CS281: Advanced ML **September 18, 2017**

Lecture 5: Linear Classification

Lecturer: Sasha Rush *Scribes: Demi Guo, Artidoro Pagnoni, Luke Melas-Kyriazi, Florian Berlinger*

5.1 Classification Introduction

Last time we saw linear regression. In linear regression we were predicting $y \in \mathbb{R}$, in classification instead we deal with a discrete set, for example $y \in \{0, 1\}$ or $y \in \{1, \ldots, C\}$. This distinction only matters for this lecture, starting from next class we will generalize the topics and treat them as the same thing.

Among the many applications, linear classification is used in sentiment analysis, spam detection, and facial and image recognition. We will use generative models of the data, which means that we will model both the x and the y explicitly, and we are not keeping x fixed. In the case of the spam filter earlier, x is the email body, and y is the label $\{\text{spam, not spam}\}$. A generative model of the email and labels, we would model the distribution of x, of the text in the email itself, and not only the distribution of the category y.

We will explore the basic method of Naïve Bayes in detail. Even with a very simple method like Naïve Bayes with basic features it is possible to perform extremely well on many classification tasks when large training data sets are available. For example, this simple model performs almost as well (one percent point difference) as very complex methods on spam detection.

5.2 Naïve Bayes

Note that the term "Bayes" in Naïve Bayes (NB) does not have to do with Bayesian modeling, or the presence of priors on parameters. We won't have any priors for the moment. General Naïve Bayes takes the following form:

$$\begin{aligned} y &\sim \text{Cat}(\pi) && \text{[class distribution]} \\ \boldsymbol{x}|y &\sim \prod_j p(x_j \mid y) && \text{[class conditional]} \end{aligned}$$

where y is the class label and comes from a categorical distribution, and x_j is a dimension of the input x.

In Naïve Bayes, the form of the class distribution is fixed and parametrized independently from the class conditional distribution. The "Naïve" term in "Naïve Bayes" precisely refers to the conditional independence between y and $x_j|y$. Depending of what the data looks like we can choose a different form for the class conditional distribution.

Here we present three possible choices for the class conditional distribution:

- **Multivariate Bernoulli Naïve Bayes**:

  $$x_j|y \sim \text{Bern}(\mu_{jc}) \quad \text{if } y = c$$

  Here y takes values in a set of classes, and μ_{jc} is a parameter associated with a specific feature (or dimension) in the input and a specific class. We use multivariate Bernoulli when we only allow two possible values for each feature, therefore $x_j|y$ follows a Bernoulli distribution.

  We can think of x as living in a hyper cube, with each dimension j having an associated μ for each class c. From here we get the name multivariate Bernoulli distribution.

- **Categorical Naïve Bayes**:

  $$x_j|y \sim \text{Cat}(\mu_{jc}) \quad \text{if } y = c$$

  We use the Categorical Naïve Bayes when we allow different classes for each feature j, so $x_j|y$ follows a Categorical distribution.

- **Multivariate Normal Naïve Bayes**

$$\boldsymbol{x}|y \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}^c_{diag})$$

Note that here we use $\boldsymbol{x}$ vector and not a specific feature. Since we impose that $\boldsymbol{\Sigma}^c$ is a diagonal matrix, we have no covariance between features, so this comes down to having an independent normal for each feature (or dimension) of the output. This is also required by the "Naïve" assumption of conditional independence. We would use MVN Naïve Bayes when the features take continuous values in $\mathbb{R}^n$.

5.3 General Naïve Bayes

We consider the data points $\{(\boldsymbol{x}_n, y_n)\}$, without specifying a particular generative model. The likelihood of each data point is:

$$p(\boldsymbol{x}_n, y_n|\text{param}) = p(y_n|\boldsymbol{\pi}) \prod_j p(x_{nj}|y_n, \text{param}) \tag{5.1}$$

$$= \prod_c \pi_c^{(y_n=c)} \prod_j \prod_c p(x_{nj}|y_n)^{(y_n=c)} \tag{5.2}$$

where in equation (5.2) we assume conditional independence (the "Naïve" assumption). The term $p(x_{nj}|y_n)$ depends on the generative model used for x and also on the class y_n.

We can then solve for the parameters maximizing the likelihood, which is equivalent to maximizing the log likelihood.

$$(\pi_{\text{MLE}}, \mu_{\text{MLE}}) = \underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_n \log p(\boldsymbol{x}_n, y_n|\text{param}) \tag{5.3}$$

$$= \underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_c N_c \log \pi_c + \sum_i \sum_c \sum_{n:y_n=c} p(x_{nj}|y_n) \tag{5.4}$$

$$= \left(\underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_c N_c \log \pi_c \right) + \left(\underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_i \sum_c \sum_{n:y_n=c} \log p(x_{nj}|y_n) \right) \tag{5.5}$$

Where $N_c = \sum_n \mathbb{1}(y_n = c)$, and $N =$ the number of data points.

This factors into two parts (5.10), the first only depending on $\boldsymbol{\pi}$ the other is the MLE for the class condition distribution on each feature or dimension of the input. This factorization allows to solve for the maximizing $\boldsymbol{\pi}$ and the maximizing parameters for the class conditional separately.

For example, if we use a Multivariate Bernoulli Naïve Bayes generative model we would get the following parameters from MLE:

$$\pi_c = \frac{N_c}{N} \tag{5.6}$$

$$\mu_{jc} = \frac{\sum_{n:y_n=c} x_{nj}}{N_c} = \frac{N_{cj}}{N_c} \tag{5.7}$$

Again, where $N_c = \sum_n \mathbb{1}(y_n = c)$, $N_{cj} = \sum_n \mathbb{1}(y_n = c)x_{nj}$ and $N =$ number of data points.

5.4 Bayesian Naive Bayes: Add a Prior

Here, instead of working with a single distribution, we are working with multiple distributions. For simplicity, let's use the following factored **prior**:

$$p(\pi, \mu) = p(\pi) \prod_j \prod_c p(\mu_{jc})$$

where $p(\pi)$ represents the prior on class distribution and $\prod_j \prod_c p(\mu_{jc})$ represents prior on class conditional distribution.

Now, **what prior should we use?**

1. π: Dirichlet (goes with Categorical)
2. μ_{jc}:
   (a) Beta (goes with Bernoulli)
   (b) Dirichlet (goes with Categorical)
   (c) Normal (goes with Normal)
   (d) Inverse-Wishart (Iw) (goes with Normal)

   Here, what distribution we choose depends on our choice of class conditional distribution.

Recall that we want to use conjugate priors to have a natural update (that's why we pair them up!). By using conjugate priors, we will have:

$$p(\pi|\text{data}) = Dir(N_1 + \alpha_1, \cdots, N_c + \alpha_c)$$

$$p(\mu_{jc}|\text{data}) = Beta((N_c - N_{jc}) + \beta_0, N_c + \beta_1)$$

5.4.1 Intuition

You can think of the α_i above as initial pseudocounts. Those pseudocounts give nonzero probability to features we haven't seen before, which is crucial for NLP. For unseen features, you could have a pseudocount of 1 or 0.5 (Laplace term) or something.
Because of this property, a Bayesian model helps prevents overfitting by introducing such priors: consider the spam email classification problem mentioned before. Say the word "subject" (call it feature j) always occurs in both classes ("spam" and "not spam"), so we estimate $\hat{\theta}_{jc} = 1$ (we overfit!) What will happen if we encounter a new email which does not have this word in it? Our algorithm will crash and burn! This is another manifestation of the black swan paradox discussed in Book Section 3.3.4.1. Note that this will not happen if we introduce pseudocounts to all features!

5.5 Posterior Predictive

Exercise 5.1. Find the posterior predictive density of the Bayesian Naive Bayes model. Assume that the following factored prior is used:

$$p(\pi, \mu) = p(\pi) \prod_j \prod_c p(\mu_{jc})$$

Proof. The posterior predictive density is given by

$$p(y = c \mid x, \text{data}) \propto p(y = c \mid \text{data}) \prod_j p(x_j \mid y = c, \text{ data})$$

On integrating out unknown parameters,

$$= \int \text{Cat}(y = c \mid \pi) p(\pi \mid \text{data}) d\pi \prod_j \int \text{Ber}(x_j = c \mid y, \mu_{jc}) p(\mu_{jc} \mid \text{data}) d\mu_{jc}$$

More succinctly, the density is

$$= {\pi_c}^{\text{MAP}} \prod_j ({\mu_{jc}}^{\text{MAP}})^{\mathbb{I}(x_j=1)} (1 - {\mu_{jc}}^{\text{MAP}})^{\mathbb{I}(x_j=0)}$$

where

$$\pi_c{}^{\text{MAP}} = \frac{N_c + \alpha_c}{N + \sum_c \alpha_c} (\text{Dirichlet MAP})$$

$$\mu_{jc}{}^{\text{MAP}} = \frac{N_{jc} + \beta_1}{N_c + \beta_1 + \beta_0} (\text{Beta MAP})$$

□

5.6 More on Predictive

Now, let's consider a little bit more about what's happening in our predictive. Consider the email spam classification problem: given some features of an email, we want to predict if the email is a spam or not a spam. We have:

$$\begin{aligned}
p(y = c|x, data) &\propto \pi_c \prod_j p(x_j|y) \text{ (try to generate observations from class)} \\
&= \pi_c \prod_j \mu_{jc}^{x_j} (1 - \mu_{jc})^{(1-x_j)} \text{ (informal parametrization)} \\
&= \exp(\log \pi_c + \sum_j x_j \log \mu_{jc} + (1 - x_j) \log(1 - \mu_{jc})) \text{ (take exp of log)} \\
&= \exp\left(\log \pi_c + \sum_j \log(1 - \mu_{jc}) + \sum_j x_j \log \frac{\mu_{jc}}{1 - \mu_{jc}}\right)
\end{aligned}$$

where the first two term $\log \pi_c + \sum_j \log(1 - \mu_{jc})$ is a constant (we call it b for bias), and the last term $\sum_j x_j \log \frac{\mu_{jc}}{1-\mu_{jc}}$ is linear (we call it θ).

5.7 Multivariate Bernoulli Naive Bayes

For Multivariate Bernoulli NB, we will have:

$$\theta_{jc} = \log \frac{\mu_{jc}}{1 - \mu_{jc}}$$

$$b_c = \log \pi_c + \log(1 - \mu_{jc})$$

So, we have:

$$p(y = c \mid x) \propto \exp(\theta_c^T x + b_c)$$

Thus, in order to determine which class ("spam" or "not spam"), for each class we simply compute a linear function with respect to x, and compare the two. Our $\theta x + b$ is going to be associated with a linear separator of the data. Even better, for prediction, we can simply compute θ and β (as shown above) using closed form for both MAP and MLE cases.

5.8 The Sigmoid Function

Before proceeding, we should name our variables to speak about them more easily.

We call μ the "informal parameters" and θ the "scores." In the case of a Multivariate Bernouli model, we have the map $\theta_{jc} = \log \frac{\mu_{jc}}{1-\mu_{jc}}$, which we call the "log odds." We may also invert this relationship to find μ as a function of θ:

$$\theta = \log \frac{\mu}{1 - \mu} \implies \mu = \frac{e^\theta}{1 + e^\theta} = \frac{1}{1 + e^{-\theta}} = \sigma(\theta)$$

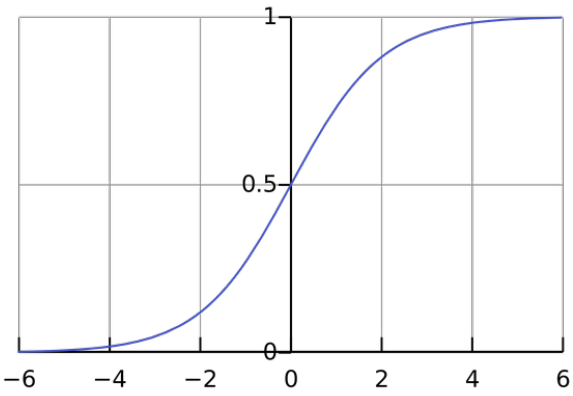

Figure 5.1: The Sigmoid Function

We denote this function $\sigma(\theta)$ as the sigmoid function. The sigmoid function is a map from the real line to the interval $[0, 1]$, so is useful as a representation of probability. It is also a common building block in constructing neural networks, as we will see later in the course.

5.9 The Softmax Function

We will now return to the predictive $p(y = c|x) \propto \exp(\theta_c^T x + c_c)$ to try to compute the normalizer Z:

$$p(y = c|x) = \frac{1}{Z} \exp(\theta_c^T x + b_c)$$

In general, we can compute the normal by summing over all our classes.

$$Z(\theta) = \sum_{c'} = \exp(\theta_{c'}^T x + b_{c'})$$

In practice, this summation is often computationally expensive. However, it is not necessary to compute this sum if we are only interested in the most likely class label given an input.

We call the resulting probability density function the *softmax*:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{i'} \exp(z_{i'})}$$

This function generalizes the sigmoid function to multiple classes/dimensions. We call it the "softmax"

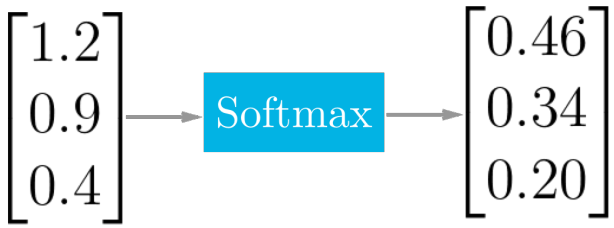

Figure 5.2: An example of the Softmax in action: higher probabilities assigned to higher weights.

because we may think of it as a smooth, differentiable version of the function which simply returns 1 for the most likely class (or argmax).

Often, when computing the log-likelihood for instance, the log of the normalization constant

$$\log(Z(\theta)) = \log \sum_{c'} \exp(\theta_{c'}^T x + b_{c'})$$

becomes a cause of much upset: it leads to wrong answers due to overflow and underflow. Intuitively, flows occur due to the very nature of the exponential function. Thus, while $e^x \approx [e, 7.39, 20.09]$, for $\boldsymbol{x} = [1, 2, 3]$, a typical program will return $e^x \approx [0, inf, inf]$, for $\boldsymbol{x} = [-1000, 1000, 2000]$. As one might guess, the inf's are caused due to a stack overflow.
The log-sum-exp trick comes to the rescue. Recall the basic property that

$$\log \sum_{n=1}^{N} \exp\{x_n\} = a + \log \sum_{n=1}^{N} \exp\{x_n - a\}, \forall a \in \mathbb{R}.$$

so it is indeed worthwhile to subtract a constant from each weight x, calculate the exponential without worry, take the log of the resultant sum, and then add back the constant. Note that there is no rule that defines a particular value for a; it should nevertheless be something sensible such as $a = \max_n x_n$.

5.10 Discriminative Classification

We may apply the mathematical tools developed in the generative classification setting discussed above to perform discriminative classification. In discriminative classification, we assume that our inputs x are fixed, rather than coming from some probability distribution.
We take the maximum likelihood estimate, as in linear regression, given that $p(y = c|x) \propto \exp(\theta_c^T x)$:

$$\text{MLE} : \operatorname*{argmax}_{\theta} p(y|x, \theta) = \operatorname*{argmax}_{\theta} \sum_{n} \log \text{softmax}(\theta_c^T x_n) c_n$$

What are the advantages and disadvantages of this approach? The primary disadvantage compared to methods we have seen earlier is that this maximum likelihood estimate has *no closed form*. It is also not clear how we might incorporate our prior (although there is recent work in this area). On the other hand, this equation is convex and it is easy (at least mathematically, not necessarily computationally) to compute gradients, so we may use gradient descent.

$$\frac{d(\cdot)}{d\theta_c} = \sum_n x_n \cdot \begin{cases} 1 - \text{softmax}(\theta_c^T x) & \text{if } y_n = c \\ \text{softmax}(\theta_c^T x) & \text{otherwise} \end{cases} \tag{5.8}$$

This model is known as **logistic regression** (even though it is used for classification, not regression) and is widely used in practice.

More Resources on Optimization

- Convex Optimization by Lieven Vandenberghe and Stephen P. Boyd

Exercise 5.2. Write code for a logistic regression model to classify images from the modified NIST (mnist) dataset found here: mnist.

Listing 1: Solution for Exercise 5.2

```
import numpy as np

import torch
from torch.autograd import Variable
from torch import optim
```

```
from data_util import load_mnist


def build_model(input_dim, output_dim):
    # We don't need the softmax layer here since CrossEntropyLoss already
    # uses it internally.
    model = torch.nn.Sequential()
    model.add_module("linear",
                     torch.nn.Linear(input_dim, output_dim, bias=False))
    return model


def train(model, loss, optimizer, x_val, y_val):
    x = Variable(x_val, requires_grad=False)
    y = Variable(y_val, requires_grad=False)

    # Reset gradient
    optimizer.zero_grad()

    # Forward
    fx = model.forward(x)
    output = loss.forward(fx, y)

    # Backward
    output.backward()

    # Update parameters
    optimizer.step()

    return output.data[0]


def predict(model, x_val):
    x = Variable(x_val, requires_grad=False)
    output = model.forward(x)
    return output.data.numpy().argmax(axis=1)


def main():
    torch.manual_seed(42)
    trX, teX, trY, teY = load_mnist(onehot=False)
    trX = torch.from_numpy(trX).float()
    teX = torch.from_numpy(teX).float()
    trY = torch.from_numpy(trY).long()

    n_examples, n_features = trX.size()
    n_classes = 10
    model = build_model(n_features, n_classes)
    loss = torch.nn.CrossEntropyLoss(size_average=True)
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
    batch_size = 100

    for i in range(100):
        cost = 0.
        num_batches = n_examples // batch_size
        for k in range(num_batches):
```

```
            start, end = k * batch_size, (k + 1) * batch_size
            cost += train(model, loss, optimizer,
                          trX[start:end], trY[start:end])
        predY = predict(model, teX)
        print("Epoch %d, cost = %f, acc = %.2f%%"
              % (i + 1, cost / num_batches, 100. * np.mean(predY == teY)))


if __name__ == "__main__":
    main()
```