ECE 1504 Statistical Learning Course Project

Predicting Conflicting Clinical Classifications of Genetic Variants

1. Introduction

A genetic variant is a deviation in a gene from the general genome. Variants can have a wide spectrum of effects on an individual. Usually, these genetic variants are classified by different clinicians and researchers as one of: *benign, likely benign, uncertain significance, likely pathogenic, or pathogenic.* However, the categorical classification is not always the same across all researchers. This conflicting clinical classification is indicative of the fact that we haven't understood the exact nuts and bolts of how genetics work. Thus, in this project, we are exploring the genetic variant database from ClinVar on NCBI [**?**]. We are trying to explore the database with the following goals in mind:

1. If we are able to predict which genetic variant is likely to get a conflicting classification and which are likely to not, research focus in studying them can be wisely distributed.

2. Besides, we can also explore which reported features aren't so crucial in assigning a category to the variants.

2. Features

Every genetic variant is described by a set of parameters, which in machine learning terms are usually described as *features*. The ClinVar (https://www.ncbi.nlm.nih.gov/clinvar/) database assigns 45 features to each variant and a label either **0**: indicating an undisputed classification or **1**: indicating that there has been a conflicting classification between different researches for the variant. The features include but aren't limited to: the chromosome on which the variant is located, the position on the chromosome where the variant is located, the reference & alternative allele, the resulting genetic disease, the variant type, the variant allele used to calculate the consequence, etc. Not all the features are useful owing to a variety of reasons varying from missing values for a large number of variants to large number of categories or a lack of variation in their values in the database. This means that significant feature engineering is required, which we talk about in the next section.

3. Feature Engineering

Many of the features are string, so we one-hot encode them for use in classification algorithms. However, one-hot encoding increases the dimensions of the feature vector by a large number and for that reason, we used it only for those features which were demonstrably effective and useful. This approach helped to reduce the feature vector size to a great extent. We also discarded features which did not seem useful for one of the reasons mentioned in Section 2. So, the final features that we used are:

- Chromosome on which the variant is located
- Position on the chromosome
- Allele frequencies from GO-ESP, ExAC & Genomes Project [**?**] [**?**] [**?**]
- Strand value - either forward or reverse
- Phred-scaled CADD score [**?**]

Out of all these, only the chromosome on which the variant was present was one-hot encoded.

Another problem we faced with the database was the fact that it had a lop-sided distribution with 75% of the variants being classified as non-conflicting and only 25% having a conflicting classification. Thus, directly training the classifier models on the imbalanced database would have caused poor results. To deal with this, we undersampled the non-conflicting class so that the dataset was balanced between the two classes, and trained our machine learning models on the balanced database. For all the feature engineering in this section, we used the PANDAS [?] library in a Python3 environment.

4. Models Compared

We evaluated six kinds of classification algorithms using the dataset described above. Each algorithm is described in this section along with its main hyperparameters.

Logistic Regression

Logistic regression takes the dot product of the input vector with some weights, then applies the logistic function to produce an output between 0 and 1, which is interpreted as the probability of the input belonging to class 1 [?]. A variant available in scikit-learn that we also evaluated uses cross-validation on the training dataset to estimate ideal hyperparameter settings, then uses them in inference[1]. This was used in place of manually testing hyperparameter settings.

k-nearest Neighbours Classification

k-nearest neighbours classification takes the k closest entries in the training set, in terms of Euclidean distance from the input vector, and outputs the class that the majority of the neighbours belong to [?].

Decision Tree Classification

A decision tree classifier uses a tree where each node represents a decision about which branch to follow based on the value of a parameter, and the leaves represent classification results [?].

Random Forest Classification

Random forest classification trains many decision trees using random partitions of the training data, then classifies input vectors by letting the decision trees vote on the class [?]. We evaluated using different numbers of decision trees to improve results.

Multi-layer Perceptron Classification

A multi-layer perceptron (MLP) contains multiple layers that each apply a linear transformation to their input vector, then apply a non-linear function to produce an output vector that is passed to the next layer [?]. We used an MLP classifier with the softmax non-linearity on the output, the ReLU non-linearity on other layers, Adam optimization [?] for training, and L2 regularization with a corresponding hyperparameter λ. We evaluated varying the number of nodes in each hidden layer, varying the number of hidden layers, and varying the value of λ.

Support Vector Machine Classification

Support vector machines (SVMs) allow for training to take into account the margins around the decision boundary, which improves robustness. They also allow the efficient use of non-linear kernel functions on the data to produce non-linear classification boundaries [?]. We tested using different kinds of kernel functions in the classifier.

5. Experiments

The experiments were coded in Python3 using Numpy [?] and scikit-learn [?], which contains implementations of the classifiers evaluated here.

Balancing and preprocessing the dataset resulted in 32 398 records, which were randomly partitioned into a training set containing 24 298 records and a test set containing 8 100 records. All hyperparameter tuning results used the same training and test sets. The metrics used for reporting results are overall classification accuracy and the average precision, recall, and F1 score over both classes. Higher values are better for all metrics.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html

Table 1: k-nearest neighbours classifier performance with different values of k.

k	Accuracy	Average Precision	Average Recall	Average F1
2	57.8%	0.59	0.58	0.56
5	61.4%	0.61	0.61	0.61
10	**62.5%**	**0.63**	**0.63**	**0.63**
20	61.5%	0.62	0.61	0.61
50	61.4%	0.62	0.61	0.61
100	61.4%	0.61	0.60	0.60
200	60.4%	0.60	0.59	0.58

Table 2: Random forest classifier performance with different number of trees.

Number of trees	Accuracy	Average Precision	Average Recall	Average F1
2	61.0%	0.63	0.61	0.60
5	65.0%	0.65	0.65	0.65
10	66.3%	0.66	0.66	0.66
20	68.0%	0.68	0.68	0.68
50	68.4%	0.68	0.68	0.68
100	69.1%	0.69	0.69	0.69
200	69.0%	0.69	0.69	0.69
500	69.1%	0.69	0.69	0.69
1000	69.1%	0.69	0.69	0.69
2000	69.1%	0.69	0.69	0.69
5000	**69.2%**	0.69	0.69	0.69

Table 3: MLP classifier performance with different topologies.

Number of hidden layers	Number of neurons per hidden layer	Accuracy	Average Precision	Average Recall	Average F1
1	100	65.9%	0.66	0.66	0.66
1	200	63.8%	0.65	0.64	0.63
1	300	65.9%	0.66	0.66	0.66
1	400	65.9%	0.67	0.66	0.65
1	500	65.9%	0.67	0.66	0.65
2	50	64.4%	0.65	0.64	0.64
2	100	64.9%	0.65	0.65	0.65
3	20	64.3%	0.65	0.64	0.64
3	50	64.3%	0.64	0.64	0.64
3	100	64.0%	0.64	0.64	0.64

Table 4: MLP classifier performance with one hidden layer containing 300 neurons and different regularization parameter values.

λ	Accuracy	Average Precision	Average Recall	Average F1
10^{-1}	62.0%	0.62	0.62	0.62
10^{-2}	64.6%	0.65	0.65	0.64
10^{-3}	65.9%	0.66	0.66	0.66
10^{-4}	65.9%	0.66	0.66	0.66
10^{-5}	**66.6%**	**0.67**	**0.67**	0.66
10^{-6}	65.3%	0.65	0.65	0.65

Table 5: SVM classifier performance with different kernel types.

Kernel function	Accuracy	Average Precision	Average Recall	Average F1
Linear	57.1%	0.58	0.57	0.56
Polynomial	59.2%	0.60	0.59	0.58
RBF	**60.1%**	**0.61**	**0.60**	**0.59**
Sigmoid	51.2%	0.51	0.51	0.51

Table 6: Final comparison of different classifiers, with best known parameters chosen for each.

Classifier	Accuracy	Average Precision	Average Recall	Average F1
Logistic	59.9%	0.61	0.60	0.59
Logistic with auto-tuning	60.0%	0.61	0.60	0.59
k-nearest neighbours	60.9%	0.61	0.61	0.61
Decision tree	63.0%	0.63	0.63	0.63
Random forest	**68.6%**	**0.69**	**0.69**	**0.68**
SVM	60.8%	0.64	0.61	0.59
MLP	65.1%	0.65	0.65	0.65

Results

Table 1 shows the performance of the k-nearest neighbours classifier for different values of k. Setting this value too low or too high caused poorer performance, likely due to considering either too much or too little data to get a good idea of the neighbourhood characteristics. $k = 10$ was the best value tested according to all metrics used.

Table 2 shows the performance of the random forest classifier with different numbers of decision trees. For the values tested, having more decision trees generally improved performance, although with very little improvement beyond 100 trees. The highest classification accuracy was achieved with 5000 trees.

Tables 3 and 4 show the effects of topology and hyperparameter choices on the performance of the MLP classifier. In general, adding more than one hidden layer was not shown to help on this dataset. It is unclear what would be the best choice for the number of neurons in the hidden layer, since different values showed similar results. For later testing, 300 neurons were used. Low amounts of regularization provided better performance, with the best results seen at $\lambda = 10^{-5}$.

Table 5 shows the performance of the SVM classifier with different kernel functions. For all metrics measured, RBF kernels provided the best performance.

To summarize results, the best hyperparameters found were used for each model, and they were all trained and tested again with a different random training/test set split, in order to generate a better estimate of the out-of-sample error. The results are shown in Table 6. For all metrics measured, the random forest classifier had the best performance, followed by the MLP classifier.

6. Conclusions

The best classification performance was achieved by a random forest classifier. A neural network classifier provided the second best performance, but was was more complex to tune, as well as being slower to train, making the random forest classifier a much more compelling solution. Classification accuracies above 70% were not achieved, which reflects the difficulty of the classification task in this dataset.