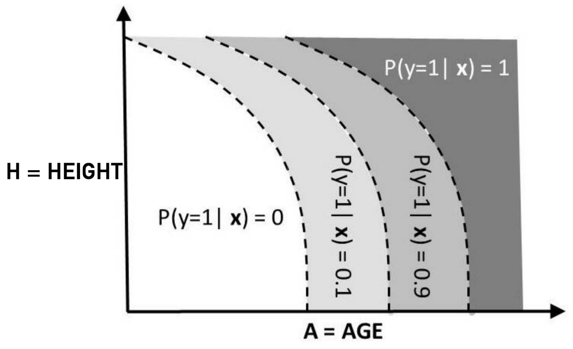
CS 189/289A Introduction to Machine Learning
Fall 2021 Jennifer Listgarten and Jitendra Malik **DIS9**

1 Nearest Neighbors and Bayes risk

A census of Middle Earth is conducted to see the extent to which we can predict that someone is an orc or an elf based on age and height. We design $x = [A, H]$ where A indicates the candidate's age and H their height. The census includes enough data for a posterior probability $P(y|x)$ where y indicates where this being is an elf.



We labels the regions above from left to right R_1 to R_4. For illustrative simplicity, we have made the conditional probability of being an elf piecewise constant so that it is constant over each of those regions.

(a) What is the Bayes risk in each of the four regions?

Here, the *Bayes risk* means the probability of error of the optimum Bayes classifier that knows the underlying pattern perfectly. The Bayes risk is the counterpart of the "irreducible error" since it reflects an intrinsic uncertainty that we cannot remove when we are trying to do prediction based on the given information.

Solution: Our Bayes' decision boundary lies between R2 and R3. According to this decision rule, left side, R1,R2, will be classified with label 0 and the right side, R3, R4, label 1. Now, we analyze them one by one.

R1: $E_{bayes} = 0$: if the sample lies in region 1, it will be labeled as 0 deterministically. Therefore, the bayes's risk of labeling it 1 will be 0
R2: $E_{bayes} = 0.1$: if the sample lies in region 2, it has a possibility of being 1 as 0.1, and we label it as 0 by the bayes' decision. Therefore, we have 0.1 chance of labeling it wrong.
R3: $E_{bayes} = 0.1$: similar to R2
R4: $E_{bayes} = 0$: similar to R1

(b) Assume that the training data is dense enough so that all the nearest neighbors of a test sample lie in the same region as the test sample itself. Now consider a test sample x which falls in region R_i. For $i \in 1, 2, 3, 4$, what is the probability that x and its nearest neighbor have different labels y?

Here, we are assuming that the training data has labels generated by the same random process that generates the labels at test time, and that the labels on training points are independent of the labels on test points.

Solution: R1: 0: if the sample lies in region 1, it will be labeled as 0 deterministically and its nearest neighbor which is also in this region labeled 0 too. Therefore, the possibility of labeling it wrong will be 0.
R2: 0.9 * 0.1 + 0.1 * 0.9 = 0.18: if the sample lies in region 2, it has a 0.1 chance to be labeled 0 and its nearest neighbor 0.9, resulting in one 0.1*0.9; by symmetry, it has 0.9 * 0.1 chance of being labeled 1 while its nearest neighbor is 0. Altogether, we have the chance of different labeled two nearest pairs 0.18.
R3: same as R2 = 0.18 R4: 0: similar to R1

(c) What is the 1-nearest neighbor classification error rate in each region?

Solution: This question is a basic explanation of the scenario mentioned in b. If we have the sample's actual label different from the nearest neighbor, which we predicted to be the sample's label, we will thus have an error. Therefore, the error rate will be what we calculated in the above question. R1: $E_{1NN} = 0$
R2: $E_{1NN} = 0.18$
R3: $E_{1NN} = 0.18$
R4: $E_{1NN} = 0$

(d) Now we generalize the problem above.

$$\begin{aligned} R1 &: P(y = 1|x) = 0 \\ R2 &: P(y = 1|x) = p \\ R3 &: P(y = 1|x) = 1 - p \\ R4 &: P(y = 1|x) = 1 \end{aligned}$$

where p is a number between 0 and 0.5.
Calculate the answers to the previous questions under this generalization. Can you see why the classification performance of 1-nearest neighbor with sufficiently dense training data is never worse than twice the Bayes risk?

Solution: We basically have the exact calculation process as above. For the previous question, we can think of p = 0.1 while 1-p = 0.9. Thus we have R1 :$E_{bayes} = 0, E_{1NN} = 0$
R2 :$E_{bayes} = p, E_{1NN} =$ 2p(1 - p)
R3 :$E_{bayes} = p, E_{1NN} =$ 2p(1 - p)

R4 :$E_{bayes} = 0, E_{1NN} = 0$

Notice here that $2p(1-p) < 2p$ and this is actually a pretty celebrated result that says that when there is enough training data and it comes from the same distribution as test data, that simple 1-nearest neighbor methods will approach something better than twice the Bayes Risk. The intuition is that there are two sources of error — the test point might itself have an irreducible error in its own label or the nearest training point might happen to be mislabled.

2 Curse of Dimensionality in Nearest Neighbor Classification

We have a training set: $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})$, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$. To classify a new point $\mathbf{x}$, we can use the nearest neighbor classifier:

$$\text{class}(\mathbf{x}) = y^{(i^*)} \quad \text{where } \mathbf{x}^{(i^*)} \text{ is the nearest neighbor of } \mathbf{x}.$$

Assume any data point $\mathbf{x}$ that we may pick to classify is inside the Euclidean ball of radius 1, i.e. $\|\mathbf{x}\|_2 \leq 1$. To be confident in our prediction, in addition to choosing the class of the nearest neighbor, we want the distance between $\mathbf{x}$ and its nearest neighbor to be small, within some positive ϵ:

$$\|\mathbf{x} - \mathbf{x}^{(i^*)}\|_2 \leq \epsilon \quad \text{for all } \|\mathbf{x}\|_2 \leq 1. \tag{1}$$

What is the minimum number of training points we need for inequality (1) to hold (assuming the training points are well spread)? How does this lower bound depend on the dimension d?

Hint: Think about the volumes of the hyperspheres in d dimensions.

Solution: Let B_0 be the ball centered at the origin, having radius 1 (inside which we assume our data lies). Let $B_i(\epsilon)$ be the ball centered at $\mathbf{x}^{(i)}$, having radius ϵ. For inequality (1) to hold, for any point $\mathbf{x} \in B_0$, there must be at least one index i such that $\mathbf{x} \in B_i(\epsilon)$. This is equivalent to saying that the union of $B_1(\epsilon), \ldots, B_n(\epsilon)$ covers the ball B_0. Let vol(B) indicate the volume of object B, then we have

$$\sum_{i=1}^{n} \text{vol}(B_i(\epsilon)) = n\text{vol}(B_1(\epsilon)) \geq \text{vol}(\cup_{i=1}^{n} B_i(\epsilon)) \geq \text{vol}(B_0).$$

where the last inequality holds because we are assuming the union of $B_1(\epsilon), \ldots, B_n(\epsilon)$ covers the ball B_0. This implies

$$n \geq \frac{\text{vol}(B_0)}{\text{vol}(B_1(\epsilon))} = \frac{c(1^d)}{c\epsilon^d} = \frac{1}{\epsilon^d}$$

Where the constant c is dependent on the formula for the volume of a hypersphere in d dimensions.

Note that we can pick $\frac{1}{\epsilon^d}$ training points and still satisfy (1) only if all the training points are well spread (the union of $B_1(\epsilon), \ldots, B_n(\epsilon)$ covers the ball B_0).

This lower bound suggests that to make an accurate prediction on high-dimensional input, we need exponentially many samples in the training set. This exponential dependence is sometimes called the *curse of dimensionality*. It highlights the difficulty of using non-parametric methods for solving high-dimensional problems.

3 Convolution and Backprop

In this problem, we will walk through how discrete 2D convolution works and how we can use the backpropagation algorithm to compute derivatives through this operation.

(a) We have an image I that has three color channels I_r, I_g, I_b each of size $W \times H$ where W is the image width and H is the height. Each color channel represents the intensity of red, green, and blue for each pixel in the image. We also have a filter G with finite support. The filter also has three color channels, G_r, G_g, G_b, and we represent these as a $w \times h$ matrix where w and h are the width and height of the mask. (Note that usually $w \ll W$ and $h \ll H$.) The output $(I * G)[x, y]$ at point (x, y) is

$$(I * G)[x,y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{r,g,b\}} I_c[x+a, y+b] \cdot G_c[a,b]$$

In this case, the size of the output will be $(1 + W - w) \times (1 + H - h)$, and we evaluate the convolution only within the image I. (For this problem we will not concern ourselves with how to compute the convolution along the boundary of the image.) To reduce the dimension of the output, we can do a strided convolution in which we shift the convolutional filter by s positions instead of a single position, along the image. The resulting output will have size $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$.

Write pseudocode to compute the convolution of an image I with a filter G and a stride of s.

Solution:

Note that the weights in a filter are shared across all the pixels in the input. For a convolutional network, we always use weight sharing because the same filter is applied across multiple positions of an input and because it reduces model complexity, which allows for far fewer parameters than using a fully connected network.

```
for x in {0,s,2s,...,W-w}
 for y in {0,s,2s,...,H-h}
  total = 0
  for c in {r,g,b}
   window = I_c[x:x+w,y:y+h]
   conv = window * G_c // * is element-wise multiplication
   total = total + summation(conv)
  out_c[x/s,y/s] = total
```

The operator $*$ is element-wise multiplication of the two matrices, and summation() is the sum of all elements in the matrix.

(b) Filters can be used to identify different types of features in an image such as edges or corners. Design a filter G that outputs a large (in magnitude) value for vertically oriented edges in image

I. By "edge," we mean a vertical line where a black rectangle borders a white rectangle. (We are not talking about a black line with white on both sides.)

Solution: An example vertical edge detector could look like

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \\ \vdots & \vdots \\ -1 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ & \vdots & \\ -1 & 0 & 1 \end{bmatrix}$$

This detector would return a large positive value for edges that go from low color intensity to high color intensity and will return a large negative value for edges that go from high color intensity to low color intensity. The height of the detector determines the length of the edge that it can detect.

(c) Although handcrafted filters can produce edge detectors and other useful features, convolutional networks *learn* filters via the backpropagation algorithm. These filters are often specific to the problem that we are solving. Learning these filters is a lot like learning weights in standard backpropagation, but because the same filter (with the same weights) is used in many different places, the chain rule is applied a little differently and we need to adjust the backpropagation algorithm accordingly. In short, during backpropagation each weight w in the filter has a partial derivative $\frac{\partial L}{\partial w}$ that receives contributions from every patch of image where w is applied.

Let L be the loss function or cost function our neural network is trying to minimize. Given the input image I, the convolution filter G, the convolution output $R = I * G$, and the partial derivative of the error with respect to each scalar in the output, $\frac{\partial L}{\partial R[i,j]}$, write an expression for the partial derivative of the loss with respect to a filter weight, $\frac{\partial L}{\partial G_c[x,y]}$, where $c \in \{r, g, b\}$.

Solution:

By the chain rule, we have

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} \frac{\partial R[i,j]}{\partial G_c[x,y]}.$$

From the equation for discrete convolution, the derivative for each entry $R[i,j]$ is

$$\begin{aligned} \frac{\partial R[i,j]}{\partial G_c[x,y]} &= \frac{\partial}{\partial G_c[x,y]} \sum_{c\in\{r,g,b\}} \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} I_c[i+a, j+b] \cdot G_c[a,b] \\ &= I_c[i+x, j+y]. \end{aligned}$$

When $x - i$ or $y - j$ go outside the boundary of the mask, we can treat the derivative as zero.

It follows that

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} I_c[i+x, j+y]. \tag{2}$$

(d) Sometimes, the output of a convolution can be large, and we might want to reduce the dimensions of the result. A common method to reduce the dimension of an image is called max pooling. This method works similar to convolution in that we have a filter that moves around the image, but instead of multiplying the filter with a subsection of the image, we take the maximum value in the subimage. Max pooling can also be thought of as downsampling the image but keeping the largest activations for each channel from the original input. To reduce the dimension of the output, we can do a strided max pooling in which we shift the max pooling filter by s positions instead of a single position, along the input. Given a filter size of $w \times h$, and a stride s, the output will be $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$ for an input image of size $W \times H$.

Let the output of a max pooling operation be an array R. Write a simple expression for element $R[i, j]$ of the output.

Solution:

$$R[i, j] = \max_{a=\{0,\ldots,w-1\}} \max_{b=\{0,\ldots,h-1\}} I_c[s * i + a, s * j + b].$$

(e) Explain how we can use the backprop algorithm to compute derivates through the max pooling operation. (A plain English answer will suffice; equations are optional.)

Solution: Similar to how we computed the derivatives through a convolution layer, we'll be given the derivative with respect to the output of the maxpool layer.

The gradient from the next layer is passed back only to the neuron which achieved the max. All other neurons get zero gradient.

Because maxpooling doesn't have any trainable parameters, we won't need to worry about calculating any derivatives for weights.

Once we have the derivative with respect to the input, the backprop algorithm can continue on to the layer before the maxpool by using this derivative.