BCB726

Lecture 3

October 27, 2025

Today

- Supervised learning with logistic regression + optimizing model parameters.
- Discriminative vs generative models
- Common pitfalls with supervised learning.
- If time: naive bayes classifier as a generative classification approach

Set up for classification problem

- Suppose we have a dataset, $\mathbf{X} \in \mathbb{R}^{N \times P}$, measuring p features across N data instances.
- We use $\mathbf{y} \in \mathbb{R}^{N \times 1}$ to denote the response variables for $\mathbf{X}$, such that y^i is the response for data instance i.
- Let's consider the case where each data instance has a binary label so that each $y^i \in \{0, 1\}$
- Our overarching objective is to build a *classifier*, which can accurately predict each data instance

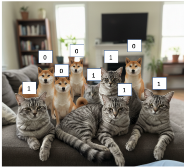


Figure: Imagine our task is to classify dogs from cats so that dogs have label '0' cats have label '1'

Generative vs discriminative classifiers

- Classification is effectively specifying a mapping, $f : X \rightarrow \mathbf{y}$ or accurately estimating $P(Y \mid X)$ between input data (X) and output labels, Y.
- **Generative classifier:** Estimate parameters of $P(X \mid Y)$ and $P(X)$ from training data. Use Bayes' rule to calculate $P(Y \mid X = \mathbf{x}^i)$
    - Reminder : Bayes' rule is $P(X \mid Y) = \frac{P(Y|X)P(X)}{P(Y)}$
- **Discriminative Classifier:** Model $P(Y \mid X)$ directly from the training data.

Generative vs discriminative illustrated

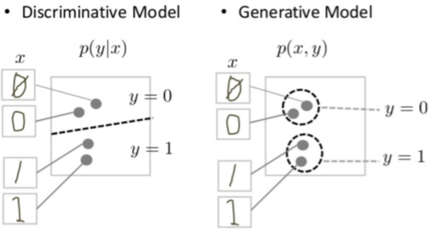


Figure: Fig. from `https://developers.google.com/machine-learning/gan/generative`

- The discriminative model tries to learn a decision boundary between 0s and 1s in the space. The decision boundary should be *drawn* to separate 0s and 1s as well as possible.
- The generative model tries to learn how to produce convincing 0s and 1s that look like the underlying data.

Learning with logistic regression

- Given a pair of input/output pairs $(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2) \dots (\mathbf{x}^N, y^N)$, we want to assign a prediction for a new data instance, $\mathbf{x}^i$ as $\hat{y}^i$
- We will build a linear model, comprised of our input data, $\mathbf{x}$ and a *learned* vector of weights reflecting the importance of each of the p input features, as $\mathbf{W} = [w_1, w_2, \dots, w_p]$.
- A high magnitude positive w indicates association with the '1' outcome and a high magnitude negative w indicates association with the '0' outcome

Computing a weighted sum of features and their importances to represent each data instance

For data instance, i, assuming we had learned our weights, $\mathbf{w} = [w_1, w_2, \ldots, w_p]$ and some bias term, b, we can compute a composite score to input into another function to output the probability of belonging to the '1' class. as,

$$z_i = \sum_{d=1}^{p} w_d x_d^i \tag{1}$$

- If this sum of very high, then we would like to use it to model a high probability of belonging to the '1' class.
- Limitation is that this is a number, not a probability, so we need a function that converts a sum to a probability

Formalizing sums to a probabilistic classifier

Given our computed zs, we seek a function that can ultimately tell us the following :

- $P(y = 1 \mid \mathbf{x}; \theta) \rightarrow$ Note that θ is general notation for the set of learned parameters (Ws and b)
- $P(y = 0 \mid \mathbf{x}; \theta)$

Logistic regression is an example of a **probabilistic classifier** because it predicts a class label, $\hat{y}$ by estimating the conditional probability $P(y \mid \mathbf{x})$.

Welcome sigmoid function

$$y^i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}} = \frac{1}{\exp(-z)} \tag{2}$$

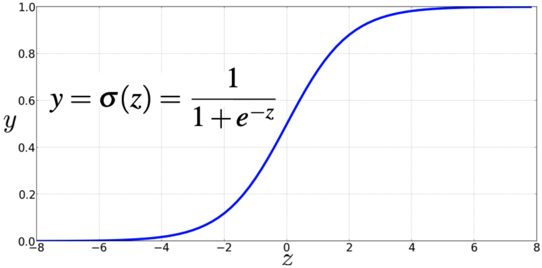


Figure: Sigmoid function converts positive and negative values to a value between 0 and 1.

Modeling probabilities with sigmoids

$$P(y^i = 1) = \sigma(z_i) = \frac{1}{1 + \exp -(\mathbf{w} \cdot \mathbf{x}^i + b)} \tag{3}$$

$$\begin{aligned} P(y^i = 0) &= 1 - \sigma(z_i) = 1 - \sigma(\mathbf{w} \cdot \mathbf{x}^i + b) \\ &= 1 - \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x}^i + b))} \\ &= \frac{\exp(-(\mathbf{w} \cdot \mathbf{x}^i + b))}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x}^i + b))} \end{aligned} \tag{4}$$

Probabilities to classifiers

$$\hat{y}^i = \begin{cases} 1 & \text{if } P(y^i = 1 \mid \mathbf{x}_i) > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

So, in this case 0.5 is our **decision boundary**

Optimizing the **w**s

- For each observation, i, we are predicting a label $\hat{y^i}$. We obviously want this to be in agreement with the true label, y^i.
- We will establish a cost or loss function to quantify divergence between y^i and $\hat{y^i}$ as $\mathcal{L}(y^i, \hat{y^i})$
- We are going to derive a loss function through the log likelihood

For a single observation, i, the likelihood can be expressed as

$$p(y^i \mid \mathbf{x}_i) = \hat{y^i}^{y^i}(1 - \hat{y^i})^{1-y^i} \tag{6}$$

Deriving cross-entropy from a likelihood

The likelihood for our single observation, i can be more formally written out as,

$$\text{Maximize :} \quad \begin{aligned} \log p(y^i \mid \mathbf{x_i}) &= \log\left[\hat{y^i}^{y^i}(1-\hat{y^i})^{1-y^i}\right] \\ &= y^i \log \hat{y^i} + (1-y^i)\log(1-\hat{y^i}) \end{aligned} \tag{7}$$

We can also consider minimizing this by multiplying by -1. This will give us a loss function instead as,

$$\text{Minimize :} \quad L_{\mathrm{CE}}(\hat{y^i}, y^i) = -\log p(y^i \mid \mathbf{x}_i) = -[y^i \log \hat{y^i} + (1-y^i)\log(1-\hat{y^i})] \tag{8}$$

Cross entropy continued

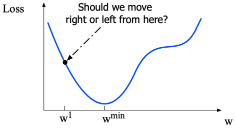
$$L_{\mathrm{CE}}(\hat{y^i}, y^i) = -[y^i \log \sigma(\mathbf{w} \cdot \mathbf{x}^i + b) + (1 - y^i) \log(1 - \sigma(\mathbf{w} \cdot \mathbf{x}^i + b))] \quad (9)$$

We want to do this optimization over all data N training data points points as,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{m} L_{\mathrm{CE}} \left(f \left(\mathbf{x}^{(i)}; \theta \right), y^{(i)} \right) \quad (10)$$

Setting up for gradient descent

We will use gradient descent to update a single scalar within our vector of **w**s. Recall a **gradient** is computed to give the direction of greatest increase in a function. So, for gradient descent, we want to move in the opposite direction.



[1]

Figure: We will be estimating the slope of the loss at a given point, such as w^1

[1] from `https://web.stanford.edu/~jurafsky/slp3/slides/5_LR_Apr_7_2021.pdf`

Using the gradient in optimization

- We will weight our gradient by a learning rate, η. So, high values of η means more dramatic moves in a particular direction.

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x; w), y) \tag{11}$$

For our **w**s where we have many weights, we are trying to understand how a change in an individual component (e.g. w_i) would influence the overall loss function. We can do this using partial derivatives as,

$$\nabla_\theta L(f(x; \theta), y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x; \theta), y) \\ \frac{\partial}{\partial w_2} L(f(x; \theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x; \theta), y) \end{bmatrix} \tag{12}$$

Partial derivatives for our logistic regression loss

The partial derivative for the *j*th weight component can be computed as,

$$\frac{\partial L_{\mathrm{CE}}(\hat{y}^i, y^i)}{\partial w_j} = [\sigma(w \cdot \mathbf{x} + b) - y]x_j \tag{13}$$

Example problem for logistic regression

- Suppose we want to predict whether a blood sample is from a **young** ($y = 0$) or **aged** ($y = 1$) donor based on proportions of two immune cell-types : T cells (x_1) and monocytes (x_2).
- Suppose we initialize our weights as $w_1 = w_2 = b = 0$.
- Imagine that our training example, i has $x_1^i = 0.25$ T cells, $x_2^i = 0.1$ monocytes and is from an aged donor ($y^i = 1$)

We will update our three parameters as follows,

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x; \theta), y) \tag{14}$$

Example continued

- $w_1 = w_2 = b = 0,\ x_1^i = 0.25,\ x_2^i = 0.1,\ y^i = 1$

$$\nabla_{w,b} = \begin{bmatrix} \frac{\partial L_{\mathrm{CE}}(\hat{y},y)}{\partial w_1} \\ \frac{\partial L_{\mathrm{C}(\hat{y},y)}}{\partial w} \\ \frac{\partial L_{\mathrm{CE}}(\hat{y},y)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w \cdot x + b) - y)x_1 \\ (\sigma(w \cdot x + b) - y)x_2 \\ \sigma(w \cdot x + b) - y \end{bmatrix} = \begin{bmatrix} (\sigma(0) - 1)x_1 \\ (\sigma(0) - 1)x_2 \\ \sigma(0) - 1 \end{bmatrix} = \begin{bmatrix} -0.5(0.25) \\ -0.5(0.1) \\ -0.5 \end{bmatrix} = \begin{bmatrix} -0.125 \\ -0.05 \\ -0.5 \end{bmatrix} \tag{15}$$

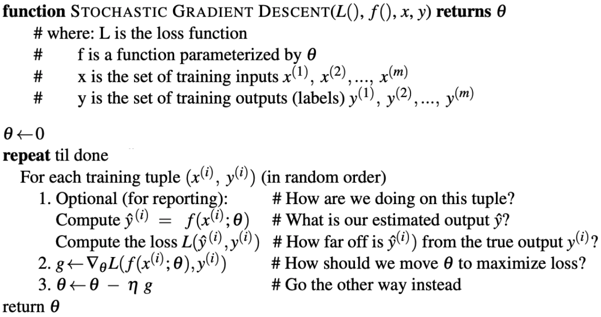
Imagining that we have a learning rate of $\eta = 0.1$

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta), y) \quad \eta = 0.1;$$

$$\theta^1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.1 \begin{bmatrix} -0.125 \\ -.05 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .0125 \\ .005 \\ .05 \end{bmatrix}$$

Learning efficiently with stochastic gradient descent

- Stochastic gradient descent updates parameters, leveraging data points in a random order.



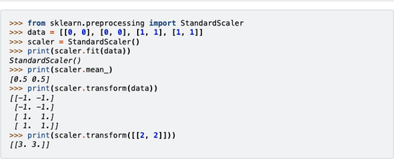
```
function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, $x$, $y$) returns $\theta$
      # where: L is the loss function
      #      f is a function parameterized by $\theta$
      #      x is the set of training inputs $x^{(1)}, x^{(2)}, ..., x^{(m)}$
      #      y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, ..., y^{(m)}$

$\theta \leftarrow 0$
repeat til done
   For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
      1. Optional (for reporting):           # How are we doing on this tuple?
         Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$      # What is our estimated output $\hat{y}$?
         Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$    # How far off is $\hat{y}^{(i)})$ from the true output $y^{(i)}$?
      2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$       # How should we move $\theta$ to maximize loss?
      3. $\theta \leftarrow \theta - \eta\ g$                        # Go the other way instead
return $\theta$
```

[2]

[2] pseudocode from https://web.stanford.edu/~jurafsky/slp3/slides/5_LR_Apr_7_2021.pdf

Common pitfall 1: data leakage

- Data leakage is when some of the test data was accidentally *seen in model training* → it's easier for this to happen than you may think!
    - **Data scaling** → if you do standard scaling (which requires estimating the mean and standard deviation of each feature) on the entire dataset before training a classifier, then you are technically encoding information about the distribution of datapoints in the test set.
    - What to do instead → fit you scalar separately in the training and test sets.



```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler()
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

[3]

[3] example implementation in python. Apply this separately in training and test sets.

Data leakage continued

- This is the most common mistake → if you have multiple samples from the same donor, you cannot have some of their instances in the training set and some of their instances in the test set.
- Example : let's say we are profiling immune cells in blood to predict whether donors were responders or non responders to some treatment. Let's say we get samples from each donor at multiple timepoints.
- If we were to split all of the samples randomly into train and test sets, then there would be instances from the same donor in both the training and test sets. This means that we have already seen information about 'test donors' in training.
- What to do instead → Split donors between training and test sets. Keep all instances from the same donor together.

Figure: Example: donors with many sample instances.

One last data leakage pitfall : double-dipping with feature selection

- Feature selection is another common pre-processing technique that is helpful for interpretability and model accuracy (more on this later)
- Feature selection should be done only using the training set
- **Less dire example**: Selecting features with high variance across data points should technically be done on the training set
- **Very dire example:** Let's say you applied logistic regression to the entire dataset, took only the features with high magnitude coefficients and trained a model only with that information. This would have double-dipped in the dataset.

Common issue: too many features

- This is not really a pitfall, as much as it is an artifact of working with modern biological datasets.
- If the number of measured features (p) is significantly larger than the number of profiled samples, N, then it is very easy for the model to overfit the training data. There is some combination of features that could explain each data point.
- The common solution is to **regularize** the loss function with a penalty term, which forces many coefficients to be 0.
- Lasso, ridge, and elastic net are common penalization approaches that are used.
- For example: The lasso penalty augments the loss function, $\mathcal{L}$ as $\mathcal{L} + \lambda \sum_{j=1}^{p} |\beta_j|$

Quick overview of a *generative* classifier: naive bayes

- **objective:** We seek to infer $P(X \mid Y)$, so, the probability of a class label, given measured features. Our $X = [X_1, X_2, \dots, X_n]$ is our collection of measured features. For now, let's assume that each X_i is one of discrete values.
- **assumption:** Each features, X_i, is conditionally independent of the Xs as $P(X_1, \dots, X_n \mid Y) = \prod_{i=1}^{n} P(X_i \mid Y)$
- How can we predict a discrete class label, Y given our measured features (specifically belonging to class y_k)?

Deriving naive bayes

$$P\left(Y = y_k \mid X_1 \ldots X_n\right) = \frac{P\left(Y = y_k\right) P\left(X_1 \ldots X_n \mid Y = y_k\right)}{\sum_j P\left(Y = y_j\right) P\left(X_1 \ldots X_n \mid Y = y_j\right)} \tag{16}$$

Then from our conditional independence assumption,

$$P\left(Y = y_k \mid X_1 \ldots X_n\right) = \frac{P\left(Y = y_k\right) \prod_i P\left(X_i \mid Y = y_k\right)}{\sum_j P\left(Y = y_j\right) \prod_i P\left(X_i \mid Y = y_j\right)} \tag{17}$$

The probability of each class is then . . .

$$Y \leftarrow \arg\max_{y_k} \frac{P\left(Y = y_k\right) \prod_i P\left(X_i \mid Y = y_k\right)}{\sum_j P\left(Y = y_j\right) \prod_i P\left(X_i \mid Y = y_j\right)} \tag{18}$$

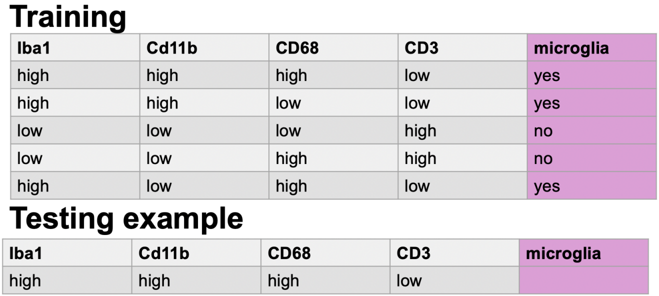
Training :

$$\hat{P}\left(X_i = x_i \mid Y = y\right) = \frac{(\ \#\ \text{training examples where } X_i = x_i \text{ and } Y = y)}{(\ \text{training examples where } Y = y)} \tag{19}$$

and,

$$\hat{P}(Y = y) = \frac{(\#\ \text{training examples where } Y = y\ )}{(\text{training examples})} \tag{20}$$

Using naive bayes to classify a cell as microglia or not



Training

Iba1	Cd11b	CD68	CD3	microglia
high	high	high	low	yes
high	high	low	low	yes
low	low	low	high	no
low	low	high	high	no
high	low	high	low	yes

Testing example

Iba1	Cd11b	CD68	CD3	microglia
high	high	high	low	

Figure: we can use the matrix of training examples to classify a new example by estimating combinations of X and Y.

Recap

- We covered logistic regression as an example of a supervised, discriminative machine learning approach
- We derived a cross-entropy loss and used gradient descent to optimize model parameters
- We learned the naive bayes algorithm as a generative classifier.
- We discussed common data leakage pitfalls that are easy to run in to!