Project 2

Edward Wang, Kyle Love

2020-08-03

1 Naive Bayes

Naive Bayes is an algorithm for recognizing objects based upon the probability of their collective features, i.e, height, width, color, et cetera. In real life, if enough features line up with our mental image of a certain class of object, we can determine that the object is likely to be in that class easily. For example, identifying a car by the fact it has four wheels, is made out of metal, and makes roaring sounds. But computers aren't able to make those deductions because they don't have the knowledge base or reasoning ability that we do. So how do we solve this issue?

The first step is building a base of knowledge to work off of. To determine that a particular object is of a certain class, we must know that each member of the class has certain features, and that the object's features match up with the class's features enough that the probability of the object being in the class is greater than the probability of any other class being the case. To that end, we must have sufficiently detailed features that there is little, if any, likelihood of false positives.

We used the training data to find probability distributions for each feature in each class of object, finding the frequency of each value at each feature. Then, we applied it. Firstly, we took the prior probability, that is, the frequency of the class with relation to the overall training set and the initial likelihood of picking the class of the object blindly. Then, for each feature in the object being checked, we multiplied by the probability of that feature having that value in the class. Naturally, if the probability of the feature was higher, the likelihood of the object being of that class would increase. We then collected the values found for each class and picked the larger of them to find the object class.

2 Perceptron

Perceptrons are the individual layers of a neural network, which has become the most important and common type of AI in our world today. Although not as accurate as its larger counterpart, Perceptron is still good at coming up with small scale predictions and judgements.

The Perceptron algorithm uses linear equations to separate the classifications of different objects. The algorithm begins by creating weight vectors for each type of class that needs to be discerned, and assigning each of them zeroes for the number of features that each object contains. These weight vectors are simply just lists that are initialized to 0, and they essentially make up the values for a very long equation. The second step of the algorithm is to train against some test data.

3 Feature Design

Our features were initially not very accurate, and they yielded results far below the 60/70% threshold for acceptability. For digits, we started by using 0 for blank spaces, 1 for #, and 2 for +, which yielded success rates around 40%. It seemed odd that this had occurred. After all, common sense would dictate that higher detail in features would result in more accurate results. But we soon realized that that was exactly the issue. By making more detailed features, we lowered the probabilities across the board for each of them, making it more likely that the object would be misidentified. So instead, we moved to use a simple, binary schema, where 0 indicated blank space, and 1 indicated a symbol.

On the other hand, the facial features suffered the opposite issue. They were too sparsely detailed to determine the faces properly, especially considering the size of each image and their largely line defined shapes. As such, instead of pixels, we moved to using 2x2 boxes to define features, giving them a numerical value based on the presence/shape of lines within them. The result was a drastic increase in the accuracy, up to proper thresholds for acceptability.

4 Test Data

Table 1: Naive Bayes Test Results

Naive Bayes	Validation Data		Test Data	
% of Training Data	% Digits	% Faces	% Digits	% Faces
10%	81	56	72	52
20%	80	92	75	71
30%	83	97	78	75
40%	85	96	78	79
50%	85	96	79	82
60%	83	93	80	83
70%	84	92	80	83
80%	83	90	80	83
90%	84	89	79	83
100%	85	89	79	82

Table 2: Perceptron Test Results

Perceptron	Validation Data		Test Data	
% of Training Data	% Digits	% Faces	% Digits	% Faces
10%	82	84	70	73
20%	71	93	71	82
30%	84	99	80	85
40%	85	99	81	86
50%	89	97	80	88
60%	87	100	85	85
70%	89	100	84	84
80%	86	99	83	87
90%	89	100	76	87
100%	89	98	83	87

Table 3: Runtime Results in Seconds

	Naive Bayes		Perceptron	
% of Training Data	Digits	Faces	Digits	Faces
10%	2.33	1.57	19.08	2.22
20%	3.13	1.79	36.97	2.88
30%	4.08	2.07	52.82	3.49
40%	4.97	2.32	69.44	4.23
50%	5.73	2.63	88.07	4.85
60%	6.74	2.76	107.51	5.43
70%	7.53	2.97	120.49	6.13
80%	8.4	3.33	135.65	6.86
90%	9.33	3.54	153.57	7.51
100%	10.24	3.7	171.68	8.1

5 Conclusions

As our data shows, both algorithms meet the requisite sixty percent minimum to be effective. However, it is clear to see that the Perceptron algorithm, although being somewhat variable in results, has a generally higher degree of accuracy than Naive Bayes. For example, while the Naive Bayes validation identification rate fluctuates around 90-95% near 100% training data for faces, Perceptron reaches 100% or near it rather consistently around the same area. Perceptron also tends to hit high degrees of accuracy earlier on, while Naive Bayes can sometimes start with accuracy in the fifties. Evidently, Naive Bayes needs a higher minimum amount of training data to hit decent accuracy.

That said, for all its apparent benefits over Naive Bayes, the Perceptron algorithm also seems to be consistently slower than Naive Bayes for both digits and faces. Although both algorithms seem to have consistent linear runtime growth, Perceptron takes more time to process each additional amount of data provided, resulting in higher runtimes across the board, and in the case of

digits, even exceeding its counterpart in time by an order of magnitude. Thus, although Naive Bayes is somewhat less accurate, it may be more usable for everyday applications, or at least more useful for consistently identifying new classes of data.

It's also interesting to note that in some cases, the success rate fell as more training data was introduced for both algorithms. It seems like it is possible to have too much data. Why? One possible reason is that, by random chance, the cumulative model from the data introduced at a certain point was closer to the ideal of the class of object, and further data only fudged the numbers, so to speak. Or conversely, the data introduced always brought the model closer to the ideal, but that resulted in an inability to account for outliers in the test data. It seems that sometimes, it's better to have a 'fuzzier' model that's less accurate to the ideal so that outliers are more likely to be properly identified.