Linear prediction: features, overfitting, and losses
Plan for today

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

1. Form pairs $x_i := \begin{bmatrix} b_i - a_i \end{bmatrix}$, and matrix

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

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 $x_i := \begin{bmatrix} b_i - a_i \\ 1 \end{bmatrix}$, and matrix

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

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

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

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

$$x_i := \phi(a_i, b_i) := \begin{bmatrix} b_i - a_i \\ 1 \end{bmatrix}.$$ 

- 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 $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$ and replace $x$ with $\phi(x)$ everywhere.

Remark. Sometimes this is called feature engineering, and viewed as (a) tedious, (b) replaced by deep learning. Feature engineering still exists, however, and is also used to interpret deep learning.
“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 $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$ and replace $x$ with $\phi(x)$ everywhere.

Remark. Sometimes this is called feature engineering, and viewed as (a) tedious, (b) replaced by deep learning.
Feature engineering still exists, however, and is also used to interpret deep learning.
1. Non-linear transformations of existing variables: for $x \in \mathbb{R}$,

$$\phi(x) = \ln(1 + x).$$

2. Logical formula of binary variables: for $x = (x_1, \ldots, x_d) \in \{0, 1\}^d$,

$$\phi(x) = (x_1 \land x_5 \land \neg x_{10}) \lor (\neg x_2 \land x_7).$$

3. Trigonometric expansion: for $x \in \mathbb{R}$,

$$\phi(x) = (1, \sin(x), \cos(x), \sin(2x), \cos(2x), \ldots).$$

4. Polynomial expansion: for $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$,

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

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

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

- linear terms
- squared terms
- cross terms
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!
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: pick

$$\phi(\mathbf{x}) = \begin{cases} e_i & \text{exists } \mathbf{x}_i = \mathbf{x}, \\ 0 & \text{otherwise.} \end{cases}$$

What about step 6?
**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 $\hat{\mathbf{w}}_{\text{ols}}$. 
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 \(\hat{\mathbf{w}}_{\text{ols}}\).

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

(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 $\hat{\mathbf{w}}_{\text{ols}}$.

$$\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_4 = 0.0062397, R_4 = 0.013063$$

(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, \(x \mapsto \mathbf{w}^\top (1, x, x^2, \ldots, x^r)\).

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

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_5 = 0.00582684, R_5 = 0.00975194
\]

(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 $\hat{\mathbf{w}}_{\text{ols}}$.

\[
\hat{R}_1 = 0.00692304, \ R_1 = 0.00897457, \hat{R}_7 = 0.00565266, \ R_7 = 0.0282631
\]

(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 \(\hat{\mathbf{w}}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, \quad R_1 = 0.00897457, \quad \hat{R}_8 = 0.00564127, \quad R_8 = 0.0440347
\]

(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 \(\hat{w}_{\text{ols}}\).

\[ \hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_9 = 0.00541878, R_9 = 0.42463 \]

(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 $\hat{\mathbf{w}}_{\text{ols}}$.

![Graph showing the fitting results with $\hat{R}_1 = 0.00692304$, $R_1 = 0.00897457$, $\hat{R}_{11} = 0.00513565$, $R_{11} = 9.33301$](image)

*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 \(\hat{w}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, R_1 = 0.00897457, \hat{R}_{12} = 0.00503499, R_{12} = 40.6796
\]

(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 \(\hat{\mathbf{w}}_{\text{ols}}\).

\[
\hat{R}_1 = 0.00692304, \quad R_1 = 0.00897457, \quad \hat{R}_{14} = 0.00444016, \quad R_{14} = 985.645
\]

(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, \(x \mapsto w^T(1, x, x^2, \ldots, x^r)\).

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

\[
\begin{align*}
\hat{R}_1 &= 0.00692304, \\
R_1 &= 0.00897457, \\
\hat{R}_{16} &= 0.00312787, \\
R_{16} &= 30910.6
\end{align*}
\]

(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 30910.6.
What happened to our test error?

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

**Degree 16:** training error 0.0031, testing error 3091.6.

**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.
- 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 as simple as “complex models are bad”. Deep networks are complex but can fail to overfit. One explanation is they and the training algorithms have a magical bias which works well with many standard prediction tasks.
- We will discuss this topic extensively in a few lectures. Generally step 6 of the “pytorch meta-algorithm” is what makes things hard.
One approach: regularization

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

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

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

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

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

$\lambda\|w\|^2_2$ 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 is 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:
  $$\nabla_w \left( \hat{R}(w) + \frac{\lambda}{2} \|w\|^2 \right) = \nabla_w \hat{R}(w) + \lambda w.$$
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 
\[ \ell_{\text{sq}}(\hat{y}, y) = (y - \hat{y})^2. \]
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:

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

<table>
<thead>
<tr>
<th>Loss</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>hinge</td>
<td>$\ell_{\text{hinge}}(z) = \max{0, 1 - z}$</td>
</tr>
<tr>
<td>squared</td>
<td>$2\ell_{\text{sq}}(z) = (1 - z)^2$</td>
</tr>
<tr>
<td>logistic</td>
<td>$\ell_{\text{logistic}}(z) / \ln(2) = \ln(1 + e^{-z}) / \ln(2)$</td>
</tr>
<tr>
<td>exponential</td>
<td>$\ell_{\text{exp}}(z) = e^{-z}$</td>
</tr>
</tbody>
</table>

**Upper bound zero-one loss:**

$\ell(z) \to 0$ implies $1[z \leq 0] \to 0$.

Sometimes called convex surrogate losses (for the zero-one loss).

Differentiable and have $\ell'(0) < 0$: helps GD.
Standard classification losses

- Upper bound zero-one loss: $\ell(z) \to 0$ implies $\mathbb{1}[z \leq 0] \to 0$.
  Sometimes called convex surrogate losses (for the zero-one loss).

- Differentiable and have $\ell'(0) < 0$: helps GD.

<table>
<thead>
<tr>
<th>Loss</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>hinge</td>
<td>$\ell_{\text{hinge}}(z) = \max{0, 1 - z}$</td>
</tr>
<tr>
<td>squared</td>
<td>$2\ell_{\text{sq}}(z) = (1 - z)^2$</td>
</tr>
<tr>
<td>logistic</td>
<td>$\ell_{\text{logistic}}(z)/\ln(2) = \ln(1 + e^{-z})/\ln(2)$</td>
</tr>
<tr>
<td>exponential</td>
<td>$\ell_{\text{exp}}(z) = e^{-z}$</td>
</tr>
</tbody>
</table>
Let’s treat parameters $w$ as giving a conditional model $p_w(y|x)$.

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

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

Note: using these loss functions is still useful even if the model is wrong.
Let’s treat parameters $w$ as giving a conditional model $p_w(y|x)$.

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

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

**Note:** using these loss functions is still useful even if the model is wrong.
Squared loss via MLE

Define conditional model

\[ p_w(y|x) = \text{standard Gaussian with mean } x^T w. \]
Squared loss via MLE

Define conditional model

\[ p_w(y|x) = \text{standard Gaussian with mean } x^\top w. \]

Then

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

and

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

\[ = \arg \min_w \frac{1}{n} \sum_{i=1}^n \ell_{sq}(x_i^\top w, y_i). \]
Logistic loss via MLE

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

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

and

\[
\arg \min_w \ln \prod_{i=1}^{n} \frac{1}{p_w(y_i|x_i)} = \arg \min_w \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{logistic}}(y_i x_i^Tw).
\]
Multivariate prediction: regression case

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 $W \in \mathbb{R}^{d \times k}$:

\[
\operatorname{arg\ min}_{W \in \mathbb{R}^{d \times k}} \|XW - Y\|_F^2.
\]
Multivariate prediction: regression case

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 $W$,

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

We have reduced multivariate regression to univariate regression:

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

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

**Two approaches:**

- **One-vs-all** (a reduction approach): train $k$ binary classifiers, each with \( \tilde{y}_i := 2 \cdot 1[y_i = j] - 1 \). Predict by choosing the largest output.

- Directly train a model $p(y = j | \mathbf{x})$ with maximum likelihood.
Cross-entropy loss (multi-class logistic loss)

Start with the maximum likelihood approach:
Given \( \tilde{y}, \hat{y} \in \Delta_k = \{ p \in \mathbb{R}^k : p \geq 0, \sum_i p_i = 1 \} \) (probability simplex), loss

\[
\sum_{j=1}^{k} \tilde{y}_j \ln \frac{1}{\hat{y}_j}.
\]

Given \( y \in \{1, \ldots, k\} \), pick \( \tilde{y} := e_y \).
Cross-entropy loss (multi-class logistic loss)

Start with the maximum likelihood approach:
Given \( \tilde{y}, \hat{y} \in \Delta_k = \{ p \in \mathbb{R}^k : p \geq 0, \sum_i p_i = 1 \} \) (probability simplex), loss

\[
\sum_{j=1}^{k} \tilde{y}_j \ln \frac{1}{\hat{y}_j}.
\]

Given \( y \in \{1, \ldots, k\} \), pick \( \tilde{y} := e_y \).

Given predictor \( f : \mathbb{R}^d \to \mathbb{R}^k \), define softmax model \( \hat{y} \propto \exp(f(\mathbf{x})) \), thus

\[
\sum_{j=1}^{k} \tilde{y}_j \ln \frac{1}{pf(\hat{y} = j|\mathbf{x})} = \sum_{j=1}^{k} \tilde{y}_j \ln \frac{\sum_{r=1}^{k} \exp(f(\mathbf{x})_r)}{\exp(f(\mathbf{x}))_j}
\]

\[
= -f(\mathbf{x})_y + \ln \sum_{r=1}^{k} \exp (f(\mathbf{x})_r) .
\]

In pytorch, this is torch.nn.CrossEntropyLoss()(f(x), y).
The cross-entropy loss prefers that we put all probability on \( y \).
Summary for today

- Features and overfitting.
- Loss construction and multiclass output.
(Appendix.)
FPR/TPR, confusion matrix, ROC/AUC curve, cost-sensitive losses. KL and cross-entropy.
Supplemental reading

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