Natural Language Processing
Text classification

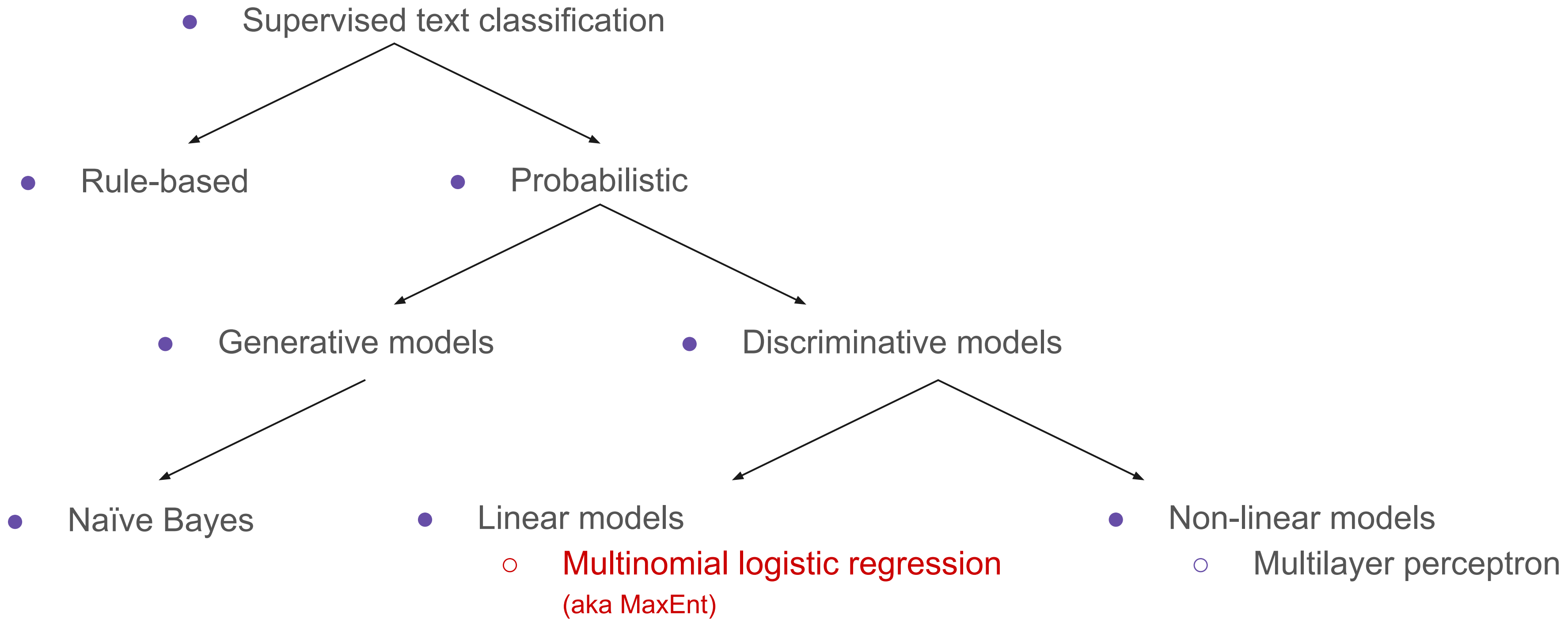
Sofia Serrano
sofias6@cs.washington.edu

Credit to Yulia Tsvetkov and Noah Smith for slides

Announcements

- We’ll be holding extra office hours next week
    - Exact times will be posted on the course website and as an announcement on Ed by the end of the weekend

Logistic regression

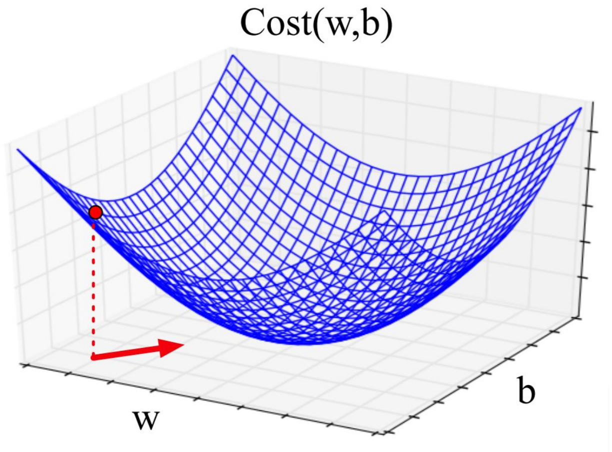


Last time we established...

- The structure of a (binary) logistic regression model
  - weights w corresponding to features
  - a sigmoid function applied as the last layer in order to form probabilities
- How to apply an existing (binary) logistic regression model

And we *started* to talk about

- How to learn w (and b)
  - Gradient descent
    - Note: NOT solely for logistic regression



What’s left from logistic regression?

- The **loss function** that we use in conjunction with gradient descent
- Transitioning from binary logistic regression to **multinomial** logistic regression

And some additional loose ends that are nevertheless important in practice:

- Gradient descent → Stochastic Gradient Descent
- Preventing overfitting
  - Regularization
- Hyperparameters

Cross-entropy loss

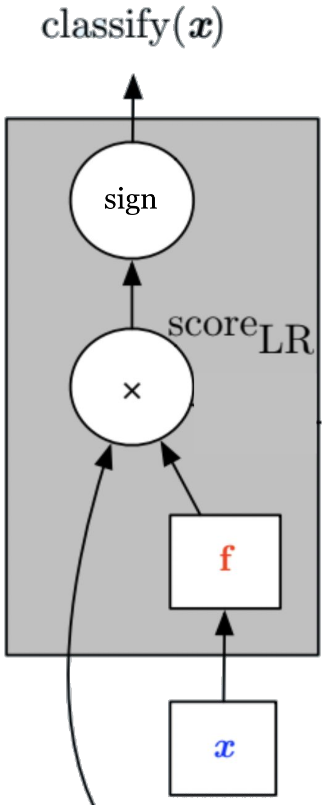
Intuition of negative log likelihood loss = cross-entropy loss

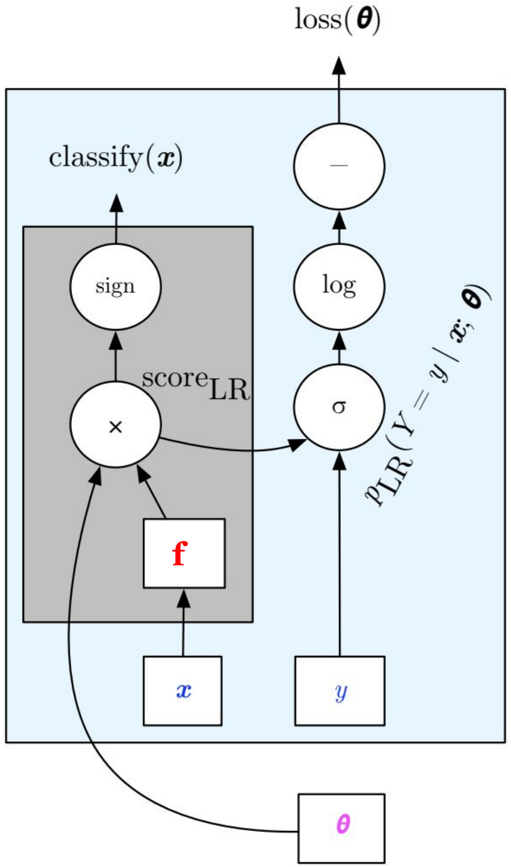
A case of conditional maximum likelihood estimation

We choose the parameters w,b that maximize

- the log probability
- of the true y labels in the training data
- given the observations x

From classification to a loss function





loss($\boldsymbol{\theta}$)
classify($\boldsymbol{x}$)
−
sign
log
score$_{\mathrm{LR}}$
×
σ
$p_{\mathrm{LR}}(Y = y \mid \boldsymbol{x}; \boldsymbol{\theta})$
f
$\boldsymbol{x}$
y
$\boldsymbol{\theta}$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Since there are only 2 discrete outcomes (0 or 1) we can express the probability $p(y|x)$ from our classifier (the thing we want to maximize) as

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Since there are only 2 discrete outcomes (0 or 1) we can express the probability $p(y|x)$ from our classifier (the thing we want to maximize) as

$$p(y|x) = \hat{y}^{y}\,(1-\hat{y})^{1-y}$$

Noting:

if $y=1$, this simplifies to $\hat{y}$

if $y=0$, this simplifies to $1 - \hat{y}$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label p(y|x)

Maximize: $p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$

Now take the log of both sides (mathematically handy)

Maximize:
$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^y (1-\hat{y})^{1-y}\right] \\ &= y \log \hat{y} + (1-y) \log(1-\hat{y}) \end{aligned}$$

Whatever values maximize log p(y|x) will also maximize p(y|x)

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Maximize: $$\begin{aligned} \log p(y|x) &= \log\left[\hat{y}^{y}(1-\hat{y})^{1-y}\right] \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y}) \end{aligned}$$

Now flip sign to turn this into a loss: something to minimize

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label p(y|x)

Maximize:
$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^{y}\,(1-\hat{y})^{1-y}\right]\\ &= y\log\hat{y}+(1-y)\log(1-\hat{y})\end{aligned}$$

Now flip sign to turn this into a loss: something to minimize

$$L_{\mathrm{CE}}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y}+(1-y)\log(1-\hat{y})]$$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label p(y|x)

Maximize:
$$\begin{aligned}\log p(y|x) &= \log \left[\hat{y}^{y}(1-\hat{y})^{1-y}\right] \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y})\end{aligned}$$

Now flip sign to turn this into a **cross-entropy loss**: something to minimize

Minimize:
$$L_{\mathrm{CE}}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

Or, plug in definition of $\hat{y} = \sigma(w{\cdot}x+b)$

$$\boxed{L_{\mathrm{CE}}(\hat{y},y) = -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))]}$$

Zooming out for a sec...

Remember: the loss on the last slide is for a single instance of training data!

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))]$$

In practice, the function we *actually* want to minimize is an averaged version of that loss over *all* our training examples:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

Let's see if this works for our sentiment example

We want loss to be:

- smaller if the model estimate $\hat{y}$ is close to correct
- bigger if model is confused

Let's first suppose the true label of this is $y=1$ (positive)

It's hokey . There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Let's see if this works for our sentiment example

True value is y=1 (positive). How well is our model doing?

$$\begin{aligned} p(+|x) = P(Y=1|x) &= \sigma(w \cdot x + b) \\ &= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\ &= \sigma(.833) \\ &= 0.70 \end{aligned}$$

Pretty well! What's the loss?

$$\begin{aligned} L_{CE}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log \sigma(\mathbf{w} \cdot \mathbf{x} + b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

Let's see if this works for our sentiment example

Suppose the true value instead was y=0 (negative).

$$\begin{aligned} p(-|x) = P(Y=0|x) &= 1-\sigma(w\cdot x+b) \\ &= 0.30 \end{aligned}$$

What's the loss?

$$\begin{aligned} L_{\text{CE}}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -[\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

Let's see if this works for our sentiment example

The loss when the model was right (if true y=1)

$$\begin{aligned} L_{\mathrm{CE}}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -[\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

The loss when the model was wrong (if true y=0)

$$\begin{aligned} L_{\mathrm{CE}}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -[\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

Sure enough, loss was bigger when model was wrong!

The gradient

Remember that we need to take the gradient of that loss.

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix}$$

Turns out that there's a nice closed expression for that gradient!

What are these partial derivatives for logistic regression?

The loss function:

$$L_{\mathrm{CE}}(\hat{y},y) = -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))]$$

The elegant derivative of this function (see Section 5.10 for the derivation)

$$\begin{aligned}\frac{\partial L_{\mathrm{CE}}(\hat{y},y)}{\partial w_j} &= [\sigma(w\cdot x+b)-y]x_j \\ &= (\hat{y}-y)\mathbf{x}_j\end{aligned}$$

Multinomial logistic regression

Multinomial Logistic Regression

Often we need more than 2 classes

- Positive/negative/neutral
- Parts of speech (noun, verb, adjective, adverb, preposition, etc.)
- Classify emergency SMSs into different actionable classes

If >2 classes we use **multinomial logistic regression**

Changes we'll need to make

- We need more weights per feature
- … and something other than the sigmoid function
- … which results in a slightly different loss function.

Why might we need more weights per feature?

Imagine we're doing topic classification:

Does this document talk about… river ecosystems? finance? or electricity?

What weights might we want to give to the following features?

Bank	River ecosystems: Med +	Finance: Very +	Electricity: Low +
Ground	River ecosystems: Low +	Finance: Very –	Electricity: Very +
Current	River ecosystems: Very +	Finance: Low +	Electricity: Very +

Why don't we just copy each feature for each class?

w for our classifier will now contain a separate weight w_i for each of the following:

- (bank, river ecosystems)
- (bank, finance)
- (bank, electricity)
- (ground, river ecosystems)
- (ground, finance)
- (ground, electricity)
- (current, river ecosystems)
- (current, finance)
- (current, electricity)

… but in order to take full advantage of these newly expanded features, we'll also have to replace the sigmoid.

Why are you making us drop the sigmoid, Sofia?

$$\begin{aligned} P(y=1) &= \sigma(w \cdot x + b) \\ &= \frac{1}{1+\exp(-(w \cdot x + b))} \end{aligned}$$

We need more than a single output probability that can only move either up or down…

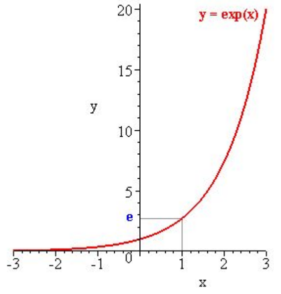
… but we would like to keep the output in the form of probabilities.

Softmax: a generalization of sigmoid

- For a vector z of dimensionality k, the softmax is:

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \le i \le k$$



Example:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Softmax properties

- Takes a vector $z = [z_1, z_2, ..., z_k]$ of k arbitrary values
- Outputs a probability distribution
- each value in the range $[0,1]$
- all the values summing to 1

We’ll see it again (a *lot*) when we get into neural networks later.

“You just ruined our loss function, Sofia.”

No I didn’t!! I’ve just… improved it >:)

This is what we had as our loss for binary logistic regression:

$$L_{\mathrm{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))]$$

Which, if we imagine our model as guessing about the true “label vector” $\mathbf{y} = [\ ? \quad ?\ ]$, is equivalent to

$$-[y \log \hat{p}(\mathbf{y}_1 = 1|\mathbf{x}) + (1-y) \log \hat{p}(\mathbf{y}_0 = 1|\mathbf{x})]$$

$$-\log \hat{p}(\mathbf{y}_{\text{whichever class is right}} = 1|\mathbf{x})$$

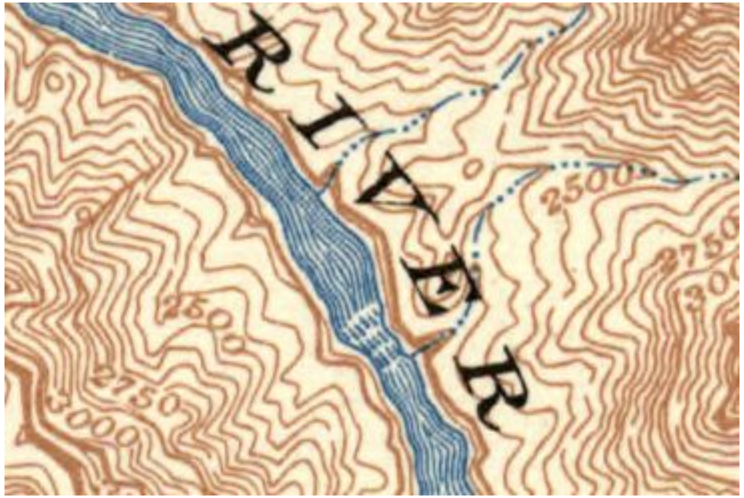
$$-\log \hat{p}(\mathbf{y}_c = 1|\mathbf{x}) \quad \text{(where } c \text{ is the correct class)}$$

New loss $\longrightarrow$

$$-\log \frac{\exp(\mathbf{w_c} \cdot \mathbf{x} + b_c)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)} \quad (c \text{ is the correct class})$$

… and some loose ends

Gradient Descent → Stochastic Gradient Descent



Key difference from our motivating scenario on Wednesday: in practice, calculating the exact gradient is really time-consuming.

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

So… we estimate the gradient using samples of data.

Mini-batch training

Stochastic gradient descent calculates gradients based on subsets of random examples from the training data at a time.

If you do this with only one instance at a time, that can result in choppy movements.

So it’s very common to compute gradient over “mini-batches” of training instances (not just single instances).

```
**function** STOCHASTIC GRADIENT DESCENT($L()$, $f()$, $x$, $y$) **returns** $\theta$
     # where: L is the loss function
     #        f is a function parameterized by $\theta$
     #        x is the set of training inputs $x^{(1)}, x^{(2)}, ..., x^{(m)}$
     #        y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, ..., y^{(m)}$

$\theta \leftarrow 0$
**repeat** til done     (or subset of training tuples that you've partitioned the training data into)
   For each $\boxed{\text{training tuple } (x^{(i)}, y^{(i)})}$ (in random order)
      1. Optional (for reporting):             # How are we doing on this tuple?
         Compute $\hat{y}^{(i)} = f(x^{(i)};\theta)$     # What is our estimated output $\hat{y}$?
         Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$    # How far off is $\hat{y}^{(i)}$) from the true output $y^{(i)}$?
      2. $g \leftarrow \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$   # How should we move $\theta$ to maximize loss?
      3. $\theta \leftarrow \theta - \eta\, g$             # Go the other way instead
return $\theta$
```

Overfitting

A model that perfectly matches the training data has a problem.

It will also **overfit** to the data, modeling noise

- A random word that perfectly predicts y (it happens to only occur in one class) will get a very high weight.
- Failing to generalize to a test set without this word.

A good model should be able to **generalize**

Regularization

A solution for overfitting

Add a **regularization** term R(θ) to the loss function (for now written as maximizing logprob rather than minimizing loss)

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta)$$

Idea: choose an R(θ) that penalizes large weights

- fitting the data well with lots of big weights not as good as fitting the data a little less well, with small weights

L2 regularization (ridge regression)

The sum of the squares of the weights

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2$$

L2 regularized objective function:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2$$

L1 regularization (aka “lasso regression”)

The sum of the (absolute value of the) weights

$$R(\theta) \;=\; ||\theta||_1 = \sum_{i=1}^{n} |\theta_i|$$

L1 regularized objective function:

$$\hat{\theta} \;=\; \underset{\theta}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j|$$

L1 regularization prefers sparse solutions. Why?

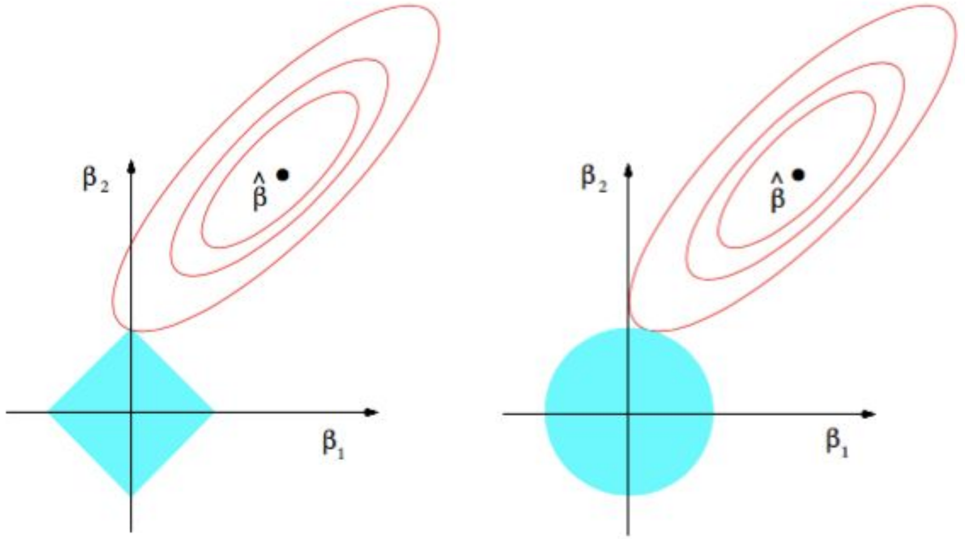


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

From *Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman (Fig. 3.11)

Hyperparameters

Hyperparameters:

- Briefly, a special kind of parameter for an ML model
- Instead of being learned by algorithm from supervision (like regular parameters), they are chosen by algorithm designer.

The coefficient multiplied by a regularization term is an example of a **hyperparameter**.

The learning rate η is another hyperparameter.

- too high: the learner will take big steps and overshoot
- too low: the learner will take too long

Components of a probabilistic machine learning classifier

Given m input/output pairs $(x^{(i)}, y^{(i)})$:

1. A **feature representation** for the input. For each input observation $x^{(i)}$, a vector of features $[x_1, x_2, \ldots, x_n]$. Feature j for input $x^{(i)}$ is x_j, more completely $x_1^{(i)}$, or sometimes $f_j(x)$.

2. A **classification function** that computes $\hat{y}$ the estimated class, via $p(y|x)$, like the **sigmoid** or **softmax** functions

3. An **objective function** for learning, like **cross-entropy loss**

4. An algorithm for **optimizing** the objective function: **stochastic gradient descent**

Next class:

- Wrapping up text classification