Review of CS 260 Concepts

CS 360 Machine Learning
Week 1, Day 2

January 25, 2024

Contents

1 Multi-class confusion matrices

Continuing from our discussion last class, here's another example of a confusion matrix and its normalization (see textbook Figure 3-9). This is for a problem of classifying handwritten digits. You have your true labels (handwritten digits) 0 through 9 and these show the raw counts (left) and the normalization by row (right). You can see (in the confusion matrix on the left) that there are thousands and thousands of examples - you can see that it's a robust dataset. On the right with the normalization though you can see that you get more interpretability. For example, what's the highest off-diagonal (error) percentage? It's 10% – it's the 5 being confused with the handwritten 8. What do we think might be happening there? The 5 might have been written with a bit of a loop and it might look like an 8. What other numbers do you see have high confusion? 3 and 8 get confused a lot, and 9 and 8, probably in both cases also because they're written in a somewhat more loopy fashion.

What other things do you notice about this matrix? The accuracy is fairly high. I might have thought that 1 and 7 get confused a lot, but the error for those terms are fairly low. You might also notice that the error is not symmetric. When you have multi-class classification you can really see a lot from a confusion matrix.

2 Naïve Bayes review

The most common way to write Bayes rule in a Bayesian model context is to write it as:

$$p(y|\vec{x}) = \frac{p(y)p(\vec{x}|y)}{p(\vec{x})}$$

But Bayes rule is not just one way of writing anything. A form of Bayes' rule that's also valid is:

$$p(\vec{x})p(y|\vec{x}) = p(y)p(\vec{x}|y).$$

In the context of a multi-class classification problem we would consider $p(y = k|\vec{x})$ where k is a specific class $k \in \{1, 2, 3, ...K\}$ where K is the total number of possible classes. (In the examples today we're just going to have 2 classes but it could be extended.) In that case, we can rewrite Bayes rule as:

$$p(y = k|\vec{x}) \;=\; \frac{p(y = k)p(\vec{x}|y = k)}{p(\vec{x})}$$

That's our typical Bayesian model. In order to make predictions (denoted $\hat{y}$) we look for the y value with the highest probability based on Bayes rule:

$$\hat{y} = \text{argmax}_{k=1}^{K} \frac{p(y = k)p(\vec{x}|y = k)}{p(\vec{x})}$$

This value is computed for each class and then we choose the highest one. That's why for many Bayes models we can ignore $p(\vec{x})$ because it's the same for all of our classes. Otherwise $p(\vec{x})$ can be hard to compute, but since it's the same for all values of k we can ignore it and still determine $\hat{y}$. The function `argmax` returns the argument that caused us to have the maximum value, so in this case that's the class that results in the largest value, and we can then return that class value.

How do we compute this and what specifically makes the Naïve Bayes algorithm "naïve"? We look at this part of Bayes rule, the likelihood: $p(\vec{x}|y = k)$. Computing the likelihood can be tricky (often the prior is not as hard). We can expand this using the understanding that $\vec{x}$ is our feature vector and potentially has many different aspects to it, like pixels in an image or different parts of a person's census data. We can rewrite the likelihood as:

$$p(\vec{x}|y = k) \;=\; p(x_1, x_2, .., x_p|y = k)$$

We use p in x_p above to denote the total number of features in $\vec{x}$, where x_i denotes the ith feature. Recall that commas in this context mean a joint probability, another way of thinking about that is as "and," this means that you need to see these features together all in the same person in the census data example, for example the probability that a person is both a specific age and gender. Determining joint probability can be difficult, so our naïve Bayes assumption—a key part of the algorithm—is that our features are conditionally independent given the label. With that assumption, we can rewrite this as an approximation where:

$$p(\vec{x}|y = k) \;\approx\; p(x_1|y = k) \cdot p(x_2|y = k) \cdot ... \cdot p(x_p|y = k)$$

This is *not* true in general; $p(A|B) \neq p(A)p(B)$. But if they are independent, then $p(A|B) = p(A)p(B)$. Suppose that I had two features with the goal of predicting what animal is being described, so for example x_1 = has four legs and x_2 = has fur and y = is cat. If we know already that an animal is a cat, knowing it has fur doesn't really tell us much about if it has four legs. Since we don't gain additional information by knowing another feature, they are conditionally independent. This is a hugely simplifying assumption, but the Naïve Bayes algorithm works anyway in a lot of contexts in practice.

Substituting the expansion of the likelihood using this naïve assumption into our Bayesian model above we get:

$$p(y = k|\vec{x}) \;\approx\; p(y = k) \prod_{j=1}^{p} p(x_j|y = k)$$

Note that $\prod$ is a function that multiplies all given values together (the equivalent for product of $\sum$).

Here $\vec{x}$ is a single example. In our matrix X, which is $n \times p$, $\vec{x}$ is a single example (row) from that matrix, which has p features. In order to estimate these probabilities you need to go through all features for each example. So this is thinking about this model on a single test example.

Consider the handwritten digits dataset - the features are the pixel values for every single pixel. They're not particularly interpretable - they're the greyscale values of the specific pixels. Some dataset features are interpretable, but some are definitely not.

How can we determine the different components for this model? The first part $p(y = k)$ is our prior and we would ideally estimate it from the population. In practice, we measure this as the empirical probability observed in our training data, in other words:

$$p(y = k) \;=\; \frac{N_k}{n}$$

where N_k is the number of training examples labeled k while n is the total number of training examples. In the case where we have no examples from a single category example, we usually add Laplace counts. Because we're multiplying a lot of small things together we need to be careful, because if one of those probabilities is 0 it'll mess up the whole calculation and we'll end up with 0 overall. In these cases, we add a small amount to make sure we avoid this case. In order to avoid this for the general case, we actually determine the value by adding 1 to the total count (termed a *Laplace numerator*), and because we want the probabilities overall to sum to 1, we do that by also changing the denominator such that:

$$p(y = k) \;=\; \frac{N_k + 1}{n + K}$$

This gives our estimate of our prior from the training data. We'll see an example of this in the handout.

We now generalize that to our likelihood term:

$$p(x_j = v|y = k) = \frac{p(x_j = v, y = k)}{p(y = k)}$$

where v is a specific value. This is just a rewrite of Bayes rule. But to approximate it, we use

$$\approx \frac{N_{k,j,v}}{N_k}$$

where the number of examples where $y = k$ and $x_j = v$ is denoted as $N_{k,j,v}$.

We still need to add our Laplace term, since especially as we restrict this counting to be based on jointly occurring values, it becomes much more likely that we won't see the combination in our training dataset. Because the number of 1s we would add on is really corresponding to the number of potential values for that feature, we consider $|f_j|$, the number of possible values for feature j. Note that j is fixed throughout this example, so for example $j = 5$ would be the 5th feature x_5 which might have possible values $x_5 \in \{red, green, blue\}$.

This gives us this final estimate:

$$\approx \frac{N_{k,j,v} + 1}{N_k + |f_j|}$$

In practice, suppose that $p = 3$ features, $\vec{x}_{test} = [x_1, x_2, x_3]^T$, and each feature has two values. Now consider just the class $y = 0$ in a binary classification problem. I would first compute the prior of class 0:

$$\frac{N_0 + 1}{n + 2}$$

Then I'd include this in the full expression:

$$p(y = 0|\vec{x}) \;\approx\; \frac{N_0 + 1}{n + 2} \cdot \frac{N_{0,1,v_1} + 1}{N_0 + 2} \cdot \frac{N_{0,2,v_2} + 1}{N_0 + 2} \cdot \frac{N_{0,3,v_3} + 1}{N_0 + 2}$$

where each fractional term is, in order from left to right, the prior, the approximate probability $p(x_1 = v_1|y = 0)$, the approximate probability $p(x_2 = v_2|y = 0)$, and the approximate probability $p(x_3 = v_3|y = 0)$.

Then we'd repeat this for $y = 1$.

Consider $p(y = 0|\vec{x}) \approx 0.015$ and $p(y = 1|\vec{x}) = 0.0718$.

Why don't these sum to 1 if I just determined them based on the estimated equation? I ignored the part of the expression that divides by $p(\vec{x})$ (the evidence). I could go ahead and include that term:

$p(y = 0|\vec{x}) \approx \frac{0.015}{0.015+0.0718} = 0.16$

$p(y = 1|\vec{x}) \approx 0.84$

This is actually the output you'll get from sklearn. What class should we predict here? Class 1.

But what if our prediction should depend on a threshold, i.e., you don't want to predict it unless the value is actually high enough?

$p(y = 1|\vec{x}) \geq t$ then we predict 1, otherwise predict 0. So if the threshold was 50%, which is often informally what we imagine, then we would say that $t = 0.5$. But we could have decided that $t = 0.9$, and then even given the above output we could determine that we should predict $\hat{y} = 0$. Finally, suppose that $t = 0.1$, what would you predict? If both options are above your threshold, one option would be to determine that you don't have enough confidence to give a prediction. But given the above definition we would still predict $\hat{y} = 1$.

2.1 ROC curve review

Overall, with a threshold in place, we can create a ROC curve. (This should be review from CS 260.) Recall that a ROC curve gives the false positive rate (FPR) on the x-axis and the true positive rate (TPR) on the y-axis. We would like our false positive rate to be very low and our true positive rate to be very high – we'd love to have a point very close to the y-axis and as high as possible (in the upper left of this chart). Of course, in practice we might not get there. Our thresholds help us to develop every point on the ROC curve. At $t = 0$ then everything is positive. If everyone has a disease we won't miss anyone who has a disease —this would be an extreme prediction situation. At the other end, $t = 1$, we won't have any false positives, but will miss a lot of true positives. One high level idea about ROC curves is that depending on the application we might want more or less conservative thresholds.

Consider a medical application — would you prefer a low or high threshold? Low. Why? Because you'd rather not miss anyone — if there's any evidence of disease you want to return class 1 (flagging the person). For something like email spam you might prefer a high threshold because a user could classify the spam on their own, and you don't want to accidentally filter out someone's real emails. Note that these application-specific threshold choices might not be the same choices that maximize the overall accuracy.

Note that the $y = x$ line is what we'd get with random guessing, where you receive the full random line based on the particular probability, so 50/50 random guessing is in the middle, but you might have a weighted coin which gives you 20/80 predictions that are still random.