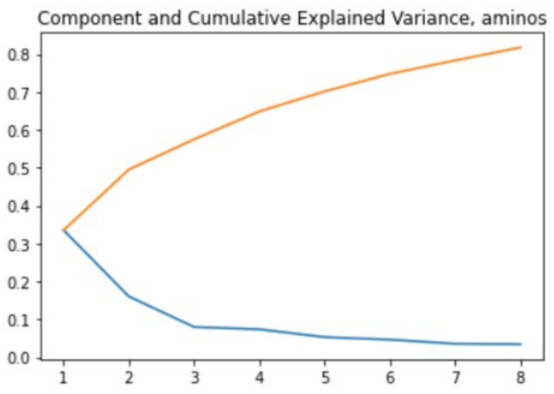


It’s a challenge to even read the index; what can be learned from this plot? One thing that can be noticed is that, despite the general redness (most correlations here are weak) we do see the plot speckled with occasional strong correlations, both positive and negative. The negative correlations are, perhaps, easy to explain: they largely represent other codons that code for the same amino acids. The handful of prominent strongly positively correlated variables are more of a challenge. Another striking feature here is the ‘checkerboard’ pattern, which is largely a result of the ordering of the data.

Principal Component Analysis

Before fitting any predictive machine learning model, I applied Principal Component Analysis, using both the codon and the amino acid frequencies. In both cases, there was a distinct ‘elbow’ at three principal components:



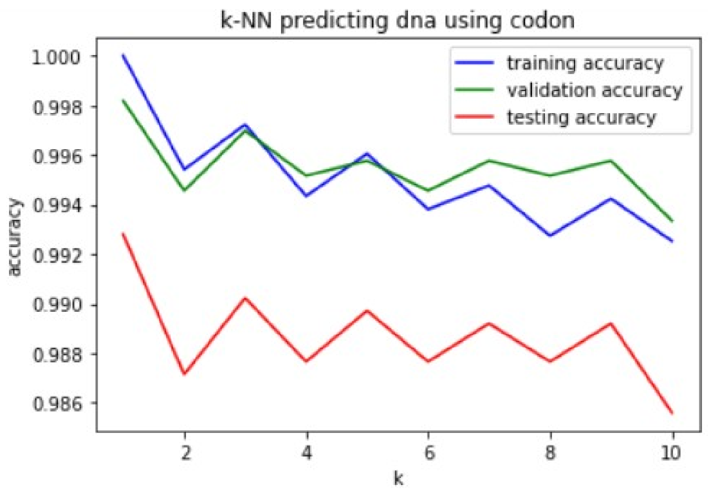


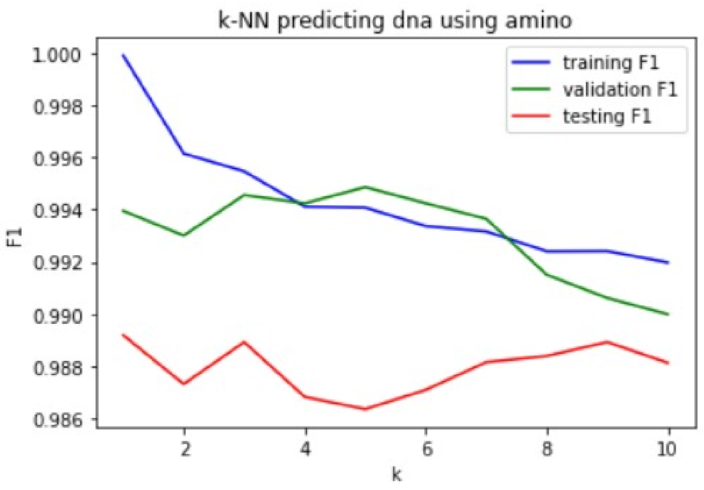
Interestingly, the PCA graphs show that using the amino acids produced more effective principal components than the codons themselves. But in both cases, the cumulative explained variance remained too low for effective modeling; I obtained 69.02% and 81.69% with eight principal components. I decided to abandon using principal components and proceed using the original codon and amino acid frequencies to construct models.

k-Nearest Neighbors

The first machine learning model that was applied – and one of the simplest – was the k-Nearest Neighbors (k-NN) model. K-NN classification is a simple and venerable[2] ML algorithm that places an observation in the same category as the plurality of its closest neighbors with respect to the predictor variables. This makes it, of course, a nonparametric method, and what is known as a 'lazy learner' – no real 'model parameters' exist, the decision is made simply by computing distances against the known data. This makes k-NN computationally difficult for high-dimensional data, and somewhat vulnerable to unbalanced data, such as those for this study. K-NN was implemented using the KneighborsClassifier in sklearn.

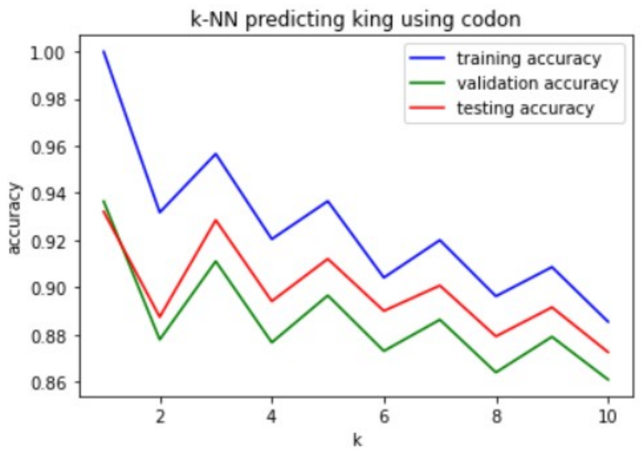
Having only limited experience in hyperparameter optimization – and with the only hyperparameter for this model being k, the number of neighbors – I decided to optimize k manually. This produced some results that are difficult to reconcile. For instance, many of the graphs demonstrated *higher* accuracy, F1, and AUC for the testing and validation sets than for the training set:

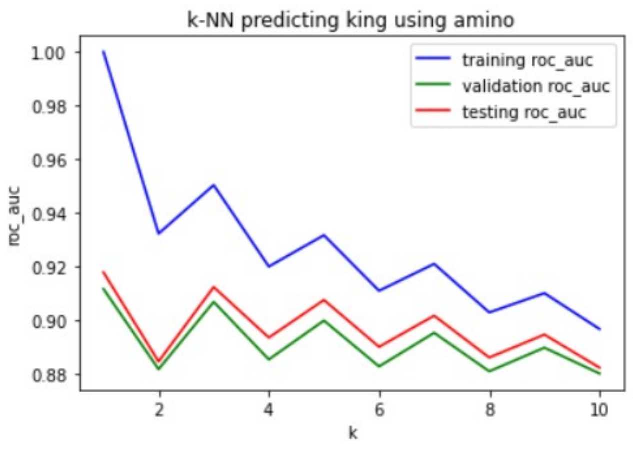




This is a little troubling. It was my general impression that, while this is certainly *possible*, in general it indicates either some kind of problem with the data split. Alternatively, it could emerge randomly with an underfit model. Having examined and tested the train-test-validation split exhaustively, I'm fairly certain this is indeed due to underfitting.

Another feature common (but not universal) among the k-NN assessment graphs was a "sawtooth" pattern among all data sets, with metrics spiking up on odd values of k:





Upon a bit of investigation, it seems clear that this is an artifact of the k-NN algorithm itself. With even values of k, observations close to a boundary risk being, in essence, a "tie" – which, likely, is exaggerated by the high number of features. At any rate, it seems to be part of the conventional wisdom that odd values of k tend to perform better for this reason.

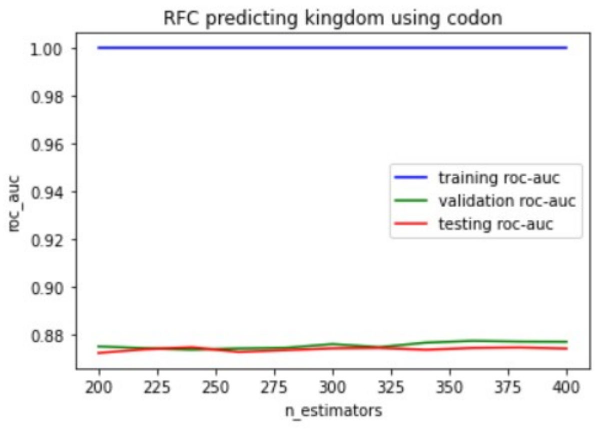
The k-NN models were ultimately evaluated (to be consistent with the other models in the study) using micro F1 scores on the test set. The results were:

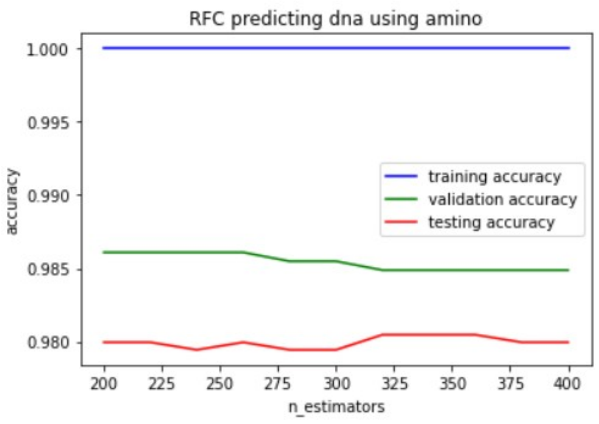
predicting	features	neighbors k	test_f1	test_accuracy	test_roc_auc	run_time
DNAtype	codon	1	0.9928	0.9928	0.9918	3.48 seconds
DNAtype	amino	1	0.9892	0.9892	0.9874	3.41 seconds
Kingdom	codon	3	0.9384	0.9285	0.9593	5.07 seconds
Kingdom	amino	3	0.8696	0.8468	0.9124	5.09 seconds

Notably, the k-NN algorithm worked substantially better at predicting DNAtype as opposed to kingdom. It's also worth noting that – particularly in comparison to some of the other ML methods employed – k-NN is extremely fast. Another interesting observation is that in the 2020 study by Khomtchouk, the optimal value of k was found to be 1 for predicting kingdom, and 3 for predicting DNA type. It's interesting that my findings are precisely reversed; it's possible and really quite likely that this is merely due to chance, but perhaps there's something behind it.

Random Forest

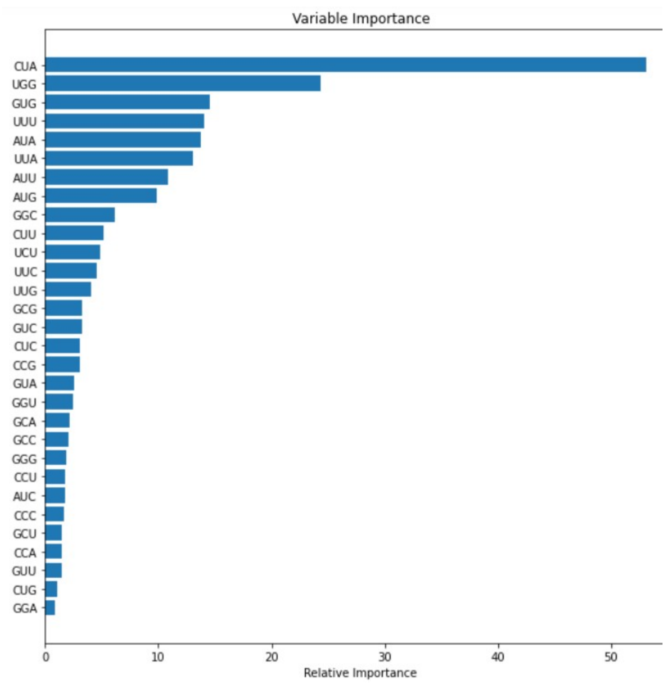
Random Forest models are an ensemble learning method in which a large number of decision trees are trained on the data, with decision by consensus.[3] Random forests are famously designed to protect against the overfitting typical of decision trees. I used the RandomForestClassifier from sklearn for these models. As with k-NN, I began by attempting to optimize over values of the most accessible hyperparameter, "n_estimators" – the number of trees in the forest. These results were, at best, inconclusive; some typical graphs follow.

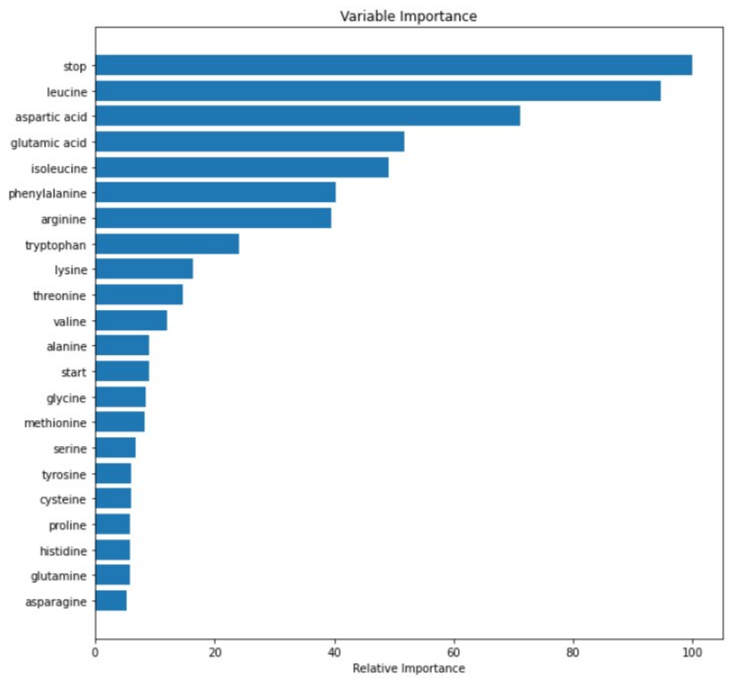




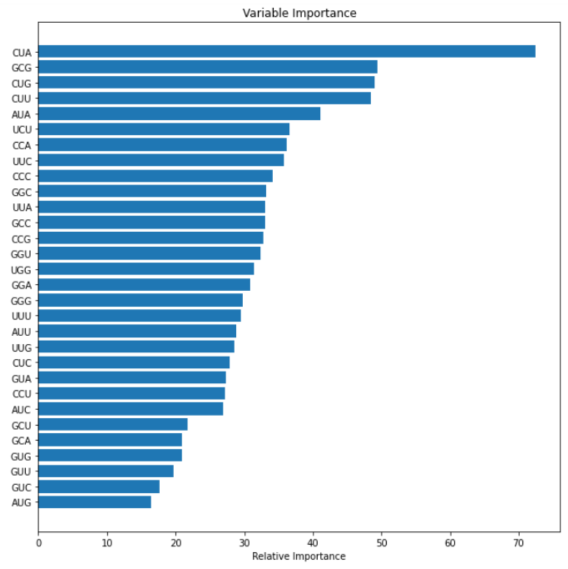
In every instance, accuracy, F1 score, and AUC for the training set were all 100%. Validation and testing accuracy were consistently slightly less, but with little variation and no discernible pattern with respect to values of the underlying hyperparameter. Over the range of values I tested (200 to 400 trees) there seemed to be no substantial variation in model performance. Choosing a model for feature importance, then, was a more or less a matter of indifference, and subsequent figures are based on the RFC defaults.

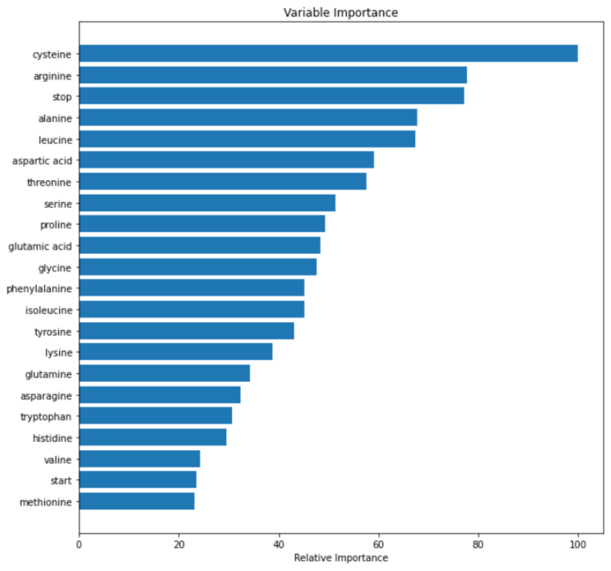
The feature importances property provided by the random forest classifier is based on Gini impurity. With default hyperparameters, the feature importances produced the following graphs. Modeling DNAtype using codons and aminos, respectively:





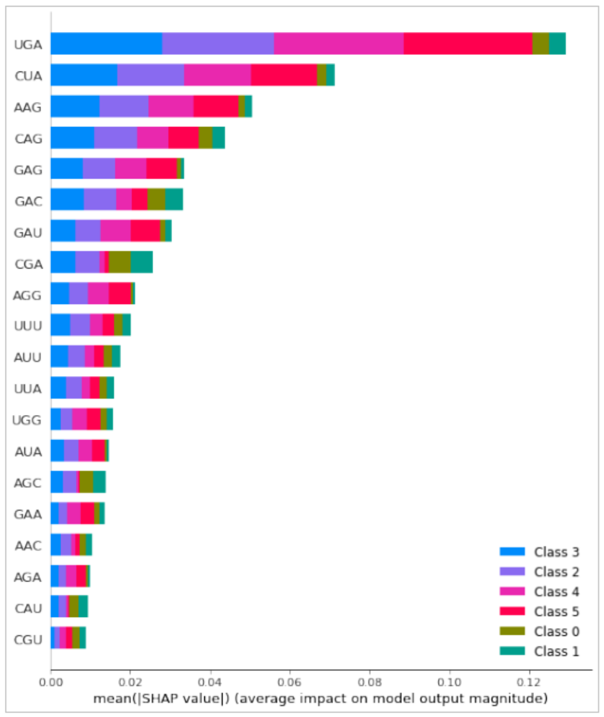
Predicting kingdom using codons and amino acids produced the following feature importance plots:

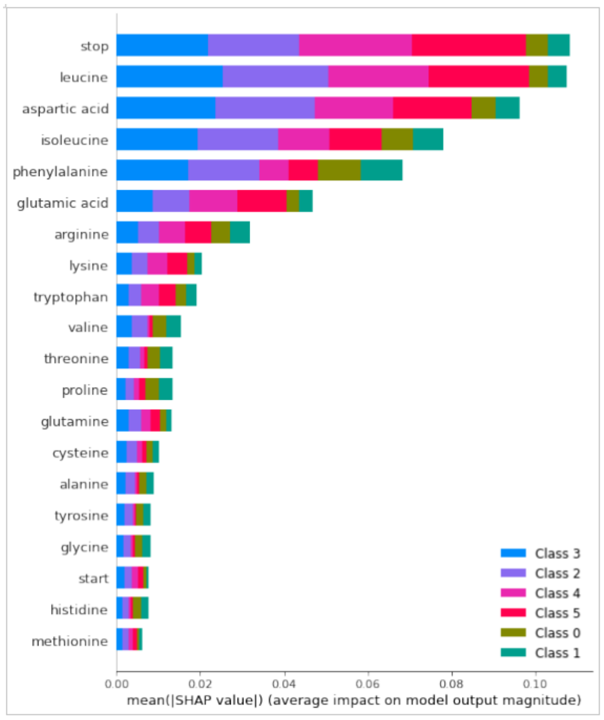




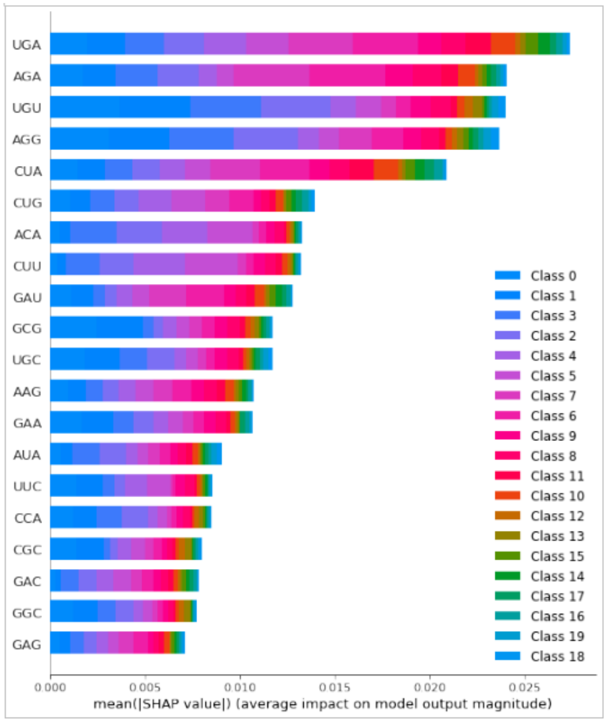
At a superficial glance, it's interesting how changing the predictor variables from codons to amino acids doesn't seem to change the overall *shape* of the feature importance charts, despite reducing the number of variables from 64 to 21. In both cases, the feature importances drop off in roughly the same pattern. This, I found surprising. It's also worth noting that – despite the fact that the two upper graphs pertain to predicting the "DNAtype" and the lower two to the "kingdom" variable – we see many of the same features near the top of the distribution in each case. The CUA codon, for example, is identified as the most important feature for predicting both DNA type and kingdom.

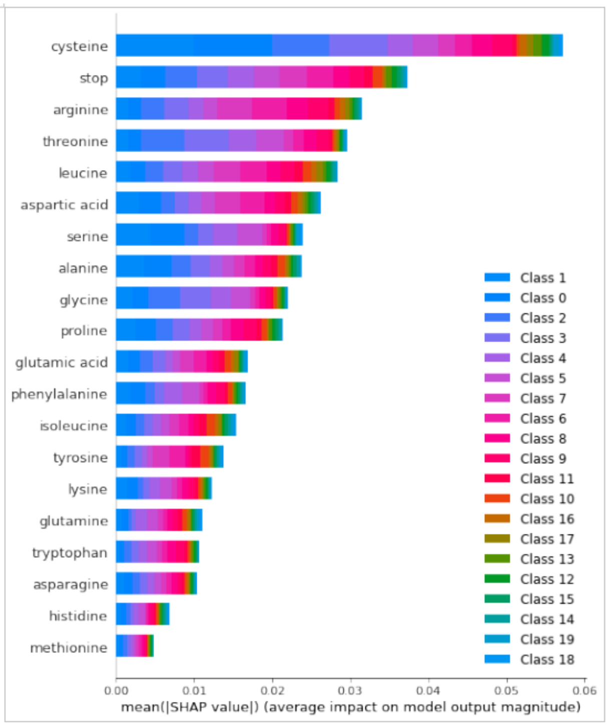
These are worth comparing to the results obtained using SHAP (SHapley Additive exPlanations) values which were also computed for the random forest models. Here are the SHAP summary plots for the models predicting DNA type based on codons and aminos:





For the models predicting kingdom, results for codons and aminos respectively:





It's obvious that the SHAP summary plots are leading to slightly different feature importances than those derived using Gini inequality in the sklearn package. Interestingly, the CUA codon still features prominently, as does the amino acid cysteine. At a cursory glance, we have many of the same features appearing near the top of the respective plots – in any subsequent effort, I will need to make a more systematic comparison of feature importance results. The SHAP and Gini feature importances seem to match substantially more closely for the amino-based models. It's also worth noting that in the SHAP-based analysis, the UGA codon ranks first for models predicting both DNAtype and kingdom.

As with k-NN, the random forest models were optimized manually for the number of estimators. The best results for each model category were:

outputs	inputs	n_estimators	test_f1	test_accuracy	test_roc_auc	run_time
DNAtype	codon	240	0.9892	0.9866	0.9849	3.25 seconds
DNAtype	amino	320	0.9840	0.9805	0.9756	2.90 seconds
DNAtype	amino	340	0.9840	0.9805	0.9756	2.91 seconds
DNAtype	amino	360	0.9840	0.9805	0.9756	2.93 seconds
Kingdom	codon	240	0.8517	0.7532	0.8746	8.73 seconds
Kingdom	amino	200	0.8202	0.7152	0.8537	7.05 seconds

As the table shows (we were expecting only four rows) the F1 scores for the test set were identical for three tested values of n_estimators on the DNAtype-amino model. I decided to take n_estimators = 340 as the optimal value, as it sits in the middle of this range. And again: there was very little difference in test metrics with respect to this hyperparameter. But the 'best' hyperparameter values are as indicated in the above table.

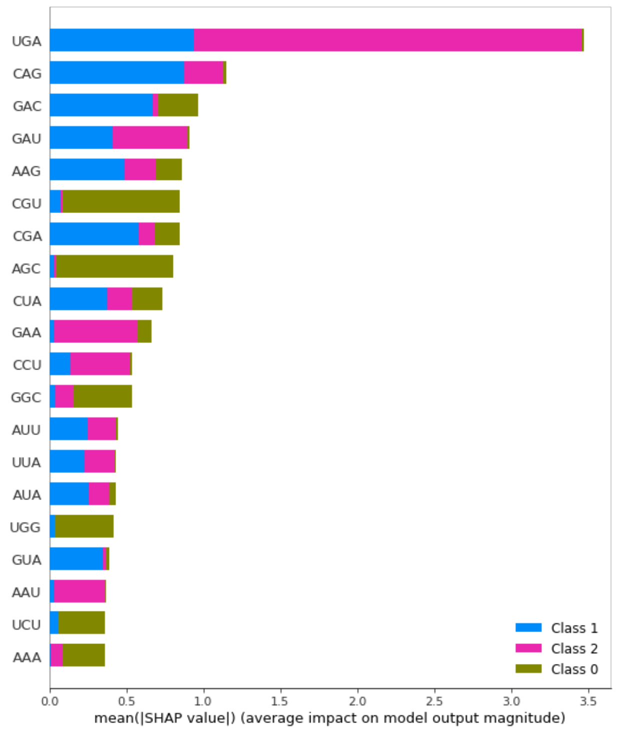
Notably, as with k-NN, the random forest models performed much better for predicting DNA type than kingdom.

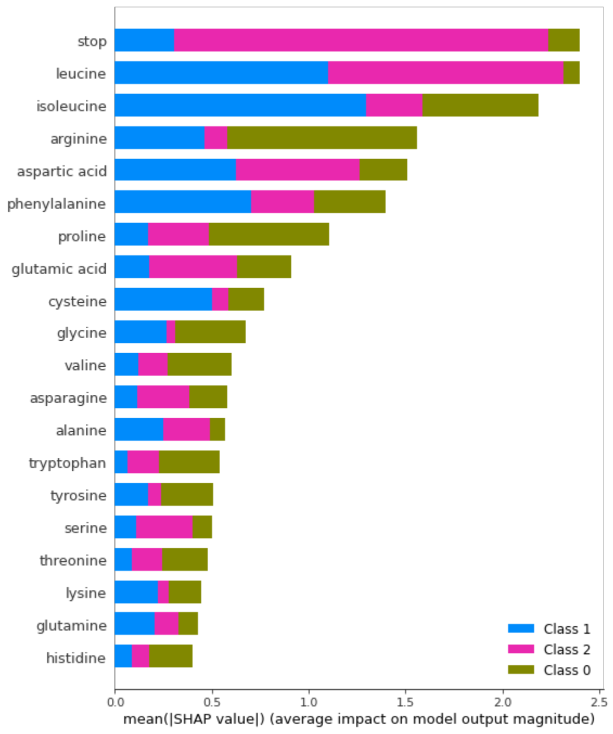
Extreme Gradient Boosting

Extreme Gradient Boosting (XGBoost) is a gradient boosting[4] decision tree library of machine learning algorithms, extremely popular since its development in the mid-2010's, which includes a classifier object.

The XGBClassifier object has a multitude of hyperparameters. Optimizing over all of them was prohibitive with respect to running time. I determined to optimize (using the hyperopt library) over a handful of hyperparameters that I gathered would be of practical importance in fitting a model to sparse data with several outcome categories. These hyperparameters were: 'gamma', which tunes the minimum loss reduction needed to make a new partition on a leaf node; 'reg_alpha' and 'reg_lambda', the L1 and L2 regularization terms for weights; 'colsample_by_tree', the subsample ratio of columns when constructing each tree; and 'min_child_weight', the minimum sum of instance weight needed in a child.

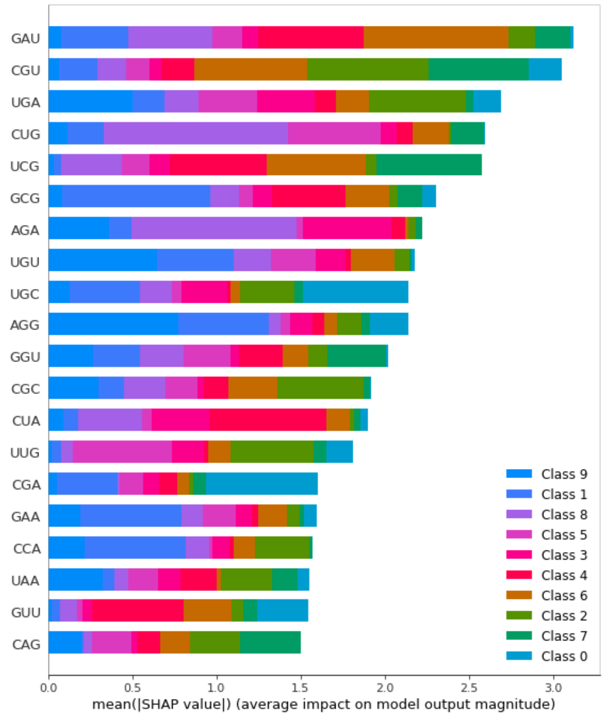
SHAP values were computed from an XGBoost model as well, and used to produce some summary plots for feature importance. For the models predicting DNA type, we have:

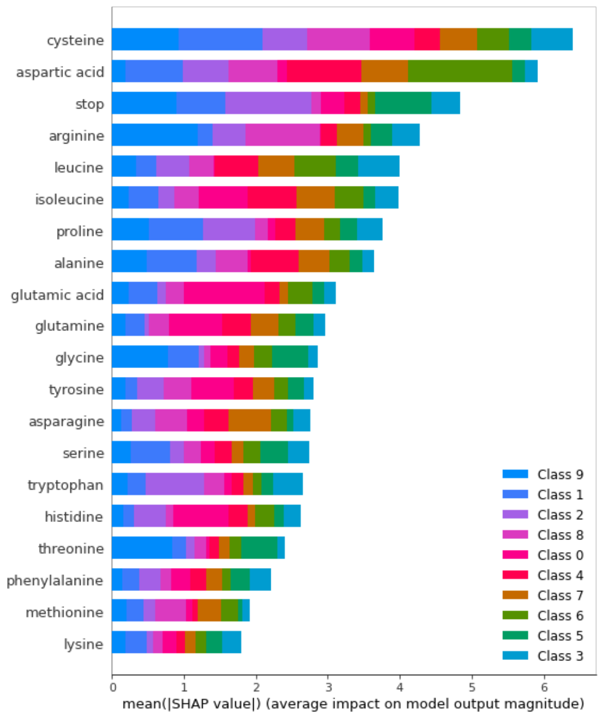




As with the SHAP feature importances developed from random forest models in the preceding section, we see UGA ranking first, and by a substantial margin. While the order isn't identical, we see many of the same features ranked near the top as we did in the preceding case.

The SHAP feature importance plots for the XGBoost models predicting kingdom are:





This, too, is not terribly dissimilar from the SHAP feature importances obtained using random forest. It’s reassuring to see that there’s some consistency in these results. I note now that I neglected to run the built-in (Gini impurity based) feature importance provided with XGBoost; it turns out that it’s also possible to perform permutation based feature importance through sklearn with these models. This might be worth doing, in a subsequent analysis.

The model metrics obtained with XGB present some problems. I began by running the XGBClassifier with its default parameter values, and only subsequently applied hyperopt to some of its more prominent parameters. This revealed the following somewhat disturbing results, and is a good lesson. Especially as the results are more daunting, I should check them as I proceed. Here are the results for the XGB default models, and those optimized using hyperopt:

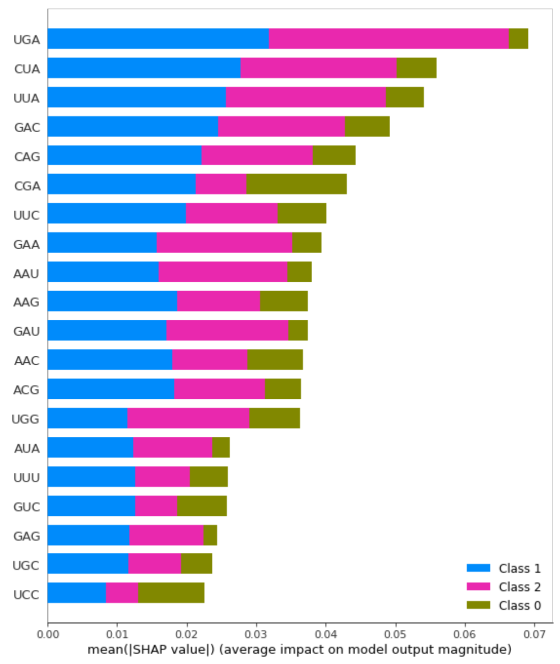
model_type	outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
XGB	dna	codon	0.9779	0.9779	0.9977	1.38 seconds
XGB	dna	amino	0.9671	0.9671	0.9764	0.85 seconds
XGB	kingdom	codon	0.7522	0.7522	0.9413	6.71 seconds
XGB	kingdom	amino	0.7208	0.7208	0.9156	4.26 seconds
XGB-Hyperopt	dna	codon	0.9069	0.9069	0.9900	48.34 seconds
XGB-Hyperopt	dna	amino	0.9661	0.9661	0.9753	23.29 seconds
XGB-Hyperopt	kingdom	codon	0.5810	0.5810	0.9145	161.09 seconds
XGB-Hyperopt	kingdom	amino	0.6159	0.6159	0.8886	161.50 seconds

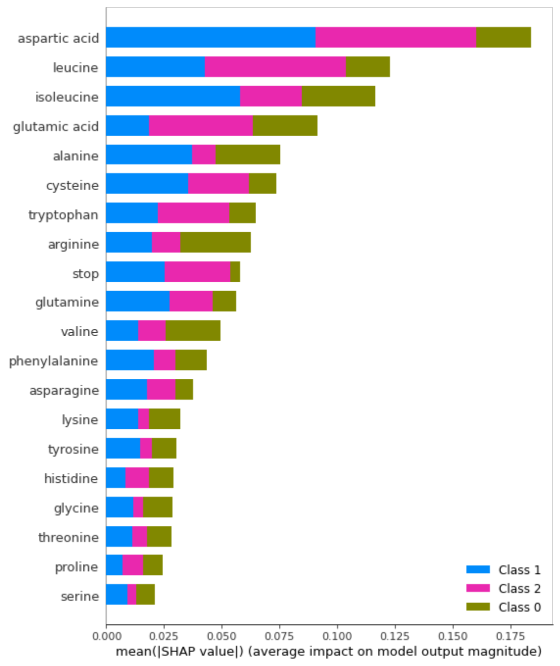
Upon examining these results, I noticed: the performance metrics for *the XGBClassifier defaults* were better than those obtained using hyperopt! (And indeed, they were available for a small fraction of the runtime.) It's not immediately obvious what happened to produce such a result. It would be desirable in a follow-up study to try to optimize again, both with a different search space and perhaps with a different optimizing algorithm. But as things stand, the best XGB models for all four categories are the defaults. These are {'colsample_bytree': 1, 'gamma': 0, 'max_depth': 3, 'min_child_weight': 1, 'reg_alpha': 0, 'reg_lambda': 1} – interestingly, a perusal of the hyperparameters chosen by hyperopt finds them to be widely distributed, but in most cases very different from these defaults. And indeed, the models perform quite differently. This is definitely an important matter on which to follow up. Optimization is supposed to *optimize*.

Multi-Layer Perceptron Classifier

The MLPClassifier from scikit-learn implements a multi-layer perceptron classifier, a variety of feedforward artificial neural net. The MLP is distinguished from a linear perceptron by its non-linear activation function and multiple layers.   This classifier was tuned by applying hyperopt on a space covering the 'hidden_layer_sizes', 'activation', and 'learning_rate_init' parameters, which define the number of layers and neurons in each layer, the activation function, and the rate at which weights are updated. The models were optimized over a wide hyperparameter space that included, crucially, all activation functions available other than the identity function. This contributed substantially to the runtime for hyperopt on these models, but was necessary for a thorough optimization.

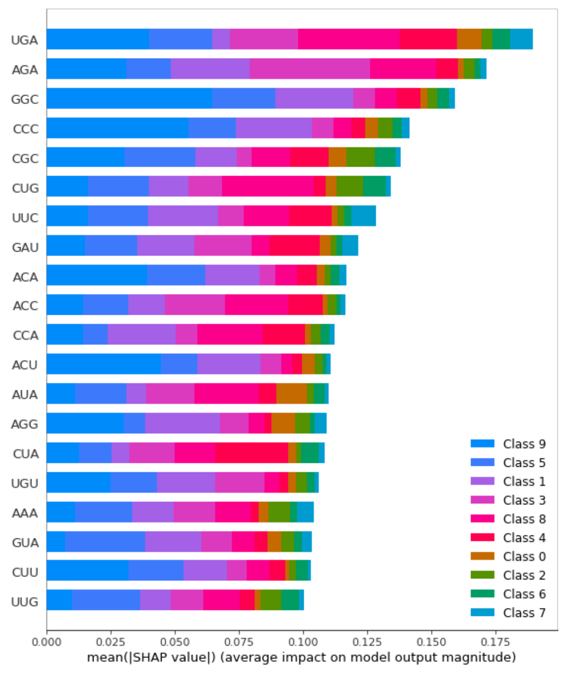
SHAP feature importance plots for the MLPClassifier, for the models predicting DNAtype, were:

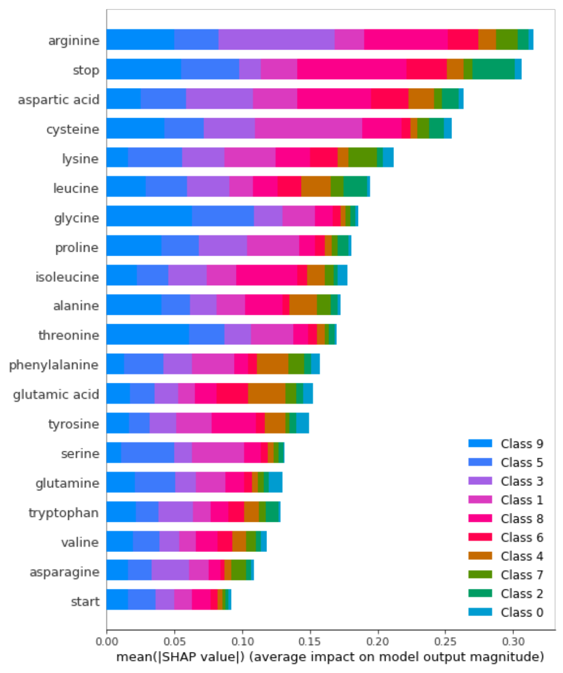




These are largely similar to the SHAP feature importance plots generated in the XGBClassifier models.

For the models predicting kingdom, we have:





There are a lot of familiar names near the top of these charts, but of course the precise order differs. Again, these feature importances should be systematically compared in a follow-up study; there's surely a lot to be discovered there, and what I'm doing here is essentially EDA. I should store these feature importances in a dataframe as I did the model assessment metrics and hyperparameters.

The optimal MLP models as determined using hyperopt were:

outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
dna	codon	0.9933	0.9933	0.9992	278.31 sec
dna	amino	0.9918	0.9918	0.9979	357.45 sec
kingdom	codon	0.9321	0.9321	0.9949	567.74 sec
kingdom	amino	0.8416	0.8416	0.9743	679.36 sec

Optimizing and training the Multi-Layer Perceptron took substantially more runtime than any other model used. However, the results are overall the best. In particular, all four models had a high (>0.95) AUC.

Naive Bayes

The naive Bayes classifier is probably the simplest model tested in this study. The great simplifying assumption – which *we can know in advance to be false in the case of this study* – is that a naive Bayes classifier treats features as independent. (In this case, because the features are relative frequencies and are exhaustive, this assumption clearly fails.)

I decided (in the absence of any prior knowledge that would make another model preferable) to use a

Gaussian Naive Bayes model, which, as the name implies, assumes an underlying normal distribution. Note, however, that this assumption is made *not on the data themselves* – it is instead the assumption made about the distributions of the likelihood functions for the values of the model parameters. I used the GaussianNB instantiation from scikit-learn.

Naive Bayes models have no 'hyperparameters' in the sense of other common machine learning models - GaussianNB has only a parameter for calculation stability and another for an array of prior probabilities. (Providing these prior probabilities would itself entail a fitted NB model; in such a case, the GaussianNB object could be used for predictions.) Thus, there was really nothing to optimize for this model. The Naive Bayes results were:

outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
dna	codon	0.9069	0.9069	0.9900	0.07 seconds
dna	amino	0.9661	0.9661	0.9753	0.03 seconds
kingdom	codon	0.5810	0.5810	0.9145	0.21 seconds
kingdom	amino	0.6159	0.6159	0.8886	0.15 seconds

The Naive Bayes models were generally worse by these metrics than any other. Their performance is particularly bad for models predicting kingdom, although I'm not sure why this would be. It's also worth mentioning that, unlike the other models considered, using aminos for predictors actually *improved* model assessment. While not producing terribly accurate models in comparison to other methods, Naive Bayes does have the advantage of very short runtime, even shorter than k-NN. But its scores are significantly worse than those of the other methods used, and there are no parameters to optimize that might improve on them.

Conclusions

One goal of this project is to reproduce the prior study (Khomtchouk, 2020) referred to in the introduction, which produced the following results:

Table 1: Kingdom Classification Results

Model	Precision	Recall	Micro F1-Score	Macro F1-Score	Accuracy	AUC
k-Nearest Neighbors	0.9660	1	0.9827	0.9293	0.9660	0.9792
Random Forests	0.9298	1	0.9636	0.8611	0.9298	0.9954
Extreme Gradient Boosting	0.9502	1	0.9745	0.8846	0.9502	0.9970
Artificial Neural Networks	0.9132	1	0.9546	0.8425	0.9132	0.9901
Naïve Bayes	0.7200	0.3529	0.4737	0.5487	0.6561	0.8410

Table 2: DNA type Classification Results

Model	Precision	Recall	Micro F1-Score	Macro F1-Score	Accuracy	AUC
k-Nearest Neighbors	0.9942	1	0.9971	0.9867	0.9942	0.9997
Random Forests	0.9915	1	0.9957	0.9832	0.9915	0.9993
Extreme Gradient Boosting	0.9938	1	0.9969	0.9860	0.9938	0.9997
Artificial Neural Networks	0.9915	1	0.9546	0.9813	0.9915	0.9997
Naïve Bayes	0.9085	0.8897	0.9379	0.8353	0.8870	0.9400

I can now compare the results from my own study:

model_type	outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
kNN	kingdom	codon	0.9384	0.9285	0.9593	5.07 sec
MLP-Hyperopt	kingdom	codon	0.9321	0.9321	0.9949	567.74 sec
RFC	kingdom	codon	0.8517	0.7532	0.8746	8.73 sec
XGB	kingdom	codon	0.7522	0.7522	0.9413	6.71 sec
XGB-Hyperopt	kingdom	codon	0.5810	0.5810	0.9145	161.09 sec
GNB	kingdom	codon	0.5810	0.5810	0.9145	0.21 sec

model_type	outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
MLP-Hyperopt	dna	codon	0.9933	0.9933	0.9992	278.31 sec
kNN	dna	codon	0.9928	0.9928	0.9918	3.48 sec
RFC	dna	codon	0.9892	0.9866	0.9849	3.25 sec
XGB	dna	codon	0.9779	0.9779	0.9977	1.38 sec
XGB-Hyperopt	dna	codon	0.9069	0.9069	0.9900	48.34 sec
GNB	dna	codon	0.9069	0.9069	0.9900	0.07 sec

Perhaps it would have been better to have computed the same metrics as used in the prior study; I decided to sort on the F1 scores for the test set, in advance, and ran hyperopt to use it for the loss function.

Several comparisons can be made. In both the 2020 study and this one, all models predicted DNA type a little better than they did taxonomic group. Also in both studies, Naive Bayes (referred to as 'GNB' in the above tables) performed quite poorly. The order in which the models rank is quite similar between the two studies; perhaps the most salient difference is that the artificial neural network models ('MLP', for Multi-Layer Perceptron in the above tables) performed better in this than in the prior study.

It's interesting to note that the metrics for Naive Bayes and Extreme Gradient Boosting using hyperopt were exactly identical. I went back to verify that some recording or other error hadn't copied rows over one or the other; their distinct run times and hyperparameters dispel this worry, though. I am at a loss as to an explanation for this – and I would write it off to chance, if it occurred in only one case. But it occurs across all four of the model categories tested, and that's simply too anomalous to accept as mere randomness. I don't have any plausible explanations, though.

These are the results for models using amino acid frequencies as features:

model_type	outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
kNN	kingdom	amino	0.8696	0.8468	0.9124	5.09 sec
MLP-Hyperopt	kingdom	amino	0.8416	0.8416	0.9743	679.36 sec
RFC	kingdom	amino	0.8202	0.7152	0.8537	7.05 sec
XGB	kingdom	amino	0.7208	0.7208	0.9156	4.26 sec
XGB-Hyperopt	kingdom	amino	0.6159	0.6159	0.8886	161.50 sec
GNB	kingdom	amino	0.6159	0.6159	0.8886	0.15 sec

model_type	outputs	inputs	test_f1	test_accuracy	test_roc_auc	run_time
MLP-Hyperopt	dna	amino	0.9918	0.9918	0.9979	357.45 sec
kNN	dna	amino	0.9892	0.9892	0.9874	3.41 sec
RFC	dna	amino	0.9840	0.9805	0.9756	2.91 sec
XGB	dna	amino	0.9671	0.9671	0.9764	0.85 sec
XGB-Hyperopt	dna	amino	0.9661	0.9661	0.9753	23.29 sec
GNB	dna	amino	0.9661	0.9661	0.9753	0.03 sec

Most notably, the performance metrics for the models using amino acid data weren't much worse than those using codons. Surprisingly, they often took somewhat *longer* to run – significantly longer, for a computationally hungry model like MLP. I would have expected some degree of time savings simply from reducing the number of features; this is something that should be investigated further. But it's clear that we can make models to predict these categories based on amino acid ratios that are very nearly as effective as those using codons, and this might be of some further use.

It's worth noting that yet again, the results for XGBClassifier using hyperopt and for Gaussian Naive Bayes were exactly identical. (This pertains across training and validation sets as well. This is quite anomalous.)

The overall best model is the MLP classifier, with hyperparameters obtained from hyperopt. The optimal hyperparameters found by model category were:

inputs	outputs	activation	hidden_layer_sizes	learning_rate_init	test_f1	test_accuracy	test_roc_auc
codon	dna	0	121	0.000739	0.993316	0.993316	0.999204
amino	dna	1	88	0.008937	0.991774	0.991774	0.997929
codon	kingdom	0	92	0.000439	0.932134	0.932134	0.994854
amino	kingdom	2	144	0.004685	0.841645	0.841645	0.974279

These are also found in .csv format in the model metrics file.

Next Steps

There are several subsequent steps that could improve upon this study. Substantially more graphical EDA would be nice. For instance, it would be helpful to produce heatmap plots to visualize shap results across all categories; I struggled to create these, but encountered persistent type errors. The significant correlations found among the amino acids and particularly the codons are worth exploring, too. This could provide insight into the bias in the particular codon use per amino acid, for those amino acids with variable encoding.

One very important anomaly that should be resolved with a follow-up study is the results surrounding Extreme Gradient Boosting using hyperopt. The model metrics when applying hyperopt were substantially lower overall than those obtained using XGBoost defaults – by itself, this is quite surprising. I double-checked to ensure that the hyperparameter search space included the defaults; this really should not happen. It would be worthwhile to repeat this, or perhaps try some other optimization tool. It's also puzzling (to say the least) that the metrics for XGBoost with hyperopt were identical to those produced by Gaussian Naive Bayes. These are completely unrelated models, and to have every metric turn out exactly equal across all tests performed is simply too improbable to be due to random chance. I investigated thoroughly the possibility that I simply wrote over results, in either the dataframe that stored them or in subsequent spreadsheet work – I'm quite satisfied that this didn't happen. But obviously something is going on here that should be explained.

Another small matter that might merit further study: these models were fitted on data trimmed of the most sparse categories to improve fit. I assumed that these observations would exert a lot of leverage on the models – it could be informative to fit models using the untrimmed data, to test that assumption.

I should also follow up and include all performance metrics used in Khomtchouk's 2020 study for more complete comparison.

1 Khomtchouk, Bogdan B. 2020. Codon Usage Bias Levels Predict Taxonomic Identity and Genetic Composition. BioRxiv doi: 10.1101/2020.10.26.356295

2 Cover, Thomas M.; Hart, Peter E. 1967. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*. 13 (1): 21 – 27.   doi:10.1109/TIT.1967.1053964

3 Breiman, L. Random Forests. *Machine Learning* 45, 5–32 (2001). https://doi.org/10.1023/A:1010933404324

4 Friedman, Jerome H. Greedy function approximation: A gradient boosting machine. Ann. Statist. 29 (5) 1189-1232, October 2001. https://doi.org/10.1214/aos/1013203451