Linear classification

Linear classification

Recall the binary classification setup:

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times D} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \{0, 1\}^N$$

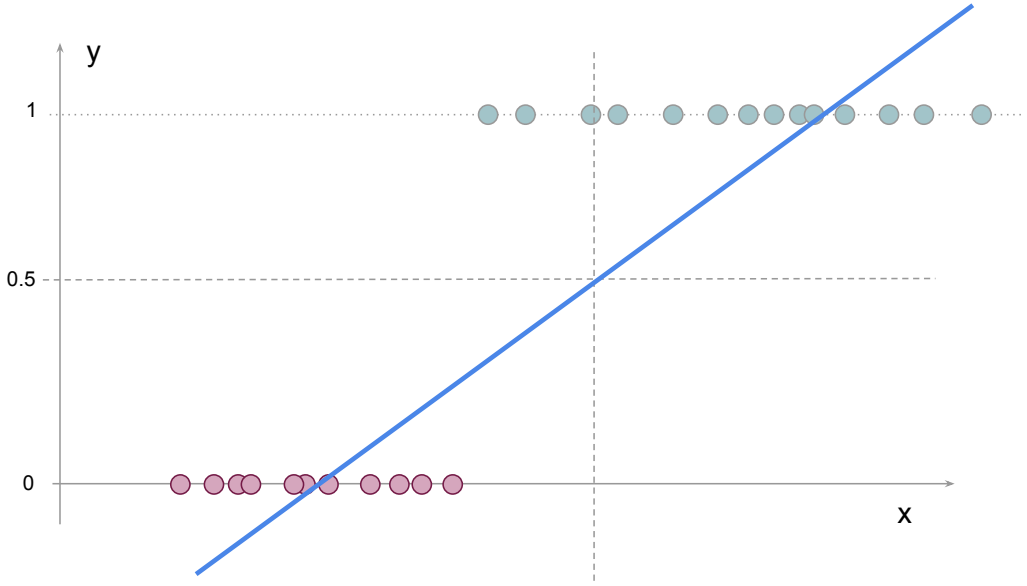
Q: Is it possible to use our linear regression model to do classification?

Linear classification

A: Sure, but it probably won't work very well.

1. The relationship between the variates and covariates is not directly linear
2. We would like our outputs to be binary variables in $\{0, 1\}$ (or probabilities in $[0 \ldots 1]$), but linear regression produces arbitrary real numbers. We would need to do some post processing to map the values to the desired range.
3. Assumption of Gaussian noise is not true for binary outputs. So square error is inappropriate.

Linear classification with least squares



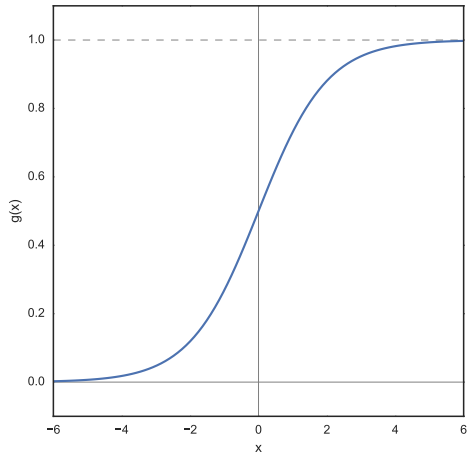
Logistic regression

Let's assume that the relationship between the input and the outputs is:

$$f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b)$$

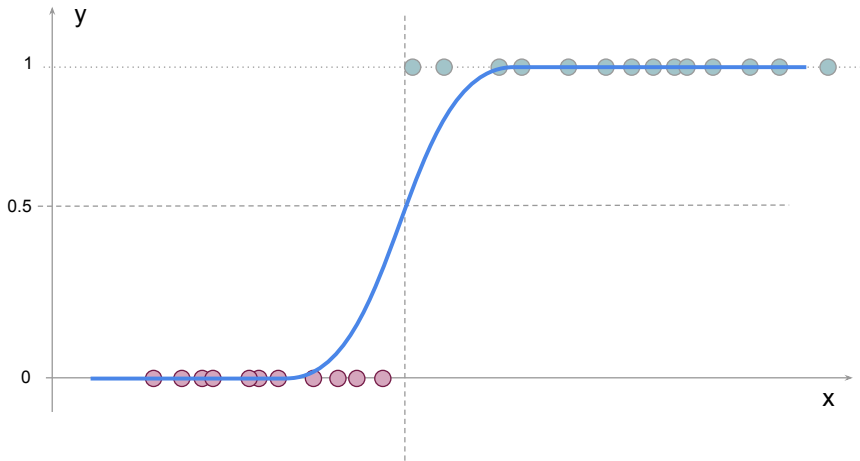
with $g(.)$ being the sigmoid (logistic) function, which "squashes" its input to the range $(0, 1)$:

$$g(x) = \frac{1}{1 + e^{-x}}$$



Logistic regression

$$f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b)$$



Logistic regression

But why the sigmoid?

Assume that the **log-odds** are linear:

$$\log\left(\frac{y}{1-y}\right) = \mathbf{w}^T\mathbf{x} + b$$

Now solve for y!

Logistic regression

To fit the parameters we would also like to use a **more appropriate loss function than square loss** (which is only appropriate for normally distributed error).

Note $f(\mathbf{x})$ can now be interpreted as a probability distribution $P(\mathbf{y} \mid X, \theta)$. Following the maximum likelihood approach, we want to find:

$$\begin{aligned}\hat{\theta}_{\text{ML}} &= \arg\max_{\theta} P(X, \mathbf{y} \mid \theta) \\ &= \arg\max_{\theta} P(\mathbf{y} \mid X, \theta) P(X) \\ &= \arg\max_{\theta} P(\mathbf{y} \mid X, \theta)\end{aligned}$$

Logistic regression

Since y is a binary variable, it is appropriate to consider the distribution of y as a **Bernoulli random variable**:

$$\mathrm{Bern}(y \mid \lambda) = \lambda^{y}(1-\lambda)^{1-y}$$

Assuming i.i.d samples we have:

$$\begin{aligned} P(\mathbf{y} \mid X, \theta) &= \prod_{i=1}^{N} \mathrm{Bern}(y_i \mid f(\mathbf{x}_i)) \\ &= \prod_{i=1}^{N} f(\mathbf{x}_i)^{y_i}(1-f(\mathbf{x}_i))^{1-y_i} \end{aligned}$$

Aside: Bernoulli random variables

Fair coin

- P(heads) = P(tails) = 0.5

Biased coin

- P(heads) = 0.75
- P(tails) = 1 - P(heads) = 0.25

In general, we say:

$$y \sim \text{Bern}(y \mid \lambda)$$

if

$$P(y) = \begin{cases} \lambda & y = 1 \\ 1 - \lambda & y = 0 \end{cases}$$

which can be written as:

$$P(y) = \lambda^y (1 - \lambda)^{1-y}$$

Logistic regression

Again, we minimize the negative log probability instead of maximizing the probability, as this is easier and gives the same estimate

$$
\begin{aligned}
\theta_{\mathrm{ML}} &= \arg\max_{\theta} \prod_{i=1}^{N} f(\mathbf{x}_i)^{y_i}(1 - f(\mathbf{x}_i))^{1-y_i} \\
&= \arg\min_{\theta} \; -\log \prod_{i=1}^{N} f(\mathbf{x}_i)^{y_i}(1 - f(\mathbf{x}_i))^{1-y_i} \\
&= \arg\min_{\theta} \; -\sum_{i=1}^{N} \log \left(f(\mathbf{x}_i)^{y_i}(1 - f(\mathbf{x}_i))^{1-y_i} \right) \\
&= \arg\min_{\theta} \; -\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))
\end{aligned}
$$

Which gives us a reasonable loss function to minimize known as the **binary cross-entropy loss**.

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))$$

We now have all the components needed to specify the logistic regression algorithm:

1. An activation function (transfer function): the logistic sigmoid
2. An appropriate loss function (cross-entropy) derived using maximum likelihood

Logistic regression

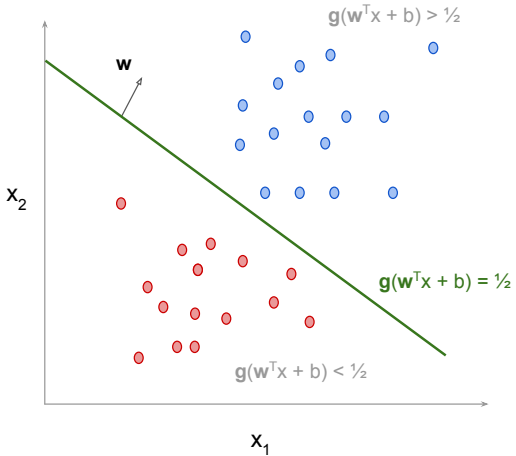
$$f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b)$$

Activation function: sigmoid

$$g(x) = \frac{1}{1 + e^{-x}}$$

Loss function: cross entropy

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))$$



Gradient descent for logistic regression

If we try to find the gradient of the loss with respect to the parameters and set this to zero and try find a closed form solution like we did with linear regression, we will fail.

Instead we turn again to an iterative solution: **gradient descent**.

We need to first find the gradient (vector) of the loss function wrt. the parameters

$$\nabla_\theta \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial \theta_i}\right]$$

Once we have this, we can perform gradient descent (or SGD) using the usual update rule to find good parameters $\hat{\theta} = \{\mathbf{w}, b\}$:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t)$$

Gradient descent for logistic regression

First note that the derivative of the transfer function (sigmoid) can be written as

$$g'(x) = g(x)(1 - g(x))$$

Loss to minimize:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))$$

with

$$f(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x})$$

where again we've prepended a 1 to $\mathbf{x}$ to eliminate the explicit bias.

Gradient descent for logistic regression

$$\nabla_{\mathbf{w}} \left[y_i \log g(\mathbf{w}^T \mathbf{x}_i) \right]$$

$$= y_i \frac{g'}{g} \mathbf{x}_i$$
$$= y_i \frac{g(1-g)}{g} \mathbf{x}_i$$
$$= (y_i - y_i g) \mathbf{x}_i$$

$$\nabla_{\mathbf{w}} \left[(1-y_i) \log(1 - g(\mathbf{w}^T \mathbf{x}_i)) \right]$$

$$= (1-y_i) \frac{-g'}{1-g} \mathbf{x}_i$$
$$= (1-y_i) \frac{-g(1-g)}{1-g} \mathbf{x}_i$$
$$= (y_i g - g) \mathbf{x}_i$$

Gradient descent for logistic regression

Putting them together

$$\begin{aligned}\nabla_{\mathbf{w}}\mathcal{L} &= -\sum_{i=1}^{N}(y_i - y_i g)\mathbf{x}_i + (y_i g - g)\mathbf{x}_i \\ &= -\sum_{i=1}^{N}(y_i - g)\mathbf{x}_i \\ &= \sum_{i=1}^{N}(f(\mathbf{x}_i) - y_i)\mathbf{x}_i\end{aligned}$$

Gradient descent for logistic regression

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{i=1}^{N} (f(\mathbf{x}_i) - y_i)\mathbf{x}_i$$

Intuitive explanation: contribution of point $\mathbf{x}_i$ to gradient depends upon the difference between the actual value and the prediction.

Comment: Note how similar this is to the gradient we get for linear regression. It's the same, except now f is a nonlinear function of $\mathbf{x}$.

So we can't set to zero and solve. But we can do gradient descent fine.

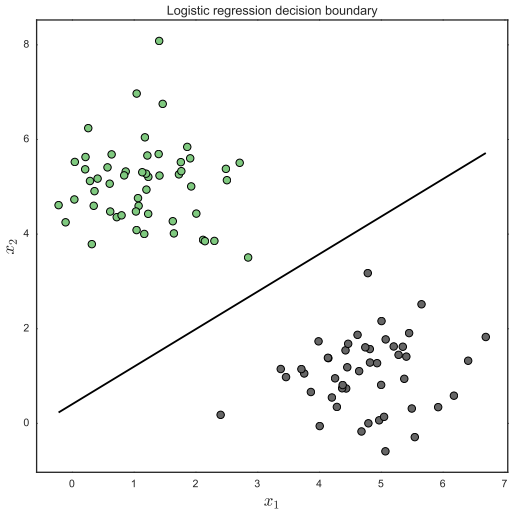
$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t)$$

Example

Training data: 100 points sampled from two $2D$ Gaussians with centers at $(1, 5)$ and $(5, 1)$.

Gradient descent:

- Initial $\mathbf{w}_0 = (0, 0)$ and $b_0 = 1$.
- Learning rate $\alpha = 0.001$.
- Gradient descent for 1000 iterations.
- Initial loss: 0.84. Final loss: 0.00489.
- Fit parameters $\hat{\mathbf{w}} = (1.75, -2.2)$ and $\hat{b} = 0.926$



Linear and logistic regression in scikit-learn

1. `sklearn.linear_model.LogisticRegression`
2. `sklearn.linear_model.LinearRegression`

Usual methods: `fit(X, y)`, `predict(X)`, and `score(X, y)`.

Fit parameters $\hat{\mathbf{w}}, \hat{b}$ accessible via `.coef_` and `.intercept_` attributes.

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X, y)
y_hat = clf.predict(X)
```

Summary

- Logistic regression is a binary **classification** algorithm.
- Decision function:

$$f(\mathbf{x}) = g(\mathbf{w}^T x + b)$$

with

$$g(z) = \frac{1}{1 + \exp(-z)}$$

- Loss function is **binary cross entropy**:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))$$

- Optimize using gradient descent (or SGD)

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{i=1}^{N} (f(\mathbf{x}_i) - y_i)\mathbf{x}_i$$

Features

More on linear regression: fitting polynomials

Possible to use linear regression for more than just fitting lines!

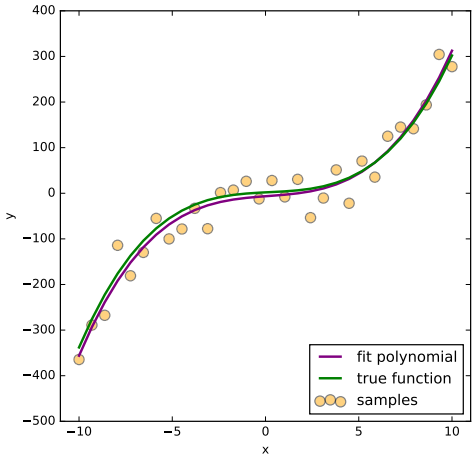
Only needs for function to be linear in the parameters

E.g. can fit polynomials:

$$f(x) = w_1 x + w_2 x^2 + w_3 x^3 + b$$

Figure on right:

$f(x) = 2x + 0.2x^2 + 0.3x + 2 \qquad y = f(x) + \epsilon$



Fitting polynomials

To fit a polynomial with degree D we just need to construct the X matrix so that contains the relevant powers of x: $x, x^2, \ldots, x^D$. E.g. to fit a degree 3 polynomial, construct X as follows:

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 \end{bmatrix}$$

Can then just use linear regression as usual.

To fit polynomials in higher dimensions, construct X to contains powers of all input variables and cross terms (if necessary).

Nonlinear mappings

More generally, we can use any (nonlinear) function to map $\mathbf{x}$ to a different space of features. The resulting function is still linear in $\mathbf{w}$.

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b$$

E.g.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \phi(\mathbf{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

Nonlinear mappings

Note: we are still fitting a hyperplane in the new space defined by the projection function. This is turns out to be nonlinear when back projected to the original space of $\mathbf{x}$.

Classification: we can do the same for logistic regression. In this case, the decision boundary is nonlinear when back projected into the original space.

Of course, we must somehow design the function $\phi(\mathbf{x})$.

Feature engineering

Designing a good feature mapping function $\phi(\mathbf{x})$ can be difficult!

- **Low dimensional**: less likely for data to be linearly separable
- **High dimensional**: possibly more prone to overfitting

Process of designing such functions by hand is called **feature engineering**.

Process of choosing features from a candidate set is called **feature selection**.

Good features are often the key to good generalization performance.

Feature engineering

$\phi(\mathbf{x})$ can be (and often must be) a very complicated function of the data.

Examples

- Computer vision: $\phi(\mathbf{x})$ could be the scale invariant feature transform (SIFT), followed by a bag-of-words encoding
- Speech recognition: $\phi(\mathbf{x})$ could be mel frequency cepstral coefficients (MFCC)
- Text classification: $\phi(\mathbf{x})$ could be a TF-IDF representation of the text
- $\phi(\mathbf{x}) =$ PCA transform
- $\phi(\mathbf{x}) =$ histogram
- $\phi(\mathbf{x}) =$ random projections

Feature learning

Feature engineering is notoriously difficult!

Alternative: learn features from the data directly.

Known as **representation learning**: $\phi(\mathbf{x})$ produces a representation of the data in which it is easier to solve the relevant problem (classification, regression, etc.)

Can be done either:

- Unsupervised: from X alone with no labels, e.g. PCA, clustering.
- Supervised: using both X and $\mathbf{y}$.
- Semi-supervised: some labeled data, some unlabeled.

Deep learning is a very successful method for representation learning. Later...

Overfitting

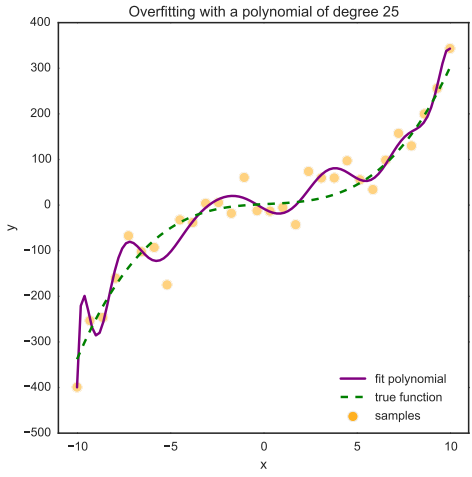
If the model has too many degrees of freedom, you can end up fitting not only the patterns of interest, but also the **noise**.

$$\mathbf{E}[(y - \hat{f})^2] = \sigma^2 + \mathbf{var}[\hat{f}] + \mathbf{bias}[\hat{f}]$$

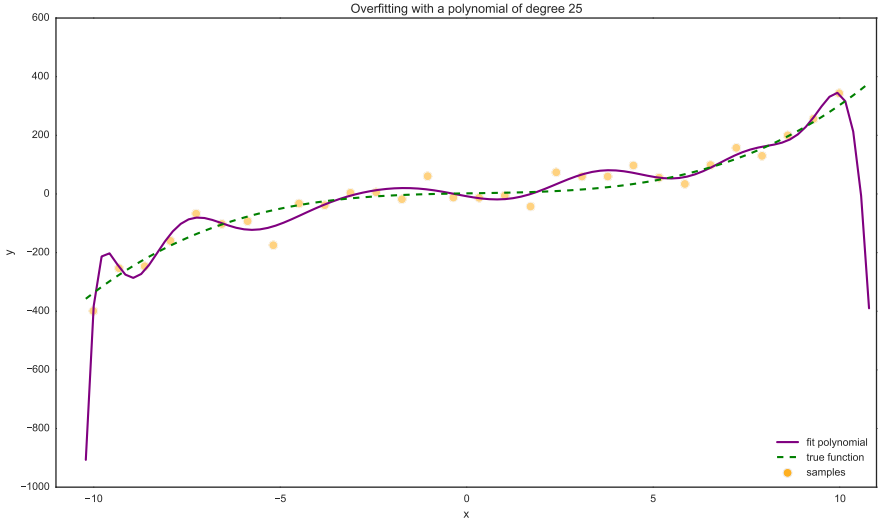
This leads to poor generalization: model fits the training data well but does poor on unseen data.

Can happen when the model has too many parameters (and too little data to train on).

Remember **model selection**: use validation data (or cross validation) to check for overfitting!



Overfitting



Curse of dimensionality

Curses of dimensionality (large D):

1. **Estimation**: more parameters to estimate (risk of overfitting).
2. **Sampling**: exponential increase in volume of space.
3. **Optimization**: slower, larger space to search.
4. **Distances**: everything is far away.
5. Harder to model data distribution $P(x_1, x_2, \ldots, x_D)$
6. Exponentially harder to compute integrals.
7. Geometric intuitions break down.
8. Difficult to visualize.

Blessings of dimensionality:

- **Linear separability**: easier to separate things in high-D using hyperplanes

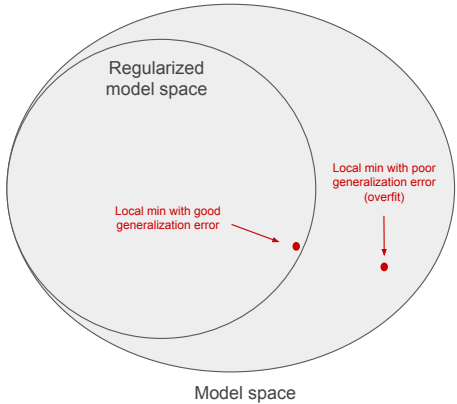
Regularization

Structural risk minimization

Structural risk minimization: prevent overfitting by balancing model complexity with success at fitting training data.

Idea: Given two models (hypotheses) with similar training error, prefer ones with lower complexity.

Regularization: add additional information to solve an ill-posed problem or prevent overfitting. Additional information is often a **penalty** included in the loss function.



L2 regularization

Add a penalty to the loss function for large weights.

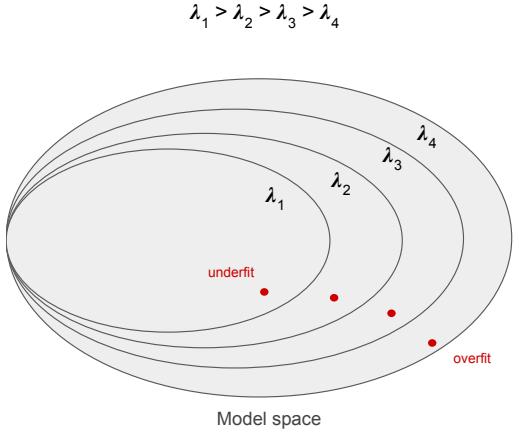
Penalty is on the L_2 norm of the weights

$$\|\mathbf{w}\|_2^2 = \mathbf{w}^T\mathbf{w} = \sum_{i=1}^{D} w_i^2$$

The modified loss is:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

where λ is the **regularization parameter**, which controls the strength of the regularization



L2 regularization and gradient descent

L_2 regularized loss:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Taking the gradient wrt. $\mathbf{w}$ we get

$$\nabla_{\mathbf{w}}\mathcal{L} = \nabla_{\mathbf{w}}\mathcal{L}_{\text{data}} + \lambda\mathbf{w}$$

which gives the gradient descent update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha\nabla_{\mathbf{w}}\mathcal{L}_{\text{data}} - \alpha\lambda\mathbf{w}_t$$

This can be understood as slowly *decaying* the weights toward zero with each iteration. In the neural networks literature L_2 regularization is known as **weight decay**.

Why the L2 penalty is reasonable

Imagine we had a training set for which the first two features are always equal $x_1 = x_2$. E.g. this could happen due to a broken sensor, or an image with a watermark.

	x_1	x_2	x_3	x_4	...	x_N	y
$\mathbf{x}_1$	1	1	4	7	...	2	0
$\mathbf{x}_2$	1	1	2	1	...	6	1
$\mathbf{x}_3$	1	1	9	2	...	5	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

If the decision function $f(x) = \mathbf{w}^T\mathbf{x} + b = w_1x_1 + w_2x_2 + \ldots + w_Nx_N + b$, then we can set $w_1 = -w_2$ to any constant we want want without changing the outcome or the loss.

Why the L2 penalty is reasonable

Lets say we choose $w_1 = 1000, w_2 = -1000$.

We observe a test point in which $x_2 = 0$. Now we have $f(x) = 1000x_1 + 1000(0) + \ldots$, which changes the value of the decision function by $+1000$!

With large weights, small changes in input can produce large changes in output!

If we chose $w_1 = w_2 = 0$, the error on the training set would be the same and there would have been no such problem with the test point.

The L_2 penalty encourages finding solutions with **smaller weights**.

Ridge regression

Applying the L_2 penalty to linear regression gives an model called **ridge regression**. The ridge regression loss is:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}(y - f(\mathbf{x}))^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

The optimal parameters are the solution to

$$\arg\min_{\mathbf{w}} \mathcal{L}$$

As with linear regression, this can be solved in closed form by taking derivatives and setting to zero. The solution turns out to be:

$$\hat{\mathbf{w}} = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

Choosing the regularization parameter

The regularization parameter λ controls the tradeoff between having small weights and fitting the training data.

- Smaller values for λ place more emphasis on fitting the training data and may be more suitable in low-noise settings or when there are few parameters to fit.
- Larger values for λ place more emphasis on making the values of $\mathbf{w}$ small, and are more suitable when there is a lot of noise or many parameters to fit.

As usual, the best way to choose an appropriate value for λ is using model selection procedures like a hold-out validation set or cross-validation.

Notes on L2 regularization

L_2 regularization is also known as **Tikhonov regularization**.

L_2 regularization in ridge regression corresponds to a Gaussian prior on the weights with zero mean and variance proportional to $1/\lambda$.

Can be used to find least squares solutions to ill-posed problems:

- more free variables than constraints,
- $(X^T X)$ is singular,
- $(X^T X + \lambda I)$ is invertible.

Essential for overparameterized problems like softmax regression.

L1 regularization

L_2 regularization encourages solutions with small weights.

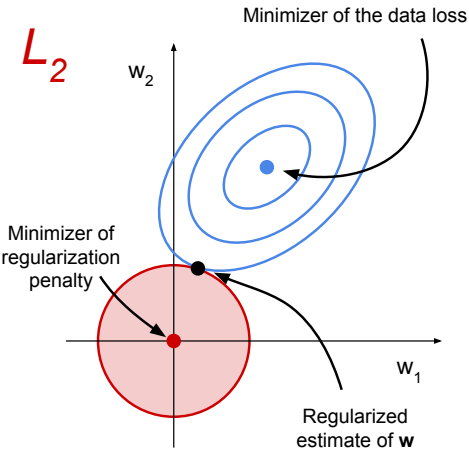
Sometimes you want to find solutions in which most of the weights are exactly zero. These are called **sparse** solutions.

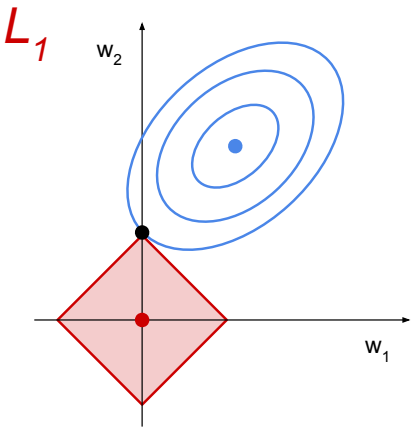
E.g. **feature selection**: you have many features, and you want your decision function to only use a subset. Any feature with weight zero is not used.

L_1 regularization encourages sparse solutions by using penalizing the L_1-norm of the weights.

$$\|\mathbf{w}\|_1 = \sum_{i=1}^{D} |w_i|$$

How does L1 regularization encourage sparsity?





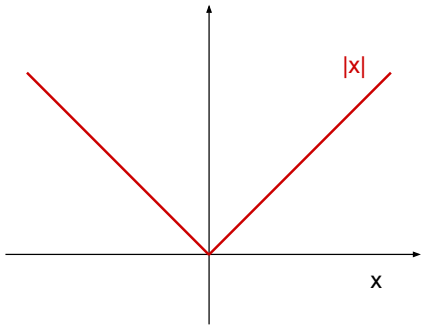
(Sub)-gradient descent for L1 regularization

L_1 regularized loss:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_{\text{data}} + \lambda \|\mathbf{w}\|_1 \\ &= \mathcal{L}_{\text{data}} + \lambda \sum_{i=1}^{D} |w_i|\end{aligned}$$

The absolute value has a discontinuity at zero so gradient is not defined everywhere.

For gradient descent we can use a **subgradient**.



(Sub)-gradient descent for L1 regularization

A subgradient for $|w|$ is:

$$\nabla_w |w| = \begin{cases} 1 & w > 0 \\ -1 & w < 0 \\ 0 & w = 0 \end{cases}$$

which can be written more succinctly using the sign function $\nabla_w |w| = \text{sign}(w)$.

This gives us the subgradient descent update rule for L_1 regularization:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} \mathcal{L} - \lambda \sum_{i=0}^{D} \text{sign}(w_i)$$

The LASSO and Elastic Net

Combining ordinary least squares regression with the L_1 penalty gives rise to the so called **LASSO** (least absolute shrinkage and selection operator).

Can solve using subgradient descent as shown. Generally preferable to use a faster algorithm like **least angle regression** (LARS).

It is also possible to use **both** L_1 and L_2 regularizers together. This model is called the **elastic net**.

In scikit-learn

- `sklearn.linear_model.Ridge`
- `sklearn.linear_model.Lasso`
- `sklearn.linear_model.LassoLars`
- `sklearn.linear_model.ElasticNet`

Logistic regression in scikit-learn can also take an L_1 or L_2 regularization penalty: `LogisticRegression(penalty='l2', C=1.0)`, where $C > 0$ is the **inverse** of λ (smaller values specify stronger regularization).

Scikit-learn also has classes that do automatic (and optimized) cross validation to figure out the best value for the regularization parameter(s). E.g.:

- `sklearn.linear_model.RidgeCV`
- `sklearn.linear_model.LassoCV`

Multiple outputs

Multiple regression

What if the target is not a scalar $y \in \mathbb{R}$ but actually a vector $\mathbf{y} \in \mathbb{R}^K$?

Easy. Just train K separate models, one for each target y_k.

Can solve the least squares problem in one shot by stacking the y_k values into a matrix:

$$Y = \begin{bmatrix} | & | & & | \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_K \\ | & | & & | \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & y_{13} & \cdots & y_{1K} \\ y_{21} & y_{22} & y_{23} & \cdots & y_{2K} \\ \vdots & \vdots & \vdots & & \vdots \\ y_{N1} & y_{N2} & y_{N3} & \cdots & y_{NK} \end{bmatrix}$$

and then solving the normal equations:

$$(X^T X)W = X^T Y$$
$$W = (X^T X)^{-1} X^T Y$$

Multi-class classification

Logistic regression is limited to **binary outputs** $\{0, 1\}$

Often if we want to do **multi-class** classification:

- Output is one of K classes $\{1, 2, \ldots, K\}$

Examples:

- Handwritten digit recognition $\{0, 1, 2, 3, 4, \ldots, 9\}$
- Audio: word/phoneme recognition
- Text classification: document is about $\{\text{politics}, \text{religion}, \text{sports}, \text{fashion}, \ldots\}$
- Action classification: person is $\{\text{walking}, \text{running}, \text{sitting}, \text{standing}, \text{jumping}, \ldots\}$
- Sign language recognition

Multi-class classification

How do we model this setting?

Two approaches:

1. **One-vs-rest** (OVR): aka one-vs-all
2. **Softmax regression**

One-vs-rest (OVR)

Want to train a logistic regression classifier for K classes.

Idea: train K separate binary classifiers.

E.g. classes are: $\{1, 2, 3\}$:

1. Train first classifier $f_1(\mathbf{x})$ with $y = 1$ for class 1 and $y = 0$ for classes $\{2, 3\}$.
2. Train second classifier $f_2(\mathbf{x})$ with $y = 1$ for class 2 and $y = 0$ for classes $\{1, 3\}$.
3. Train third classifier $f_3(\mathbf{x})$ with $y = 1$ for class 3 and $y = 0$ for classes $\{1, 2\}$.

At predict time, output class with **highest probability**:

$$\hat{y} = \arg\max_k f_k(\mathbf{x})$$

Softmax regression

In OVR classification the resulting probabilities do not sum to one (are not a distribution).

This corresponds to independent random output variables.

Sometimes this is what you want:

- multi-label classification: target may be more than one class
- target may be some other unseen class.

Sometimes it is not:

- Digit classification: target must be one of $\{0, 1, \ldots, 9\}$. Never both. Never none.
- Target is a distribution: probabilities should sum to one $\sum_{i=1}^{K} y_i = 1$

Softmax regression

The latter can be achieved using **softmax regression**, which is the direct extension of logistic regression to the multi-class case.

Encode target values $\mathbf{y}$ as a one-hot vector. E.g. $\mathbf{y} = (0, 0, 1, 0)$

The **softmax** activation is the extension of the sigmoid to the multi-class setting.

$$\hat{\mathbf{y}} = f(\mathbf{x}) = \mathrm{softmax}(\mathbf{z})$$

with

$$\mathbf{z} = \begin{bmatrix} \mathbf{w}_1^T \mathbf{x} + b_1 \\ \mathbf{w}_2^T \mathbf{x} + b_2 \\ \cdots \\ \mathbf{w}_K^T \mathbf{x} + b_K \end{bmatrix} = W\mathbf{x} + \mathbf{b}$$

Softmax regression

The softmax activation function is the analogue of the sigmoid for more than two classes:

$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_{j=1}^{K} \exp(x_j)} \begin{bmatrix} \exp(x_1) \\ \exp(x_2) \\ \vdots \\ \exp(x_K) \end{bmatrix}$$

Softmax regression

Loss function for softmax regression is the **categorical cross entropy**, which is the extension of binary cross entropy to categorical distributions:

$$\begin{aligned}\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) &= -\mathbf{y}^T \log \hat{\mathbf{y}} \\ &= -\sum_{j=1}^{K} y_i \log \hat{y}_i\end{aligned}$$

Unlike logistic regression, softmax regression is **overparameterized**:

- probabilities must sum to one,
- one extra set of weights and biases than needed,
- L_2 regularization is important to ensure unique minimizer.

Relationship between the softmax and the sigmoid

Consider the case when there are only two outputs $\mathbf{x} = [x_1 \ x_2]^T$. The softmax output for the first variable is:

$$\mathrm{softmax}(\mathbf{x})_1 = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{1}{1 + e^{x_2 - x_1}}.$$

But, since $\mathrm{softmax}(\mathbf{x})_2 = 1 - \mathrm{softmax}(\mathbf{x})_1$ the model is overparameterized and we can eliminate one of the outputs. Setting $x_2 = 0$ gives:

$$\mathrm{softmax}(\mathbf{x})_1 = \frac{1}{1 + e^{-x_1}} = \sigma(x_1).$$

Summary

Properies of linear models

Require $(D + 1)$ parameters for D features.

Strong assumptions mean they have **high bias**.

Have fairly **low variance**, but still possible to overfit when D is large relative to N.

Regularization can be used to reduce size of hypotheses space and help prevent overfitting.

Very fast at predict time: just a linear function.

Interpretable: weights specify feature importance (careful with this).

Further reading

The elements of statistical learning:

- Chapter 3: Linear methods for regression
- Chapter 4: Linear methods for classification

Resources

Stanford machine learning lectures (Andrew Ng):

- Lecture 3: linear and logistic regression
  http://www.youtube.com/watch?v=HZ4cvaztQEs
- Lecture 4: Generalized linear models
  http://www.youtube.com/watch?v=nLKOQfKLUks

Caltech machine learning lectures (Yaser Abu-Mostafa):

- Lecture 3: The linear model 1
  http://www.youtube.com/watch?v=FIbVs5GbBlQ
- Lecture 9: The linear model 2
  http://www.youtube.com/watch?v=qSTHZvN8hzs
- Lecture 12: Regularization
  http://www.youtube.com/watch?v=I-VfYXzC5ro

Resources

Oxford deep learning lectures (Nando de Freitas):

- Lecture 2: Linear models
  `http://www.youtube.com/watch?v=DHspIG64CVM`
- Lecture 3: Maximum likelihood
  `http://www.youtube.com/watch?v=kPrHqQzCkg0`
- Lecture 4: Regularization 1
  `htts://www.youtube.com/watch?v=VR0W_PNwLGw`
- Lecture 5: Regularization 2
  `http://www.youtube.com/watch?v=VR0W_PNwLGw`