Linear prediction: features, overfitting, and losses

CS 446 / ECE 449

2022-01-27 09:54:15 -0600 (a6ec9a6)
Plan for today

- Features.
- Overfitting.
- Loss construction and multiclass output.
Recall: solution to Old Faithful example.

1. Form pairs \( x_i := [b_i - a_i] \), and matrix

\[
X := \begin{bmatrix}
\leftarrow x_1^T & \rightarrow \\
\vdots & \vdots \\
\leftarrow x_n^T & \rightarrow
\end{bmatrix} = \begin{bmatrix}
b_1 - a_1 & 1 \\
\vdots & \vdots \\
b_n - a_n & 1
\end{bmatrix} \in \mathbb{R}^{n \times 2}.
\]

Form labels \( y \in \mathbb{R}^n \), \( y_i := a_{i+1} - b_i \).

2. Choose OLS solution \( \hat{w}_{ols} := X^+ y \).
Recall: solution to Old Faithful example.

1. Form pairs $x_i := [b_i - a_i]$, and matrix

\[
X := \begin{bmatrix}
\leftarrow & x_1^T & \rightarrow \\
\vdots & \ddots & \vdots \\
\leftarrow & x_n^T & \rightarrow
\end{bmatrix} = \begin{bmatrix}
b_1 - a_1 & 1 \\
\vdots & \ddots & \vdots \\
b_n - a_n & 1
\end{bmatrix} \in \mathbb{R}^{n \times 2}.
\]

Form labels $y \in \mathbb{R}^n$, $y_i := a_{i+1} - b_i$.

2. Choose OLS solution $\hat{w}_{\text{ols}} := X^+ y$.

We can view $x_i$ as the output of a feature mapping $\phi$:

\[x_i := \phi(a_i, b_i) := \left[ \begin{array}{c} b_i - a_i \\ 1 \end{array} \right].\]

Feature mappings are an easy way to make simple methods like linear predictors more powerful.

Appending 1 is very common:

instead of the linear rule $x \mapsto 2 \cdot 1[w^T x \geq 0] - 1$,

we have the affine decision rule $x \mapsto 2 \cdot 1[w^T x \geq -b] - 1$. 
"pytorch meta-algorithm".
1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

As part of step 1:
choose $\phi : \mathbb{R}^d \to \mathbb{R}^p$ and replace $x$ with $\phi(x)$ everywhere.
“pytorch meta-algorithm”.

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

As part of step 1:
choose $\phi : \mathbb{R}^d \to \mathbb{R}^p$ and replace $x$ with $\phi(x)$ everywhere.

**Remark.** Sometimes called feature engineering.
It still exists, despite promises of deep learning.
What if data are not vectors: $\mathbf{x} \in \mathcal{X} \neq \mathbb{R}^d$?

**Common example:** $\mathcal{X}$ is the set of English words. Two approaches:

1. **Bag-of-words:** pick some ordering on $\mathcal{X}$, and map the $i^{th}$ word to $e_i \in \mathbb{R}^{|\mathcal{X}|}$.
   Can encode documents as normalized word counts.

2. **Word2vec:** magic embedding $\mathcal{X} \rightarrow \mathbb{R}^d$ with $d \ll |\mathcal{X}|$ using deep networks.

Both are still heavily used. Both have many nuances (e.g., “compressing” $\mathcal{X}$ so that have/having/had/etc all have the same mapping).
Suppose $\phi(x) = (1, x, x^2)$, and $w = (a, b, c)$. Then

$$x \mapsto w^T \phi(x) = a + bx + cx^2,$$

and our linear learning tools can now learn quadratic polynomials!
Suppose $\phi(x) = (1, x, x^2)$, and $w = (a, b, c)$. Then

$$x \mapsto w^T \phi(x) = a + bx + cx^2,$$

and our linear learning tools can now learn quadratic polynomials!

**Multivariate case:** define

$$\phi(x) := (1, x_1, \ldots, x_d, x_1^2, \ldots, x_d^2, x_1 x_2, \ldots, x_1 x_d, \ldots, x_{d-1} x_d).$$

**Note:** hw1 has a coding problem with polynomial expansion, and the order there is slightly different.
Consider XOR: \( x \in (\pm 1, \pm 1) \), and \( y = x_1 x_2 \).

Not linearly separable with provided features, but linearly separable with quadratic features!
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

Feature expansion makes step 5 easy: given $x$, predict using $\phi(x)$, where

$$\phi(x) = \begin{cases} \exists x_i \in \text{training data}: x_i = x, \\ 0 \quad \text{otherwise} \end{cases}$$

What about step 6?
"pytorch meta-algorithm".

5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.
Overfitting

“pytorch meta-algorithm”.

5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

Feature expansion makes step 5 easy:
given $x$, predict using $\phi(x)$, where

$$
\phi(x) = \begin{cases} 
  e_i & \text{exists } x_i = x \text{ in the training data}, \\
  0 & \text{otherwise}.
\end{cases}
$$

What about step 6?
Fitting polynomials to a noisy constant function

**Data:** $(x, y)$ has $x \sim \text{Uniform}([0, 1])$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

**Predictor:** polynomials of varying degree, $x \mapsto w^T(1, x, x^2, \ldots, x^r)$.

**Method:** OLS solution $\hat{w}_{\text{ols}}$. 
Fitting polynomials to a noisy constant function

**Data:** \((x, y)\) has \(x \sim \text{Uniform}(0, 1)\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

**Predictor:** polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

**Method:** OLS solution \(\hat{w}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_2 = 0.00690718, R_2 = 0.00870077
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** \((x, y)\) has \(x \sim \text{Uniform}([0, 1])\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

**Predictor:** polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

**Method:** OLS solution \(\hat{w}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_3 = 0.0064944, R_3 = 0.00998528
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** $(x, y)$ has $x \sim \text{Uniform}(0, 1)$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

**Predictor:** polynomials of varying degree, $x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)$.

**Method:** OLS solution $\hat{\mathbf{w}}_{\text{ols}}$.

(Training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** $(x, y)$ has $x \sim \text{Uniform}([0, 1])$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

**Predictor:** polynomials of varying degree, $x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)$.

**Method:** OLS solution $\hat{\mathbf{w}}_{\text{ols}}$.

- Training data is red, testing data is green.

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_5 = 0.00582684, R_5 = 0.00975194
\]
Fitting polynomials to a noisy constant function

Data: \((x, y)\) has \(x \sim \text{Uniform}(0, 1)\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

Predictor: polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

Method: OLS solution \(\hat{w}_{ols}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_6 = 0.00571136, R_6 = 0.0142185
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** $(x, y)$ has $x \sim \text{Uniform}(0, 1)$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

**Predictor:** polynomials of varying degree, $x \mapsto w^T(1, x, x^2, \ldots, x^r)$.

**Method:** OLS solution $\hat{w}_{\text{ols}}$.

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** $(x, y)$ has $x \sim \text{Uniform}(0, 1)$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

**Predictor:** polynomials of varying degree, $x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)$.

**Method:** OLS solution $\hat{\mathbf{w}}_{\text{ols}}$.

(\text{training data is red, testing data is green.})
Fitting polynomials to a noisy constant function

Data: $(x, y)$ has $x \sim \text{Uniform}([0, 1])$ and $y \sim \mathcal{N}(0, 1)$ (independent!).

Predictor: polynomials of varying degree, $x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)$.

Method: OLS solution $\hat{\mathbf{w}}_{\text{ols}}$.

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

Data: \((x, y)\) has \(x \sim \text{Uniform}(0, 1)\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

Predictor: polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

Method: OLS solution \(\hat{w}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_{11} = 0.00513565, R_{11} = 9.33301
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** \((x, y)\) has \(x \sim \text{Uniform}([0, 1])\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

**Predictor:** polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

**Method:** OLS solution \(\hat{w}_{ols}\).

(Training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** \((x, y)\) has \(x \sim \text{Uniform}([0, 1])\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

**Predictor:** polynomials of varying degree, \(x \mapsto w^T (1, x, x^2, \ldots, x^r)\).

**Method:** OLS solution \(\hat{w}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_{13} = 0.00503326, R_{13} = 30.7593
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

**Data:** \((x, y)\) has \(x \sim \text{Uniform}(0, 1)\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

**Predictor:** polynomials of varying degree, \(x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)\).

**Method:** OLS solution \(\hat{\mathbf{w}}_{ols}\).

\[
R_1 = 0.00692304, \quad R_1 = 0.00897457, \quad R_{14} = 0.00444016, \quad R_{14} = 985.645
\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

Data: \((x, y)\) has \(x \sim \text{Uniform}([0, 1])\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

Predictor: polynomials of varying degree, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

Method: OLS solution \(\hat{w}_{\text{ols}}\).

\[\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_{15} = 0.00361723, R_{15} = 13158.5\]

(training data is red, testing data is green.)
Fitting polynomials to a noisy constant function

Data: \((x, y)\) has \(x \sim \text{Uniform}([0, 1])\) and \(y \sim \mathcal{N}(0, 1)\) (independent!).

Predictor: polynomials of varying degree, \(x \mapsto \mathbf{w}^T(1, x, x^2, \ldots, x^r)\).

Method: OLS solution \(\hat{\mathbf{w}}_{\text{ols}}\).

(training data is red, testing data is green.)
What happened to our test error?

**Degree 1:** training error 0.0069, testing error 0.00897.

**Degree 16:** training error 0.0031, testing error 3.09106.

Overfitting is when training error is good, but testing error is bad. Often it means a model is complicated in a way that is incompatible with the data.

Step 6 of pytorch meta-algorithm is typically about tuning model complexity.

▶ Sometimes this is called a bias/variance tradeoff: the bias (observed training error) is good, but the variance (random test error) is bad.

▶ It is not true that "complex models overfit" (deep networks are often fine).

The situation is not well-understood.
What happened to our test error?

**Degree 1**: training error 0.0069, testing error 0.00897.
**Degree 16**: training error 0.0031, testing error 30910.6.

**Overfitting** is when training error is good, but testing error is bad.
What happened to our test error?

**Degree 1**: training error 0.0069, testing error 0.00897.
**Degree 16**: training error 0.0031, testing error 30910.6.

**Overfitting** is when training error is good, but testing error is bad.

- Often it means a model is complicated in a way that is incompatible with the data. Step 6 of pytorch meta-algorithm is typically about tuning model complexity.
- Sometimes this is called a bias/variance tradeoff: the bias (observed training error) is good, but the variance (random test error) is bad.
- It is not true that “complex models overfit” (deep networks are often fine). The situation is not well-understood.
Reduce model complexity via **regularized ERM**: pick $\lambda > 0$, and approximately solve

$$\min_{\mathbf{w} \in \mathbb{R}^d} \hat{R}(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2.$$ 

$\lambda\|\mathbf{w}\|_2^2$ is **regularization**, and this is just one choice.
One approach: regularization

Reduce model complexity via regularized ERM: pick $\lambda > 0$, and approximately solve

$$\min_{w \in \mathbb{R}^d} \mathcal{R}(w) + \frac{\lambda}{2} \|w\|_2^2.$$  

$\lambda \|w\|_2^2$ is regularization, and this is just one choice.

- For least squares, it has a nice form: the solution becomes

$$\left( X^T X + \lambda n I \right)^{-1} X^T y,$$

where this inverse always exists (why?). (It has a name: “ridge regression”.)

- This perspective has lost favor in deep networks, as they can generalize well even if weight norms are large, and regularization doesn’t change their test error much.

- $\lambda$ is a hyper-parameter; needs to be found some other way.

- Sometimes this specific regularizer is called weight decay:

$$\nabla_w \left( \mathcal{R}(w) + \frac{\lambda}{2} \|w\|_2^2 \right) = \nabla_w \mathcal{R}(w) + \lambda w.$$
Loss functions and multiclass prediction

So far we have seen:

\[ \ell_{\text{sq}}(z) = \frac{1}{2} (1 - z)^2, \]  
(squared loss)

\[ \ell_{\text{logistic}}(z) = \ln(1 + \exp(-z)), \]  
(logistic loss)

Questions:

▶ Are there other choices?
▶ What about other output spaces, e.g., multiclass?

Remark:
We introduced squared loss as \( \ell_{\text{sq}}(\hat{y}, y) = (y - \hat{y})^2 / 2 \).
This makes sense for regression; for classification, combining arguments into a single \( z := \hat{y}y \) suffices.
Loss functions and multiclass prediction

So far we have seen:

\[ \ell_{\text{sq}}(z) = \frac{1}{2}(1 - z)^2, \]  
  (squared loss),

\[ \ell_{\text{logistic}}(z) = \ln(1 + \exp(-z)), \]  
  (logistic loss).

Questions:
▶ Are there other choices?
▶ What about other output spaces, e.g., multiclass?

Remark:
We introduced squared loss as \[ \ell_{\text{sq}}(\hat{y}, y) = \frac{(y - \hat{y})^2}{2}. \]  
This makes sense for regression; for classification, combining arguments into a single \( z := \hat{y}y \) suffices.
So far we have seen:

\[ \ell_{\text{sq}}(z) = \frac{1}{2} (1 - z)^2, \] (squared loss),

\[ \ell_{\text{logistic}}(z) = \ln(1 + \exp(-z)), \] (logistic loss).

Questions:

▶ Are there other choices?
▶ What about other output spaces, e.g., multiclass?
So far we have seen:

\[ \ell_{\text{sq}}(z) = \frac{1}{2}(1 - z)^2, \quad \text{(squared loss)}, \]
\[ \ell_{\text{logistic}}(z) = \ln(1 + \exp(-z)), \quad \text{(logistic loss)}. \]

**Questions:**
- Are there other choices?
- What about other output spaces, e.g., multiclass?

**Remark:** We introduced squared loss as \( \ell_{\text{sq}}(\hat{y}, y) = (y - \hat{y})^2/2. \n\)
This makes sense for regression; for classification, combining arguments into a single
\[ z := \hat{y}y \] suffices.
Standard classification losses

- **hinge**
  \[ \ell_{\text{hinge}}(z) = \max\{0, 1 - z\} \]

- **squared**
  \[ 2\ell_{\text{sq}}(z) = (1 - z)^2 \]

- **logistic**
  \[ \ell_{\text{logistic}}(z) / \ln(2) = \ln(1 + e^{-z}) / \ln(2) \]

- **exponential**
  \[ \ell_{\text{exp}}(z) = e^{-z} \]
Standard classification losses

\[ \ell_{\text{hinge}}(z) = \max\{0, 1 - z\} \]
\[ 2\ell_{\text{sq}}(z) = (1 - z)^2 \]
\[ \ell_{\text{logistic}}(z) / \ln(2) = \ln(1 + e^{-z}) / \ln(2) \]
\[ \ell_{\text{exp}}(z) = e^{-z} \]

- Upper bound zero-one loss: \( \ell(z) \to 0 \) implies \( 1[z \leq 0] \to 0 \).
  Sometimes called \textit{convex surrogate losses} (for the zero-one loss).
- Differentiable and have \( \ell'(0) < 0 \): helps GD.
Designing losses with maximum likelihood

Stronger goal than classification:
suppose our parameters $w$ define a conditional model $p_w(y|x)$.

Note: can use these losses even if true label probabilities have a different model.
Designing losses with maximum likelihood

Stronger goal than classification: suppose our parameters $w$ define a conditional model $p_w(y|x)$.

Maximum likelihood (MLE) approach to model training: given $((x_i, y_i))_{i=1}^n$, choose $w$ via

$$\arg \max_w \prod_{i=1}^n p_w(y_i|x_i) = \arg \min_w -\ln \prod_{i=1}^n p_w(y_i|x_i) = \arg \min_w \sum_{i=1}^n \ln \frac{1}{p_w(y_i|x_i)}.$$
Designing losses with maximum likelihood

Stronger goal than classification: suppose our parameters $w$ define a conditional model $p_w(y|x)$.

Maximum likelihood (MLE) approach to model training: given $((x_i, y_i))_{i=1}^n$, choose $w$ via

$$
\text{arg max}_w \prod_{i=1}^n p_w(y_i|x_i) = \text{arg min}_w \sum_{i=1}^n \ln \frac{1}{p_w(y_i|x_i)}.
$$

Looks like ERM with loss $\ln \frac{1}{p_w(y|x)}$!

**Note:** can use these losses even if true label probabilities have a different model.
Squared loss via MLE

Define conditional model $p_\omega$ using standard Gaussian with mean $x^T \omega$:

$$p_\omega(y|x) = \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{(y - x^T \omega)^2}{2} \right).$$
Define conditional model $p_w$ using standard Gaussian with mean $x^T w$:

$$p_w(y|x) = \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{(y - x^T w)^2}{2} \right).$$

Then

$$\ln \frac{1}{p_w(y|x)} = \frac{1}{2} \left( y - x^T w \right)^2 + \frac{1}{2} \ln(2\pi),$$

and

$$\arg \min_w \sum_{i=1}^{n} \ln \frac{1}{p_w(y_i|x_i)} = \arg \min_w \sum_{i=1}^{n} \ell_{sq}(x_i^T w, y_i)$$

$$= \arg \min_w \frac{1}{n} \sum_{i=1}^{n} \ell_{sq}(x_i^T w, y_i).$$
Define conditional model

\[ p_w(y = 1|x) = \frac{1}{1 + \exp(-x^T w)}. \]

whereby

\[ p_w(y = -1|x) = 1 - p_w(y = 1|x) = \frac{\exp(-x^T w)}{1 + \exp(-x^T w)} = \frac{1}{1 + \exp(x^T w)}. \]
Logistic loss via MLE

Define conditional model

\[ p_w(y = 1|x) = \frac{1}{1 + \exp(-x^TW)} . \]

whereby

\[ p_w(y = -1|x) = 1 - p_w(y = 1|x) = \frac{\exp(-x^TW)}{1 + \exp(-x^TW)} = \frac{1}{1 + \exp(x^TW)} . \]

Then

\[ \ln \frac{1}{p_w(y|x)} = \ln \frac{1}{p_w(y = 1|x)(1+y)/2(1 - p_w(y = 1|x))(1-y)/2} \]
\[ = \frac{1+y}{2} \ln(1 + \exp(-x^TW)) + \frac{(1-y)}{2} \ln(1 + \exp(x^TW)) \]
\[ = \ln(1 + \exp(-y x^TW)) , \]

and

\[ \arg \min_w \sum_{i=1}^n \ln \frac{1}{p_w(y_i|x_i)} = \arg \min \frac{1}{n} \sum_{i=1}^n \ell_{\text{logistic}}(y_i, x_i^TW) . \]
**Multiclass classification**

**Setup:** given $x$ predict $y \in \{1, \ldots k\}$.

**Two approaches:**

- **One-vs-all** (a reduction approach): train $k$ binary classifiers, each with $	ilde{y}_i := 2 \cdot 1[y_i = j] - 1$. Predict by choosing the largest output.
- Directly train a model $p(y = j|x)$ with maximum likelihood.
Cross-entropy loss (multi-class logistic loss) via MLE

**Conditional model:** given predictor \( f : \mathbb{R}^d \rightarrow \mathbb{R}^k \), model \( p_f(\hat{y} = j \mid x) \propto \exp(f(x)) \), which means

\[
p_f(\hat{y} = j \mid x) = \frac{\exp(f(x)_j)}{\sum_{i=1}^{k} \exp(f(x)_i)}.
\]

Corresponding **cross-entropy loss** \( \ell_{ce} \): given true label \( y \in \{1, \ldots, k\} \),

\[
\ell_{ce}(f(x), y) := \ln 1 \frac{\exp(f(x)_y)}{\sum_{i=1}^{k} \exp(f(x)_i)} = -f(x)_y + \ln k \sum_{i=1}^{k} \exp(f(x)_i).
\]
Cross-entropy loss (multi-class logistic loss) via MLE

**Conditional model:** given predictor $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$, model $p_f(\hat{y} = j|\mathbf{x}) \propto \exp(f(x))$, which means $p_f(\hat{y} = j|\mathbf{x}) = \frac{\exp(f(x)_j)}{\sum_{i=1}^k \exp(f(x)_i)}$.

Corresponding **cross-entropy loss** $\ell_{ce}$: given true label $y \in \{1, \ldots, k\}$,

$$
\ln \frac{1}{p_f(\hat{y} = y|\mathbf{x})} = \ln \frac{\sum_{j=1}^k \exp(f(x)_j)}{\exp(f(x))_y} = -f(x)_y + \ln \sum_{j=1}^k \exp(f(x)_j) =: \ell_{ce}(f(x), y).
$$
Cross-entropy loss (multi-class logistic loss) via MLE

**Conditional model:** given predictor \( f : \mathbb{R}^d \rightarrow \mathbb{R}^k \), model \( p_f(\hat{y} = \cdot|\mathbf{x}) \propto \exp(f(\mathbf{x})) \), which means

\[
p_f(\hat{y} = j|\mathbf{x}) = \frac{\exp(f(\mathbf{x})_j)}{\sum_{i=1}^k \exp(f(\mathbf{x})_i)}.
\]

Corresponding **cross-entropy loss** \( \ell_{ce} \): given true label \( y \in \{1, \ldots, k\} \),

\[
\ln \frac{1}{p_f(\hat{y} = y|\mathbf{x})} = \ln \frac{\sum_{j=1}^k \exp(f(\mathbf{x})_j)}{\exp(f(\mathbf{x}))_y} = -f(\mathbf{x})_y + \ln \sum_{j=1}^k \exp(f(\mathbf{x})_j)) =: \ell_{ce}(f(\mathbf{x}), y).
\]

**Notes.**

In pytorch, this is `torch.nn.CrossEntropyLoss()(f(x), y)`. Loss is minimized when

\[
p_f(\hat{y} = y|\mathbf{x}) \approx 1.
\]

Name comes from the cross-entropy expression

\[
\sum_{j=1}^k (e_{y})_j \ln \frac{1}{p_f(\hat{y} = j|\mathbf{x})}.
\]
Features.
Overfitting.
Loss construction and multiclass output.
FPR/TPR, confusion matrix, ROC/AUC curve, cost-sensitive losses. KL and cross-entropy.
Multivariate prediction: regression case

Suppose we have $k$ labels for each example:

$$Y := \begin{bmatrix} y_1^T \rightarrow \\ \vdots \\ y_n^T \rightarrow \end{bmatrix}.$$ 

It’s natural to also learn $W \in \mathbb{R}^{d \times k}$:

$$\arg \min_{W \in \mathbb{R}^{d \times k}} \|XW - Y\|_F^2.$$
Suppose we have $k$ labels for each example:

$$Y := \begin{bmatrix} \leftarrow y_1^T \rightarrow \\ \vdots \\ \leftarrow y_n^T \rightarrow \end{bmatrix}.$$ 

It’s natural to also learn $W \in \mathbb{R}^{d \times k}$:

$$\arg \min_{W \in \mathbb{R}^{d \times k}} \|XW - Y\|_F^2.$$ 

This is the same as $k$ regular linear regressions: given $W$,

$$\|XW - Y\|_F^2 = \sum_{j=1}^n \|XW:j - Y:j\|_2^2 = \sum_{j=1}^n \sum_{i=1}^n (X_{i:j}^T W:j - y_{i:j})^2.$$ 

We have reduced multivariate regression to univariate regression:

$$\min_{W \in \mathbb{R}^{d \times k}} \|XW - Y\|_F^2 = \sum_{j=1}^k \min_{w \in \mathbb{R}^d} \|Xw - Y:j\|_2^2.$$
Supplemental reading

- Shalev-Shwartz/Ben-David: chapters 9, 24.