MAP4112 – Final Project Report
Andre Rodriguez

Introduction

Classification problems are a common type of machine learning problem in which the model learns to separate samples of data into discrete classes, as opposed to returning a continuous value as the output. There are of course plenty of algorithms to separate data, some may work like the models for a continuous value but set a range of possible values for each class, some may classify a sample by finding what class the samples most like it in the training set belong to, and some may find a way calculate a probability for a sample to be within each class among many ways. This project will cover a probabilistic model for our classification problem where we will use Bayes Theorem to calculate probabilities. Bayes Theorem is given by this equation:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

This theorem describes a convenient way to find the conditional probability that event A occurs given event B has occurred; this is otherwise known as the posterior. For the purposes of machine learning and classification, we can apply this calculation to each sample, where A would be the class we seek to find the probability for, and B is the set of data already available where the samples all fall into class A. Through the report we will discuss how to turn this into a machine learning model, give an example of a handwritten model performing this computation, and compare against other common classification algorithms using a common dataset.

Discussion

Let's describe the basic aspects of the algorithm called Bayes Optimal Classifier, which we will try to implement later for our machine learning problem. Let's say there is a hypotheses space H and a space of known data D. For a sample we are looking to classify, we want to find the class $h \in H$ such that the probability that the sample is in h is maximized. Given sample x we can mathematically model the classifier very simply as:

$$argmax \sum_{h_i \in H} P(x|h_i) * P(h_i|D)$$

Here $P(x|h_i)$ is the posterior probability describing how likely x exists given the class h, and $P(h_i|D)$ is the posterior describing how likely the class h_i appears given the dataset D. Altogether this will calculate the probability that our sample is in each class and return the most likely class.

This classifier is said to be optimal, such that no other classification model with the same input spaces can outperform its accuracy. However, this is only in theory as the classifier functionally has problems that may make this classifier nearly impossible to use. The main problem occurs with the dataset and calculating conditional probabilities from it. The classifier assumes that every feature $x_i \in X$, where X is the set of sample inputs, is conditionally dependent on each other, which can often be true in real life scenarios. The computational cost of trying to find the likelihood probabilities and the incredible size of the input data needed when the features are conditionally dependent makes computing Bayes Optimal Classifier impractical or even impossible. So, to make as much use of the optimal accuracy as possible we can modify the model to instead assume the opposite, where every

feature is independent of each other. This simplification model is called Naïve Bayes, and while the assumption leads to some loss in accuracy, it still excels as a classifier.

The mathematical model for Naïve Bayes still takes the shape of the same optimization problem as before however the calculation for the likelihood probability of $P(h_i|\{x_1, x_2, \dots x_n\})$ changes when applying Bayes theorem to the conditional probabilities. The model for the probability for one class and one sample changes as such:

$$P(h_i|\{x_1, x_2, \dots x_n\}) = P(\{x_1, x_2, \dots x_n\}|h_i)P(h_i) \approx P(x_1|h_i)P(x_2|h_i) \dots P(x_n|h_i) * P(h_i)$$

Now each input variable has its own probability given each specific class that can be found even in relatively small datasets, and now finding these probabilities by assuming each input can be modelled by a simple distribution is computationally very simple.

For comparisons sake, we will look at how other classifiers perform. These classifiers all exist in the scikit-learn library. So very quickly let's list each classifier we will test and a simple description of how they work.

First, AdaBoost is a very common classification algorithm that makes use of repeated uses of a singular classification algorithm. It first fits on the original dataset, and then uses a new estimator to fit on the same dataset weighted on the instances that the previous estimator was fit incorrectly. It repeats this process as many times as told or until a perfect fit.

Second, Random Forest is another classifier like AdaBoost where it uses an ensemble of classifiers to make its prediction. Here Random Forest uses a given number of decision tree classifiers on subsets of the dataset and uses averaging to find predictions. Each decision tree makes classifications based on simple rules taken from modelling the dataset. This creates what is mathematically a piecewise function that models the dataset.

Third, K-nearest neighbors is a non-linear classifier, where instead of crafting a functional model of the dataset to calculate what to classify each value as, it takes each sample and compares it to the closest k examples in the training data. Using these comparisons, it pulls a prediction from what is most likely.

Last, we will use Naïve Bayes classifiers given by scikit-learn as well as my own. The first of the two assumes the features follow a Multinomial distribution, the second will assume they follow a Gaussian distribution.

Implementation

In the code BayesClassifier.py, I created a simple python class for my Naïve Bayes classifier, aptly called MyNBClassifier. This class only implements three functions: fit, predict, and bayes. Both fit and predict are structured in the same way any scikit-learn model is, so code can be easily reproduced. The Bayes function is mainly a helper function for the prediction that calculates Bayes theorem for a given class using the Naïve Bayes adjustment to Bayes theorem as previously discussed. The model fits to the dataset by creating a 3D array that consists of the mean and standard deviation for each (feature, class) combination, and it calculates the priors as well. The means and standard deviations held in this array exist as parameters that model a Gaussian distribution. From this distribution we can find the likelihood for any feature value given the class.

MAP4112 – Final Project Report
Andre Rodriguez

Within the code ClassifierComparison.py, I simply initialize, fit, and predict all the classifiers previously listed along with my own to a given dataset. Following that, I print basic accuracy results and show runtime for each model. The dataset I have chosen is a dataset from the UCI Machine Learning Repository and can be found with the file name wine.csv. It gives data on chemical analysis of wine that was grown and produced from three different cultivars in Italy. This dataset can be considered quite small with only 178 wine samples and 13 attributes for each. It is also a multivariate classification problem instead of a binary one, as there are 3 classes of wine, one for each cultivar. The dataset is also quite simple, there are no missing data points, and every value is continuous.

Results

For the results shown in this report, I have split the dataset so about 30% of it was used for the testing data. I also arbitrarily set a random_state=5 for any function that takes advantage of randomness to compute, so that there could be a reproduceable set of results for this report.

	Cultivar 1 acc.	Cultivar 2 acc.	Cultivar 3 acc.	Total Accuracy %	Runtime (s)
AdaBoost	22/23	16/18	10/13	88.89%	0.0369
RandomForest	22/23	17/18	13/13	96.30%	0.0637
KNN	20/23	11/18	7/13	70.37%	0.00142
MultinomialNB	17/23	14/18	11/13	77.78%	0.00049
GaussianNB	20/23	18/18	13/13	94.44%	0.00046
MyNB	20/23	18/18	13/13	94.44%	0.00278

Above is the table documenting the accuracies for each classifier I tested, as well as the runtime given the specific random state. First thing to note is that the best performing algorithm in this case is the Random Forest Classifier with 96% accuracy. Following that is both the GaussianNB and my handwritten Naïve Bayes classifier. In retrospect, both algorithms fit Gaussian models along each feature, and calculate Bayes theorem under the same principle, so of course they are functionally the same algorithm and will return the same results, regardless of random state as well. I, however, was a bit surprised when I saw this result on first run. I had assumed that the more sophisticated scikit-learn would have small optimizations and improvements that would make their algorithm perform better, even if just slightly. The big difference here is that the model from scikit-learn is much more efficient runtime-wise.

Along with the Random Forest AdaBoost also generally performed quite well, with an 89% accuracy on this trial, but upon rerunning with truly random states, it seemed to be more accurate than shown here. Although these complex ensemble models perform well, their runtimes made up the two slowest here. They can achieve the same results as Naïve Bayes but less efficiently, or at least given this small of a dataset.

The two underperforming classifiers in comparison were the K-Nearest neighbors and the Naïve Bayes using a Multinomial distribution. For K-nearest neighbors, it seems it struggled to differentiate between Cultivar 2 and Cultivar 3. Considering K-nearest neighbors takes advantage of clustering, there may have been some overlap between the clusters for the two classes. Increasing the value k to consider more neighbors may make this perform better as it could dilute away the outlier samples causing the model to predict incorrectly. For the Multinomial version of Naïve Bayes, it is probably as

simple as the Multinomial distribution did not fit each feature set very well. The Gaussian on the other hand, was a great fit for the dataset.

Conclusion

From what we have gathered here, the Naïve Bayes algorithm is a very simple and efficient way to estimate the results of the Bayes Optimal classifier, and it performs very well along with other common and very sophisticated classification models. As I was able to make a handwritten classifier that equaled in performance to that of scikit-learn's, it shows not only how effective Naïve Bayes is as a model, but also how simple it is to set up and understand. Likewise, it is very simple computationally and can run quite efficiently. One thing missing in my trials comparing Naïve Bayes to other classification models was hyperparameter tuning the models for maximum performance. I had considered implementing it but left it out for a few reasons. One, for the sake of simplicity, but two, I found it interesting that Naïve Bayes makes use of essentially no hyperparameters outside of adding weights to the feature sets or using a different distribution for the calculation. Whereas AdaBoost lets you fine tune the model for its learning rate and number of estimators used, or Random Forest includes parameters for decision criteria, depth of tree, samples needed to split or to end a tree, and plenty more. Not to mention that performing hyperparameter tuning through a process like grid search or random search with cross validation is increasingly computationally expensive and starts to eat away at the runtime, which isn't an issue with Naïve Bayes as there isn't much to tune. I figured to test against the default values of all the models, as there was no hyperparameter tuning to be had in Naïve Bayes, which only adds to its incredible simplicity. However, I could foresee this also being a con, as you are stuck with what you have when using Naïve Bayes, while with other classifiers you can fine tune to continuously perfect how you model a dataset.

Among other issues to discuss in the project was also the sample set. I previously commented a bit on how it is quite a small and simple dataset. Again, I had a few reasons to do this: firstly, the sake of brevity and ease of displaying results that are clearly understandable. Secondly, it was a point at the introduction of the Bayes Optimal classifier that its use is mostly theoretical as you would need an extremely large dataset to make use of said classifier. Choosing this dataset was able to show that a quick modification to the classifier takes it from needing a nearly impossibly large dataset and excessive amounts of computational power to such a simple and efficient classifier that performs well on even very small datasets.

If I were to run this project again, I would investigate hyperparameter tuning the other classifiers to find maximal performance, as well as find a dataset or two that is on the large side. All of this could go into evaluating not only the performance, but the flexibility of each model as well. We could've potentially seen a big change in runtime as well, with bigger datasets possibly putting more emphasis on a model's computational efficiency per element and the setup time for a complex model being more irrelevant. There is also testing on binary datasets, like a dataset where instead of maximizing the accuracy of all classes, its goal is to identify as many negative classifications with a false positive rate as low as possible. Other datasets could've also included ones with missing values, as in theory Naïve Bayes wouldn't have a need for data wrangling or imputing like some of the other classifiers would. Classification problems are quite varied, and there are many possible projects to consider how the results would differ after seeing the result of this simple and generalized problem.

MAP4112 – Final Project Report
Andre Rodriguez

References

- https://machinelearningmastery.com/bayes-optimal-classifier/
- https://scikit-learn.org/stable/modules/classes.html
- https://archive.ics.uci.edu/ml/datasets/Wine
- https://numpy.org/doc/stable/reference/
- https://docs.scipy.org/doc/scipy/
- https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html