Linear prediction: features, overfitting, and losses

CS 446 / ECE 449

2022-02-04 23:03:41 -0600 (`1fc7e22`)

Plan for today

- Features.
- Overfitting.
- Loss construction and multiclass output.

Recall: solution to Old Faithful example.

1. Form pairs $\boldsymbol{x}_i := \left[\begin{smallmatrix} b_i - a_i \\ 1 \end{smallmatrix}\right]$, and matrix

$$\boldsymbol{X} := \begin{bmatrix} \leftarrow & \boldsymbol{x}_1^\mathsf{T} & \rightarrow \\ & \vdots & \\ \leftarrow & \boldsymbol{x}_n^\mathsf{T} & \rightarrow \end{bmatrix} = \begin{bmatrix} b_1 - a_1 & 1 \\ \vdots & \\ b_n - a_n & 1 \end{bmatrix} \in \mathbb{R}^{n \times 2}.$$

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

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

Recall: solution to Old Faithful example.

1. Form pairs $\boldsymbol{x}_i := \left[\begin{smallmatrix} b_i - a_i \\ 1 \end{smallmatrix}\right]$, and matrix

$$\boldsymbol{X} := \begin{bmatrix} \leftarrow & \boldsymbol{x}_1^\mathsf{T} & \rightarrow \\ & \vdots & \\ \leftarrow & \boldsymbol{x}_n^\mathsf{T} & \rightarrow \end{bmatrix} = \begin{bmatrix} b_1 - a_1 & 1 \\ & \vdots \\ b_n - a_n & 1 \end{bmatrix} \in \mathbb{R}^{n \times 2}.$$

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

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

- We can view $\boldsymbol{x}_i$ as the output of a feature mapping ϕ:

$$\boldsymbol{x}_i := \phi(a_i, b_i) := \left[\begin{smallmatrix} b_i - a_i \\ 1 \end{smallmatrix}\right].$$

- 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 $\boldsymbol{x} \mapsto 2 \cdot \mathbb{1}[\boldsymbol{w}^T \boldsymbol{x} \geq 0] - 1$,
  we have the affine decision rule $\boldsymbol{x} \mapsto 2 \cdot \mathbb{1}[\boldsymbol{w}^\mathsf{T} \boldsymbol{x} \geq -b] - 1$.

"`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 $\boldsymbol{x}$ with $\phi(\boldsymbol{x})$ 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 $\boldsymbol{x}$ with $\phi(\boldsymbol{x})$ everywhere.

Remark. Sometimes called feature engineering.
It still exists, despite promises of deep learning.

What if data are not vectors: $\boldsymbol{x} \in \mathcal{X} \neq \mathbb{R}^d$?

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 $\boldsymbol{e}_i \in \mathbb{R}^{|\mathcal{X}|}$.
   Can encode documents as normalized word counts.
2. **Word2vec:** magic embedding $\mathcal{X} \to \mathbb{R}^d$ 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).

Example feature mapping of vector data: monomial features

Suppose $\phi(x) = (1, x, x^2)$, and $\boldsymbol{w} = (a, b, c)$. Then

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

and our linear learning tools can now learn quadratic polynomials!

Example feature mapping of vector data: monomial features

Suppose $\phi(x) = (1, x, x^2)$, and $\boldsymbol{w} = (a, b, c)$. Then

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

and our linear learning tools can now learn quadratic polynomials!

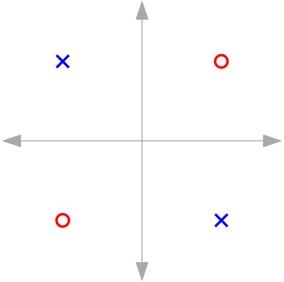
Multivariate case: define

$$\phi(\boldsymbol{x}) := (1, \underbrace{x_1, \ldots, x_d}_{\text{linear terms}}, \underbrace{x_1^2, \ldots, x_d^2}_{\text{squared terms}}, \underbrace{x_1x_2, \ldots, x_1x_d, \ldots, x_{d-1}x_d}_{\text{cross terms}}).$$

Note: hw1 has a coding problem with polynomial expansion, and the order there is slightly different.

XOR example

Consider XOR: $\boldsymbol{x} \in (\pm 1, \pm 1)$, and $y = x_1 x_2$.



Not linearly separable with provided features,
but linearly separable with quadratic features!

Overfitting

Overfitting

“`pytorch` meta-algorithm”.

$\vdots$

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**".

$\vdots$

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 $\boldsymbol{x}$, predict using $\phi(\boldsymbol{x})$, where

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

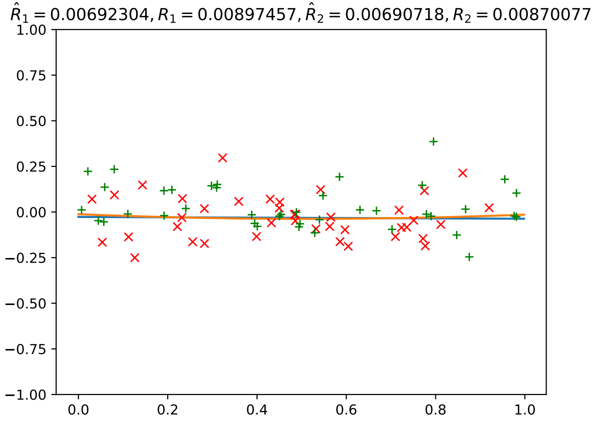
What about step 6?

Fitting polynomials to a noisy constant function

Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathsf{ols}}$.

Fitting polynomials to a noisy constant function

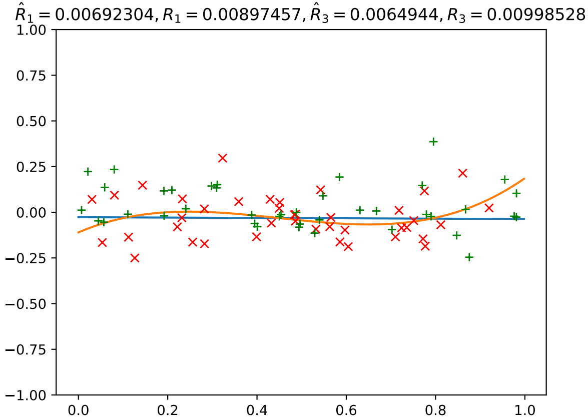
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^{\mathsf{T}}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

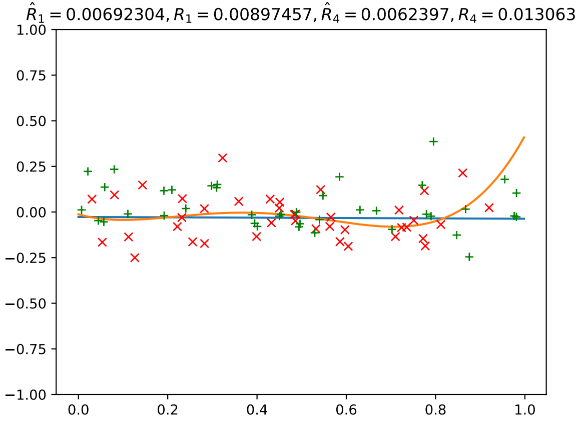
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^{\mathsf{T}}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

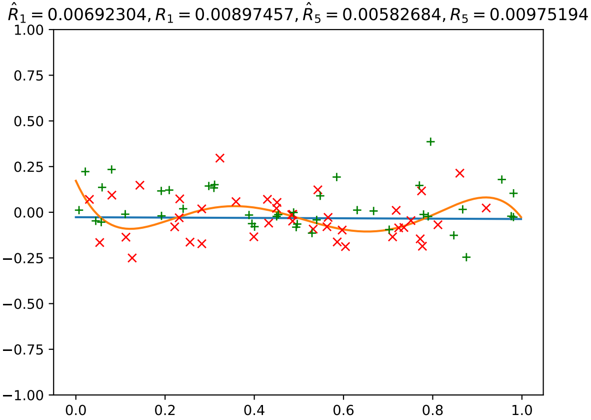
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

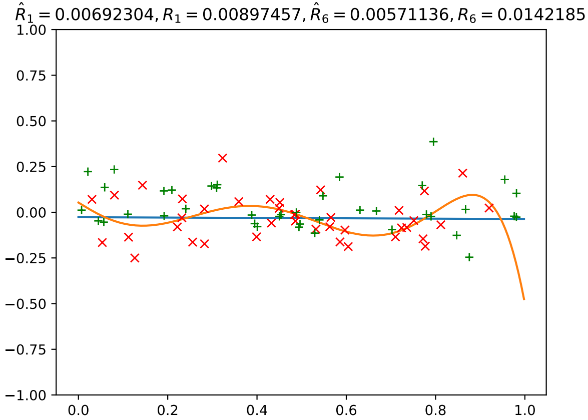
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^{\mathsf{T}}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathsf{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

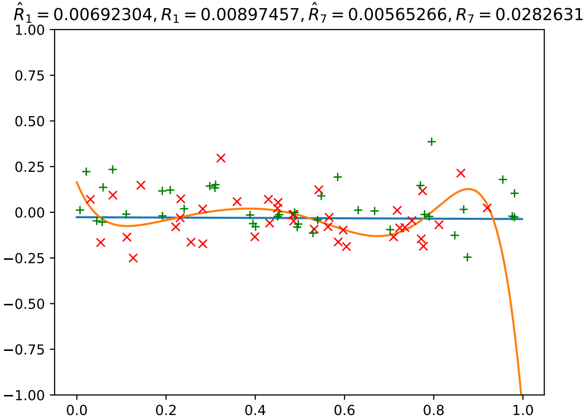
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathrm{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

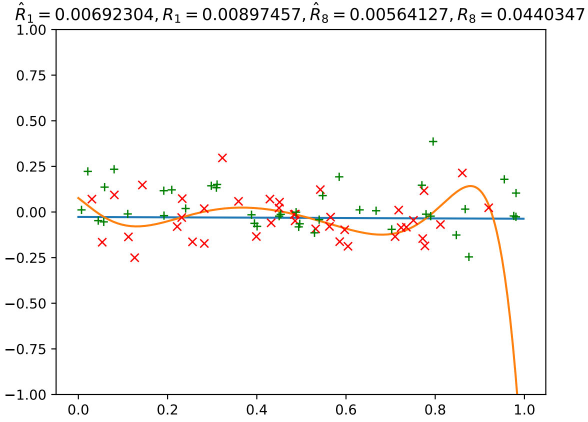
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^{\mathsf{T}}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

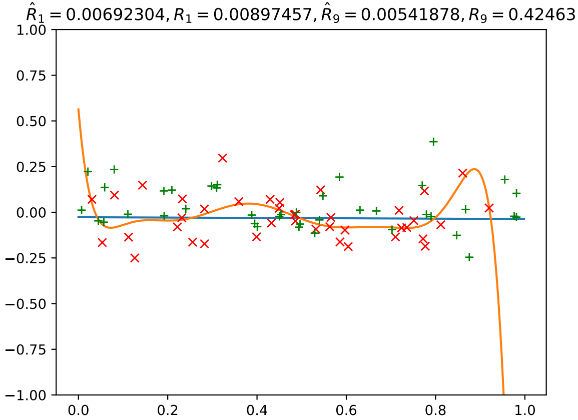
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

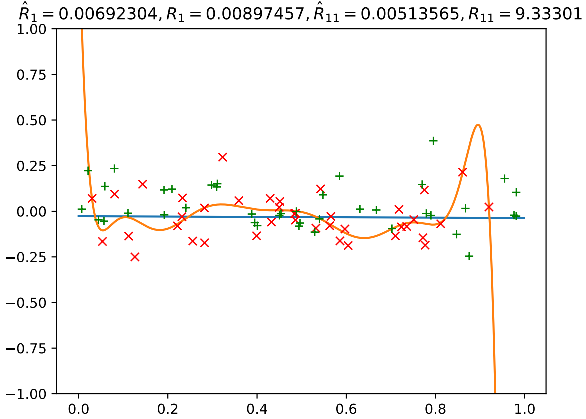
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\top(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathrm{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

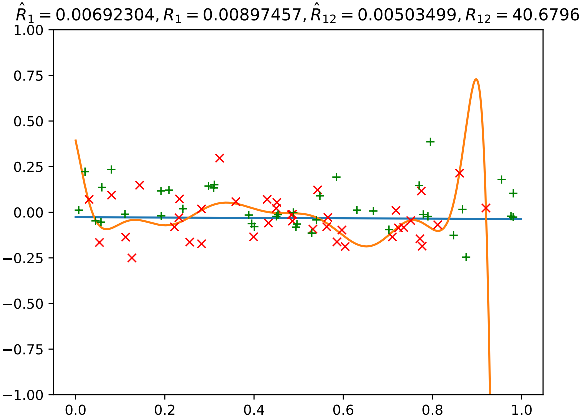
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\top(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

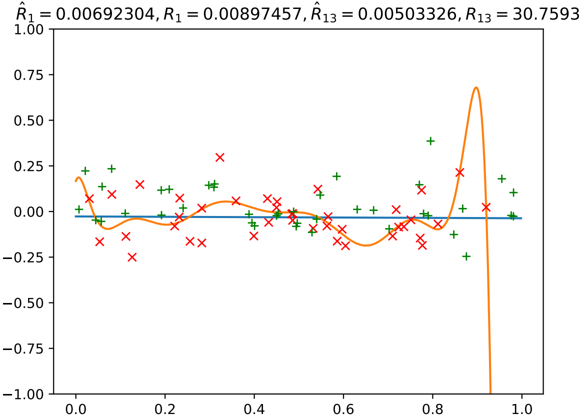
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathrm{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

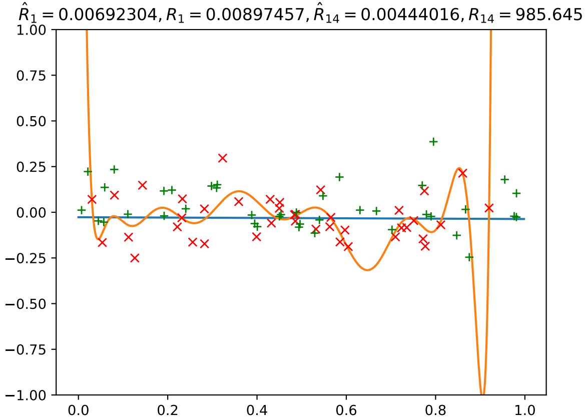
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^{\mathsf{T}}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathsf{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

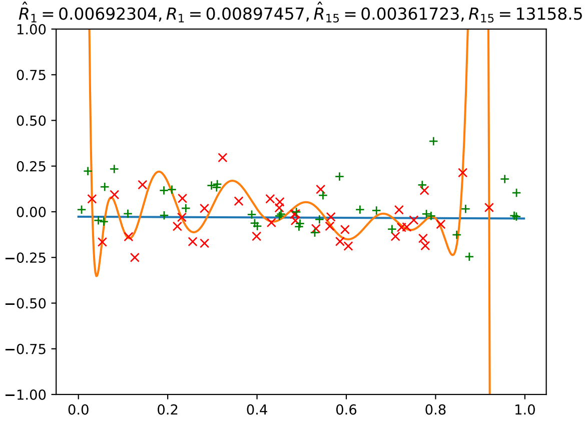
Data: (x, y) has $x \sim \text{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(training data is red, testing data is green.)

Fitting polynomials to a noisy constant function

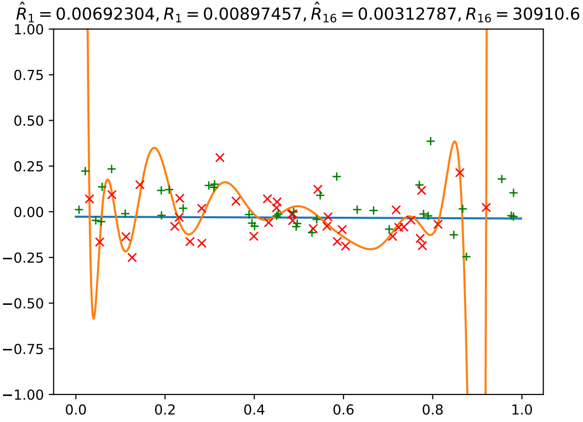
Data: (x, y) has $x \sim \mathrm{Uniform}([0,1])$ and $y \sim \mathcal{N}(0,1)$ (independent!).
Predictor: polynomials of varying degree, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\mathsf{ols}}$.



(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, $x \mapsto \boldsymbol{w}^\mathsf{T}(1, x, x^2, \ldots, x^r)$.
Method: OLS solution $\hat{\boldsymbol{w}}_{\text{ols}}$.



(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 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 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.

One approach: regularization

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

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

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

One approach: regularization

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

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

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

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

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

  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.
- λ is a hyper-parameter; needs to be found some other way.
- Sometimes this specific regularizer is called weight decay:

  $$\nabla_{\boldsymbol{w}} \left(\widehat{\mathcal{R}}(\boldsymbol{w}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2 \right) = \nabla_{\boldsymbol{w}} \widehat{\mathcal{R}}(\boldsymbol{w}) + \lambda \boldsymbol{w}.$$

Loss functions and multiclass prediction

Loss functions and multiclass prediction

So far we have seen:

$$\ell_{\mathrm{sq}}(z) = \frac{1}{2}(1-z)^2, \qquad \text{(squared loss)},$$

$$\ell_{\mathrm{logistic}}(z) = \ln(1+\exp(-z)), \qquad \text{(logistic loss)}.$$

Loss functions and multiclass prediction

So far we have seen:

$$\ell_{\mathrm{sq}}(z) = \frac{1}{2}(1 - z)^2, \qquad \text{(squared loss)},$$
$$\ell_{\mathrm{logistic}}(z) = \ln(1 + \exp(-z)), \qquad \text{(logistic loss)}.$$

Questions:

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

Loss functions and multiclass prediction

So far we have seen:

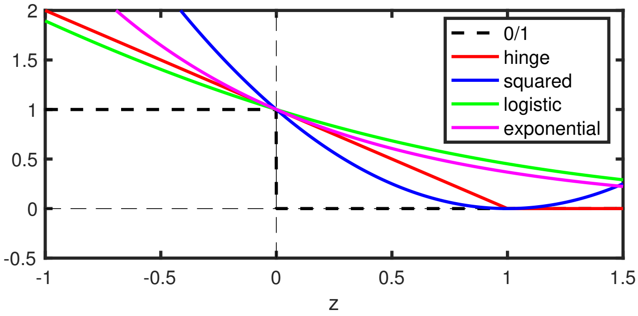
$$\ell_{\mathrm{sq}}(z) = \frac{1}{2}(1-z)^2, \qquad \text{(squared loss)},$$
$$\ell_{\mathrm{logistic}}(z) = \ln(1+\exp(-z)), \qquad \text{(logistic loss)}.$$

Questions:

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

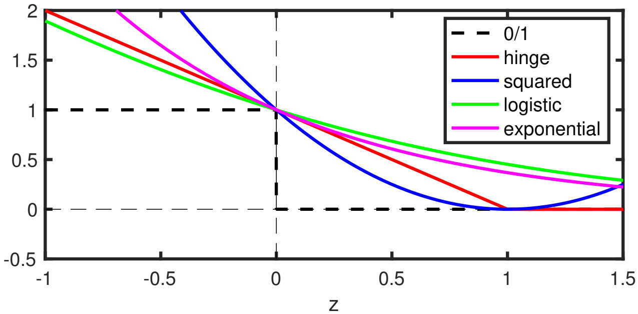
Remark: We introduced squared loss as $\ell_{\mathrm{sq}}(\hat{y}, y) = (y-\hat{y})^2/2$.
This makes sense for regression; for classification, combining arguments into a single $z := \hat{y}y$ suffices.

Standard classification losses



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	$\ell_{\exp}(z) = e^{-z}$

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	$\ell_{\exp}(z) = e^{-z}$

- Upper bound zero-one loss: $\ell(z) \to 0$ implies $\mathbb{1}[z \le 0] \to 0$. Sometimes called 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 $\boldsymbol{w}$ define a conditional model $p_{\boldsymbol{w}}(y|\boldsymbol{x})$.

Designing losses with maximum likelihood

Stronger goal than classification:
suppose our parameters $\boldsymbol{w}$ define a conditional model $p_{\boldsymbol{w}}(y|\boldsymbol{x})$.

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

$$\arg\max_{\boldsymbol{w}} \prod_{i=1}^n p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i) = \arg\min_{\boldsymbol{w}} -\ln \prod_{i=1}^n p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i) = \arg\min_{\boldsymbol{w}} \sum_{i=1}^n \ln \frac{1}{p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i)}.$$

Designing losses with maximum likelihood

Stronger goal than classification:
suppose our parameters $\boldsymbol{w}$ define a conditional model $p_{\boldsymbol{w}}(y|\boldsymbol{x})$.

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

$$\arg\max_{\boldsymbol{w}} \prod_{i=1}^{n} p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i) = \arg\min_{\boldsymbol{w}} -\ln \prod_{i=1}^{n} p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i) = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ln \frac{1}{p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i)}.$$

Looks like ERM with loss $\ln \frac{1}{p_{\boldsymbol{w}}(y|\boldsymbol{x})}$!
Note: can use these losses even if true label probabilities have a different model.

Squared loss via MLE

Define conditional model $p_{\boldsymbol{w}}$ using standard Gaussian with mean $\boldsymbol{x}^\top \boldsymbol{w}$:

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

Squared loss via MLE

Define conditional model $p_{\boldsymbol{w}}$ using standard Gaussian with mean $\boldsymbol{x}^{\mathsf{T}}\boldsymbol{w}$:

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

Then

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

and

$$\begin{aligned} \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ln \frac{1}{p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i)} &= \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ell_{\mathrm{sq}}(\boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{w}, y_i) \\ &= \arg\min_{\boldsymbol{w}} \frac{1}{n}\sum_{i=1}^{n} \ell_{\mathrm{sq}}(\boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{w}, y_i). \end{aligned}$$

Logistic loss via MLE

Define conditional model

$$p_{\boldsymbol{w}}(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{x}^\top \boldsymbol{w})}.$$

whereby

$$p_{\boldsymbol{w}}(y = -1|\boldsymbol{x}) = 1 - p_{\boldsymbol{w}}(y = 1|\boldsymbol{x}) = \frac{\exp(-\boldsymbol{x}^\top \boldsymbol{w})}{1 + \exp(-\boldsymbol{x}^\top \boldsymbol{w})} = \frac{1}{1 + \exp(\boldsymbol{x}^\top \boldsymbol{w})}.$$

Logistic loss via MLE

Define conditional model

$$p_{\boldsymbol{w}}(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{x}^\mathsf{T}\boldsymbol{w})}.$$

whereby

$$p_{\boldsymbol{w}}(y = -1|\boldsymbol{x}) = 1 - p_{\boldsymbol{w}}(y = 1|\boldsymbol{x}) = \frac{\exp(-\boldsymbol{x}^\mathsf{T}\boldsymbol{w})}{1 + \exp(-\boldsymbol{x}^\mathsf{T}\boldsymbol{w})} = \frac{1}{1 + \exp(\boldsymbol{x}^\mathsf{T}\boldsymbol{w})}.$$

Then

$$\begin{aligned}
\ln \frac{1}{p_{\boldsymbol{w}}(y|\boldsymbol{x})} &= \ln \frac{1}{p_{\boldsymbol{w}}(y = 1|\boldsymbol{x})^{(1+y)/2}(1 - p_{\boldsymbol{w}}(y = 1|\boldsymbol{x}))^{(1-y)/2}} \\
&= \frac{1+y}{2} \ln(1 + \exp(-\boldsymbol{x}^\mathsf{T}\boldsymbol{w})) + \frac{(1-y)}{2} \ln(1 + \exp(\boldsymbol{x}^\mathsf{T}\boldsymbol{w})) \\
&= \ln(1 + \exp(-y\boldsymbol{x}^\mathsf{T}\boldsymbol{w})),
\end{aligned}$$

and

$$\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ln \frac{1}{p_{\boldsymbol{w}}(y_i|\boldsymbol{x}_i)} = \arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{logistic}}(y_i \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}).$$

Multiclass classification

Setup: given $\boldsymbol{x}$ predict $y \in \{1, \dots k\}$.

Two approaches:

- One-vs-all (a reduction approach): train k binary classifiers, each with $\tilde{y}_i := 2 \cdot \mathbb{1}[y_i = j] - 1$. Predict by choosing the largest output.
- Directly train a model $p(y = j | \boldsymbol{x})$ with maximum likelihood.

Cross-entropy loss (multi-class logistic loss) via MLE

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

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

Cross-entropy loss (multi-class logistic loss) via MLE

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

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

$$
\begin{aligned}
\ln \frac{1}{p_f(\widehat{y} = y|\boldsymbol{x})} &= \ln \frac{\sum_{j=1}^k \exp(f(\boldsymbol{x})_j)}{\exp(f(\boldsymbol{x}))_y} \\
&= -f(\boldsymbol{x})_y + \ln \sum_{j=1}^k \exp\left(f(\boldsymbol{x})_j\right) \\
&=: \ell_{\mathsf{ce}}(f(\boldsymbol{x}), y).
\end{aligned}
$$

Cross-entropy loss (multi-class logistic loss) via MLE

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

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

$$\begin{aligned}
\ln \frac{1}{p_f(\widehat{y} = y|\boldsymbol{x})} &= \ln \frac{\sum_{j=1}^k \exp(f(\boldsymbol{x})_j)}{\exp(f(\boldsymbol{x}))_y} \\
&= -f(\boldsymbol{x})_y + \ln \sum_{j=1}^k \exp\left(f(\boldsymbol{x})_j\right) \\
&=: \ell_{\mathsf{ce}}(f(\boldsymbol{x}), y).
\end{aligned}$$

Notes.
In `pytorch`, this is `torch.nn.CrossEntropyLoss()(f(x), y)`.
Loss is minimized when $p_f(\widehat{y} = y|\boldsymbol{x}) \approx 1$.
Name comes from the cross-entropy expression $\sum_{j=1}^k (\boldsymbol{e}_y)_j \ln \frac{1}{p_f(\hat{y}=j|\boldsymbol{x})}$.

Summary for today

- Features.
- Overfitting.
- Loss construction and multiclass output.

(Appendix.)

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:

$$\boldsymbol{Y} := \begin{bmatrix} \leftarrow & \boldsymbol{y}_1^\mathsf{T} & \rightarrow \\ & \vdots & \\ \leftarrow & \boldsymbol{y}_n^\mathsf{T} & \rightarrow \end{bmatrix}.$$

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

$$\operatorname*{arg\,min}_{\boldsymbol{W} \in \mathbb{R}^{d \times k}} \|\boldsymbol{X}\boldsymbol{W} - \boldsymbol{Y}\|_\mathsf{F}^2 .$$

Multivariate prediction: regression case

Suppose we have k labels for each example:

$$\boldsymbol{Y} := \begin{bmatrix} \leftarrow & \boldsymbol{y}_1^\top & \rightarrow \\ & \vdots & \\ \leftarrow & \boldsymbol{y}_n^\top & \rightarrow \end{bmatrix}.$$

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

$$\operatorname*{arg\,min}_{\boldsymbol{W} \in \mathbb{R}^{d\times k}} \|\boldsymbol{X}\boldsymbol{W} - \boldsymbol{Y}\|_{\mathsf{F}}^2 .$$

This is the same as k regular linear regressions: given $\boldsymbol{W}$,

$$\|\boldsymbol{X}\boldsymbol{W} - \boldsymbol{Y}\|_{\mathsf{F}}^2 = \sum_{j=1}^{n} \|\boldsymbol{X}\boldsymbol{W}_{:j} - \boldsymbol{Y}_{:j}\|^2 = \sum_{j=1}^{n}\sum_{i=1}^{n} \left(\boldsymbol{X}_{i:}^\top \boldsymbol{W}_{:j} - y_{ij}\right)^2 .$$

We have reduced multivariate regression to univariate regression:

$$\min_{\boldsymbol{W} \in \mathbb{R}^{d\times k}} \|\boldsymbol{X}\boldsymbol{W} - \boldsymbol{Y}\|_{\mathsf{F}}^2 = \sum_{j=1}^{k} \min_{\boldsymbol{w} \in \mathbb{R}^d} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{Y}_{:j}\|^2 .$$

Remark on modeling assumptions **(incomplete slide!)**

1. It is **not true** that logistic loss and squared loss only work well when the preceding likelihood models are correct.
2. There is a complicated way to argue that they can still behave optimally even when their model is wrong, but here I'll give a simpler perspective.
3. We can derive the notion of sample mean and sample covariance by solving for mean and variance in a maximum likelihood computation for multivariate Gaussian parameters over some point set. That is to say, the sample mean and covariance will recover the Gaussian parameters when the data is generated by a Gaussian. However, when the data is **not** generated by a Gaussian, while we are not "recovering parameters of an underlying model", we are still obtaining useful statistics which describe the data well (and, e.g., if we were to get all moments, we would in a sense completely characterize the data.

Supplemental reading

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