Foundations of AI **Fall 2025**
Professor Eric Ewing[1] **Discrete Optimization Unit**

Logistic Regression and Entropy

1 Introduction: From Regression to Classification

So far, we've mostly studied regression problems where our goal is to predict continuous values, like total ice cream sales. In these cases, our output y can take on any real value within some range. The K-NN algorithm can be used for classification or regression, but is a non-parametric model and doesn't require as much background as we will cover today.

Today, we shift our focus to classification problems, where instead of predicting a continuous value, we want to predict which discrete category or class an input belongs to. In the simplest case, called binary classification, we have exactly two possible outcomes, typically given as 0 and 1.

In email spam detection, we want to classify each incoming email as either spam or not spam. In medical diagnosis, we might want to determine whether a patient has a particular disease or not based on their test results and symptoms. In image recognition, we might classify images as containing either a cat or a dog. For credit card applications, a bank needs to decide whether to approve or deny each application.

1.1 Why Not Linear Regression for Classification?

Given that we've just learned about linear regression, you might wonder: can't we just use linear regression for classification problems? Let's consider binary classification where our target $y \in \{0, 1\}$. We might be tempted to use our familiar linear model:

$$f(x) = w^T x$$

However, this approach has several serious problems. First, linear regression can produce predictions that are less than 0 or greater than 1. If we want to interpret our predictions as probabilities (which is natural for classification), these values don't make sense. A probability must lie in the range $[0, 1]$.

Second, linear regression is extremely sensitive to outliers in classification settings. Imagine we're classifying tumors as malignant (1) or benign (0) based on tumor size. If we have a few training examples with very large tumors, these outliers will pull our linear decision boundary in ways that hurt performance on typical examples. The squared error loss we used in linear regression gives these outliers disproportionate influence.

Third, there's no natural probabilistic interpretation. When our linear model predicts 0.3 or 1.7, what does that mean? We need predictions that we can interpret as the probability that an example belongs to the positive class.

What we really need is a function that takes any real-valued input (like $w^T x$) and squashes it into the range $[0, 1]$, giving us valid probabilities. This is exactly what the sigmoid function does.

[1]This notes were compiled in conjunction with Professor Amy Greenwald.

2 The Sigmoid Function

The sigmoid function, also called the logistic function, is defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

This function solves all the problems we identified with using linear regression for classification. Let's examine its key properties to understand why it's so well-suited for our purposes.

2.1 Properties of the Sigmoid

The most important property of the sigmoid is its range. No matter what value we input, the sigmoid always outputs a value strictly between 0 and 1 (though it approaches these bounds asymptotically). Any value z that we put in, will always map to a number between 0 and 1.

The sigmoid is monotonically increasing, meaning that larger inputs always produce larger outputs. This preserves the ordering from our linear combination $\theta^T x$: if we believe one example is more likely to be in the positive class than another based on the raw linear combination, the sigmoid will preserve this ordering.

The function has a symmetry property: $\sigma(-z) = 1 - \sigma(z)$. This means the sigmoid is symmetric around the point $(0, 0.5)$. When our linear combination $w^T x = 0$, the sigmoid outputs exactly 0.5, which we can interpret as maximum uncertainty between the two classes.

The sigmoid is smooth and differentiable everywhere, which is crucial for gradient-based optimization. We'll see shortly that its derivative has a particularly easy form.

2.2 Derivative of the Sigmoid

Let's derive the derivative. Starting with the definition:

$$\frac{d\sigma(z)}{dz} = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right)$$

We can rewrite this as $(1+e^{-z})^{-1}$ and apply the chain rule:

$$\begin{aligned} &= -(1+e^{-z})^{-2} \cdot (-e^{-z}) \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \end{aligned}$$

We can factor this expression:

$$\begin{aligned} &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\ &= \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}} \\ &= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) \\ &= \sigma(z) \cdot (1-\sigma(z)) \end{aligned}$$

The derivative of the sigmoid function is:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

This is not just surprising how simple it is, but it's also computationally convenient. Once we've computed $\sigma(z)$, we can compute its derivative using just one multiplication and one subtraction, without needing to re-evaluate any exponentials.

3 Logistic Regression Model

3.1 Our model

Now we can define our logistic regression model. Our hypothesis function is simply the sigmoid applied to our familiar linear combination:

$$f(x) = \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}}$$

The key insight is how we interpret this output. We interpret $f(x)$ as the probability that $y = 1$ given the input x and parameters θ. In other words:

$$f(x) = P(y = 1|x; w)$$

Since there are only two classes in binary classification, we also know that:

$$P(y = 0|x; w) = 1 - f(x)$$

For example, if $f(x) = 0.7$, we interpret this as saying there's a 70% probability this example belongs to the positive class and a 30% probability it belongs to the negative class.

3.2 Decision Boundary

In practice, we need to make a hard decision: should we classify an example as positive or negative? The standard approach is to predict $y = 1$ if our probability estimate is at least 0.5, and predict $y = 0$ otherwise. That is, we predict $y = 1$ when:

$$f(x) \geq 0.5$$

Because the sigmoid outputs 0.5 exactly when its input is 0, this is equivalent to:

$$w^T x \geq 0$$

This inequality defines what we call the decision boundary. In two dimensions with features x_1 and x_2, if we have $w = [w_0, w_1, w_2]^T$, our decision boundary is the line:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

This is a linear decision boundary, which is why logistic regression is called a linear classifier despite using the nonlinear sigmoid function. The sigmoid provides probabilistic outputs and smooth gradients, but the actual boundary between classes is still a straight line (or hyperplane in higher dimensions).

Points on one side of this boundary are classified as positive, and points on the other side are classified as negative. The further a point is from the boundary, the more confident we are in its classification, as reflected by probabilities closer to 0 or 1.

4 Understanding Entropy

4.1 Motivation: Measuring Information

Suppose you're waiting for the result of a coin flip, and someone is about to tell you the outcome. How much information do you gain when you learn whether it came up heads or tails? Intuitively, the answer depends on the coin itself.

If the coin is perfectly fair with probability $p = 0.5$ for heads, there's high uncertainty before the flip. You genuinely don't know what the outcome will be. When you learn the result, you gain significant information because the outcome was truly unpredictable.

On the other hand, if the coin is heavily biased with $p = 0.99$ for heads, there's very little uncertainty before the flip. You can be pretty confident it will come up heads. When you learn the result (which is almost certainly heads), you gain very little information because you could have guessed the outcome with high confidence.

In the extreme case where $p = 1.0$ (the coin always comes up heads), there's no uncertainty whatsoever. Learning the outcome gives you exactly zero information because you already knew what would happen.

We want a mathematical function that captures this intuition: a measure of uncertainty that's high for fair coins and low for biased coins.

4.2 Entropy: Derivation from First Principles

Let's consider a coin that comes up heads with probability p. We want to design a function $H(p)$ that measures the uncertainty or unpredictability of this coin. What properties should such a function have?

First, it should be continuous. Small changes in the probability p should cause small changes in our uncertainty measure $H(p)$. We don't want abrupt jumps as we make the coin slightly more biased.

Second, it should be monotonic in uncertainty. The uncertainty should be maximum when $p = 0.5$, because a fair coin is the most unpredictable. As we move away from 0.5 in either direction, making the coin more biased toward heads or tails, the uncertainty should decrease.

Third, it should be symmetric. Our measure of uncertainty should be the same whether we have a coin with p probability of heads or a coin with p probability of tails. Mathematically, we want $H(p) = H(1 - p)$. The labels "heads" and "tails" are arbitrary.

Fourth, boundary conditions: when $p = 0$ or $p = 1$, the coin is deterministic. There's no uncertainty, so $H(0) = H(1) = 0$.

Finally, it should be additive for independent events. If we flip two independent coins, the total information should be the sum of the information from each coin. This property is crucial and leads us to the logarithm.

4.3 Information Content of a Single Outcome

Let's start by thinking about a single outcome. When an event with probability p occurs, we want to quantify how "surprising" or informative this outcome is. We define the information content as:

$$I(p) = -\log p$$

Why does this make sense? Consider the extreme cases. If $p = 1$ (the event was certain to occur), then

$I(1) = -\log 1 = 0$. There's no surprise when something certain happens, so we gain zero information. On the other hand, if p is very small (the event was unlikely), then $-\log p$ is large. Rare events are surprising and informative.

The logarithm also gives us the additive property we want. If two independent events occur with probabilities p_1 and p_2, the probability of both occurring is $p_1 \cdot p_2$. The information from both events is:

$$I(p_1 p_2) = -\log(p_1 p_2) = -\log p_1 - \log p_2 = I(p_1) + I(p_2)$$

Information from independent events adds up, which matches our intuition.

The base of the logarithm determines our unit of measurement. If we use base 2, we measure information in bits. If we use the natural logarithm (base e), we measure in nats. For most machine learning applications, we use the natural logarithm, though the choice doesn't affect the optimization since it's just a scaling factor.

4.4 Expected Information: Entropy

Now we can define entropy as the expected information content. Before we flip our coin, we don't know which outcome will occur. But we can compute the average information we expect to gain:

For a coin with $P(\text{heads}) = p$, the entropy is:

$$\begin{aligned} H(p) &= \mathbb{E}[\text{Information}] \\ &= p \cdot I(p) + (1-p) \cdot I(1-p) \\ &= p \cdot (-\log p) + (1-p) \cdot (-\log(1-p)) \\ &= -p \log p - (1-p) \log(1-p) \end{aligned}$$

This is called the binary entropy function. Let's verify it has the properties we wanted. By convention, we define $0 \log 0 = 0$ (which is consistent with the limit as $p \to 0$).

4.5 Properties and Examples

Let's compute some examples using base 2 logarithms to get intuition. For a fair coin with $p = 0.5$:

$$H(0.5) = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = -0.5 \cdot (-1) - 0.5 \cdot (-1) = 1 \text{ bit}$$

Note, that the units of Entropy are *bits* (when $\log_2$ is used). We need exactly 1 bit of information to describe the outcome of a fair coin flip. This matches our intuition that a bit can represent exactly one binary choice.

For a biased coin with $p = 0.9$:

$$H(0.9) = -0.9 \log_2(0.9) - 0.1 \log_2(0.1) \approx 0.47 \text{ bits}$$

The uncertainty is lower than for a fair coin, so we need less than 1 bit on average to describe the outcome. This makes sense: we could use a coding scheme where we use a short code for heads (which happens 90% of the time) and a longer code for tails.

For a deterministic coin with $p = 1.0$:

$$H(1.0) = -1 \cdot \log_2(1) - 0 \cdot \log_2(0) = 0 \text{ bits}$$

No information is needed because there's no uncertainty.

The maximum entropy occurs at $p = 0.5$, giving us 1 bit. The minimum entropy occurs at the boundaries $p = 0$ or $p = 1$, giving us 0 bits. The function is symmetric around $p = 0.5$, as we desired. And entropy is always non-negative, which makes sense since we can't have negative uncertainty.

One way to interpret entropy is as the average number of yes/no questions you'd need to ask to determine the outcome, using an optimal questioning strategy. A fair coin requires 1 question on average. A biased coin requires fewer questions on average because you can make good guesses.

4.6 Cross-Entropy

Now we can understand cross-entropy, which is central to training our logistic regression model. Suppose we have two probability distributions:

The true distribution p represents reality. In classification, this is determined by the actual labels in our dataset. For a single training example, p is 1 for the correct class and 0 for the incorrect class.

The predicted distribution q represents our model's beliefs. In logistic regression, this is given by $f(x)$ for the positive class and $1 - f(x)$ for the negative class.

Cross-entropy measures what happens when we use the wrong distribution q to encode outcomes that actually come from distribution p. Mathematically:

$$H(p, q) = -\mathbb{E}_{x\sim p}[\log q(x)] = -\sum_i p_i \log q_i$$

For binary classification with true label $y \in \{0, 1\}$ and prediction $\hat{y} = f(x)$, this becomes:

$$H(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Let's understand what this formula is doing. If the true label is $y = 1$, the formula reduces to $-\log \hat{y}$. This is large when $\hat{y}$ is small (we predicted low probability but the true class was positive). If the true label is $y = 0$, the formula reduces to $-\log(1 - \hat{y})$, which is large when $\hat{y}$ is large (we predicted high probability but the true class was negative).

There are several key insights about cross-entropy. First, if our predictions perfectly match reality ($q = p$), then the cross-entropy equals the entropy: $H(p, q) = H(p)$. This is the best we can do.

Second, if our predictions differ from reality ($q \neq p$), then the cross-entropy is always larger than the entropy: $H(p, q) > H(p)$. The difference $H(p, q) - H(p)$ is called the Kullback-Leibler divergence, and it measures how much extra information we need because we're using the wrong distribution.

Third, cross-entropy heavily penalizes confident wrong predictions. If the true label is $y = 1$ but we predict $\hat{y} \approx 0$, then $-\log \hat{y}$ approaches infinity. We pay a huge penalty for being confidently wrong. On the other hand, if we're uncertain (predict $\hat{y} \approx 0.5$), the penalty is moderate.

Finally, and most importantly, minimizing cross-entropy is exactly equivalent to maximizing the log-likelihood of the correct labels. This is why cross-entropy emerges naturally as the right loss function for classification.

5 Cross-Entropy Loss

Now we can derive the loss function for logistic regression from first principles using maximum likelihood estimation. This will show us why cross-entropy is not just a reasonable choice, but the principled choice for classification.

5.1 Likelihood Approach

For a single training example $(x^{(i)}, y^{(i)})$ where $y^{(i)} \in \{0, 1\}$, we want to write down the probability of observing this label given our parameters. If $y^{(i)} = 1$, the probability is simply $f(x^{(i)})$. If $y^{(i)} = 0$, the probability is $1 - f(x^{(i)})$.

We can write this compactly as a single expression:

$$P(y^{(i)}|x^{(i)}; w) = f(x^{(i)})^{y^{(i)}} \cdot (1 - f(x^{(i)}))^{1-y^{(i)}}$$

Let's verify this works. If $y^{(i)} = 1$, then we get $f(x^{(i)})^1 \cdot (1 - f(x^{(i)}))^0 = f(x^{(i)})$, which is correct. If $y^{(i)} = 0$, we get $f(x^{(i)})^0 \cdot (1 - f(x^{(i)}))^1 = 1 - f(x^{(i)})$, also correct.

5.2 Log-Likelihood

For our entire dataset of m training examples, assuming each example is independent (which is a standard assumption), the likelihood of observing all the labels is:

$$\mathcal{L}(w) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}; w)$$

This is a product of many probabilities, each less than or equal to 1, so the likelihood will be a very small number. More importantly, products are numerically unstable and difficult to optimize.

The standard trick is to maximize the log-likelihood instead of the likelihood. Since the logarithm is a monotonically increasing function, maximizing the log-likelihood is equivalent to maximizing the likelihood. But the log has several advantages: it converts products to sums, it's more numerically stable, and it's easier to differentiate.

Taking the logarithm:

$$\begin{aligned}
\log \mathcal{L}(w) &= \log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}; w) \\
&= \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}; w) \\
&= \sum_{i=1}^{m} \left[y^{(i)} \log f(x^{(i)}) + (1 - y^{(i)}) \log(1 - f(x^{(i)})) \right]
\end{aligned}$$

5.3 Cross-Entropy Loss Function

In machine learning, we typically talk about minimizing a loss (or cost) rather than maximizing a likelihood. These are equivalent: minimizing the negative log-likelihood is the same as maximizing the log-likelihood.

We also typically compute the average per example rather than the sum, which doesn't change the optimization but gives us a loss that doesn't grow with dataset size. This gives us our final cross-entropy loss function:

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log f(x^{(i)}) + (1-y^{(i)})\log(1-f(x^{(i)}))\right]$$

This is called the cross-entropy loss, binary cross-entropy, or log loss. Notice that this is exactly the average cross-entropy between the true labels and predicted probabilities that we derived in the previous section.

This loss function has excellent properties for optimization. Unlike squared error with the sigmoid, it is convex in the parameters w, meaning gradient descent will converge to the global minimum. It also naturally handles the probabilistic interpretation we want for classification.

6 Gradient Descent for Logistic Regression

To minimize our loss function $J(w)$, we need to compute its gradient with respect to the parameters. We'll use the chain rule, leveraging the nice derivative of the sigmoid function we derived earlier.

Let's compute the partial derivative with respect to parameter w_j. We need to work through the chain rule carefully:

$$\frac{\partial J(w)}{\partial w_j} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\frac{1}{f(x^{(i)})} - (1-y^{(i)})\frac{1}{1-f(x^{(i)})}\right]\frac{\partial f(x^{(i)})}{\partial w_j}$$

Now we use the fact that $\frac{\partial f(x)}{\partial w_j} = f(x)(1-f(x))\cdot x_j$ (applying our sigmoid derivative):

$$= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{y^{(i)}}{f(x^{(i)})} - \frac{1-y^{(i)}}{1-f(x^{(i)})}\right] f(x^{(i)})(1-f(x^{(i)}))x_j^{(i)}$$

After simplification (multiplying through), we get:

$$\begin{aligned}
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}(1-f(x^{(i)})) - (1-y^{(i)})f(x^{(i)})\right]x_j^{(i)} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} - f(x^{(i)})\right]x_j^{(i)} \\
&= \frac{1}{m}\sum_{i=1}^{m}(f(x^{(i)}) - y^{(i)})x_j^{(i)}
\end{aligned}$$

6.1 Update Rule

The gradient descent update rule is:

$$w_j := w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(f(x^{(i)}) - y^{(i)})x_j^{(i)}$$

where α is the learning rate, a hyperparameter that controls how large a step we take in the direction of the negative gradient.

In practice, we repeat this update for all parameters $w_0, w_1, \ldots, w_n$ simultaneously until convergence. Convergence means that the loss $J(w)$ stops decreasing significantly, indicating we've found (at least) a local minimum. Because our loss function is convex, this local minimum is actually the global minimum.

The interpretation of each update is intuitive. For each training example, we compute the error $(f(x^{(i)})-y^{(i)})$, which is positive if we over-predicted and negative if we under-predicted. We adjust each parameter w_j proportionally to this error and the corresponding feature value $x_j^{(i)}$. Features that are present (large x_j) get larger updates, while absent features (small or zero x_j) get smaller updates.