EEN1083/EEN1085

Data analysis and machine learning I

Ali Intizar

Semester 1
2024/2025

DCU
Ollscoil Chathair Bhaile Átha Cliath
Dublin City University

Linear Models

DCU Ollscoil Chathair Bhaile Átha Cliath Dublin City University

Outline

Outline

1. Introduction
2. Linear Regression
3. Linear Classification
4. Features
5. Multi-class Regression

Introduction

What are linear models?

Linear models are **parameteric models** that are characterized by having prediction functions of the form:

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

- $\mathbf{w}, b$ are parameters. These need to be learned from the training data
- $\mathbf{w}$ is called the **weight vector**
- b is a **scalar bias**
- $g()$ is an activation function, which may be the identity (i.e. $g(x) = x$).
- if $\mathbf{x} \in \mathrm{R}^D$ then $\mathbf{w} \in \mathrm{R}^D$ and there are $(D + 1)$ parameters.

Example

Assume we have two features, $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \in \mathbb{R}^2$ and we are using the identity activation function $g(\mathbf{x}) = \mathbf{x}$.

The decision function is:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + b \\ &= \sum_{i=1}^{D} w_i x_i + b \\ &= w_1 x_1 + w_2 x_2 + b \end{aligned}$$

There are three free parameters to be learned: $\theta = \{w_1, w_2, b\}$

Note that it is also possible to prepend a 1 to the feature vector, e.g. $\mathbf{x} = \begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix}^T \in \mathbb{R}^3$. We can then write the decision function more succinctly as:

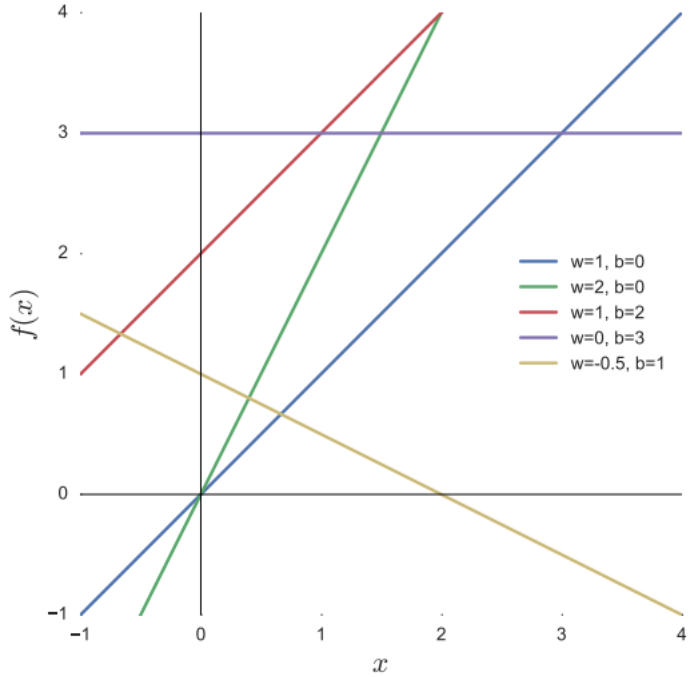
$$f(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x})$$

And the free parameters are now $\theta = \{w_0, w_1, w_2\}$, with w_0 being the implicit bias.

Models like this are called linear models because $\mathbf{w}^T\mathbf{x} + b$ (or $\mathbf{w}^T\mathbf{x} + b = 0$) defines **a line**.

- Plane or **hyperplane** in higher dimension

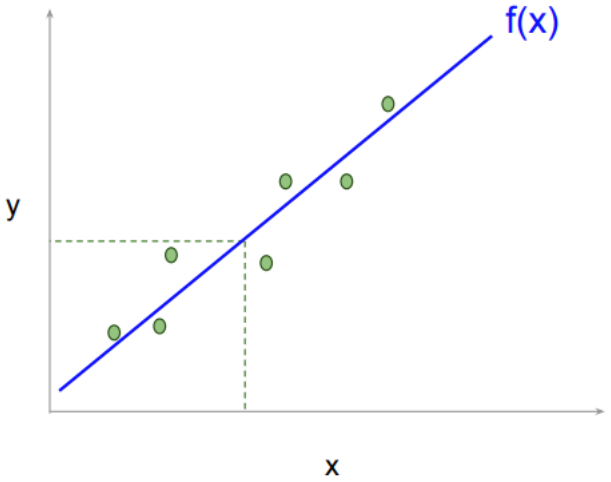
The slope (normal) of the line is given by $\mathbf{w}$. The y-intercept is given by b in regression.

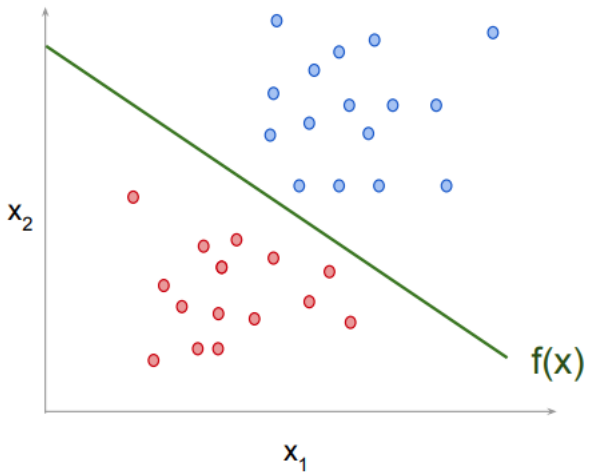


Classification and regression with linear models

When using a linear model for regression, we simply output the value of $\hat{y} = \mathbf{w}^T \mathbf{x} + b$

When using a linear model for classification, we are interested in which side of the line $f(\mathbf{x}) = c$ the target appears on.





Why linear models?

The linear assumption seems extremely rigid. It appears at first glance that linear models would be of very limited use.

Turns out that linear models are extremely useful!

Reasons:

1. Even if there is no linear relationship (or decision boundary), often possible to **project** data to a space in which relationship is linear.
2. If you have very little data, you need to make strong assumptions to fit a model. **Low variance**.
3. **Occam's razor** (principle of parsimony). Simpler models that explain the data are better than more complicated ones.
4. Possible to build much more complex models using linear models as a building block. This is the basis of **neural networks** and **deep learning**.

What can you do with linear models?

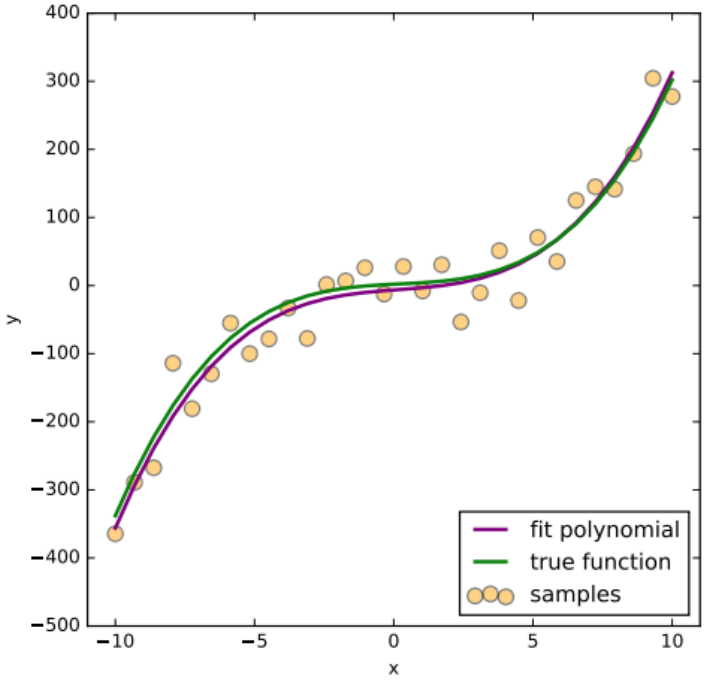
Obviously, fit lines.

But also:

- Polynomials
- Circles

Anything that is **linear in the weights**.

Will see how to do this later.



Linear Regression

Linear regression

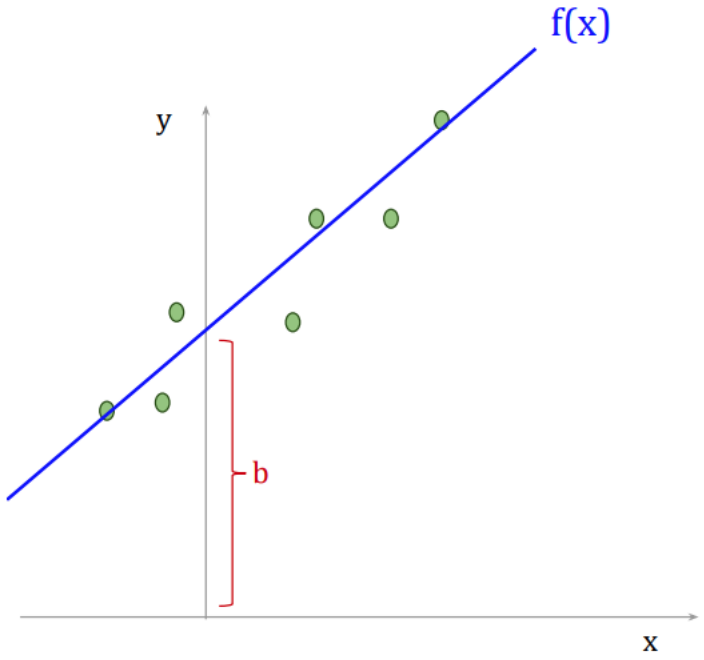
Regression: output variable y is continuous real values $y \in \mathrm{R}$.

Decision function:

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

In 1D, f is a line. In 2D a plane. In N-D a hyperplane.

$\mathbf{w}$ defines the slope of the line, b defines its offset from the origin on the y-axis.



Linear regression

In linear regression (aka **ordinary least squares**), we would like to find parameters $\{\mathbf{w}, b\}$ such that the resulting line $f(\mathbf{x})$ fits the training data well.

Need to define what it means to fit the data well.

A natural way to do this is using squared error. The squared error for the prediction $\hat{y} = f(\mathbf{x})$ is:

$$(y - \hat{y})^2$$

Total squared error for the training set $T = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ is the sum over the individual squared errors:

$$\frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

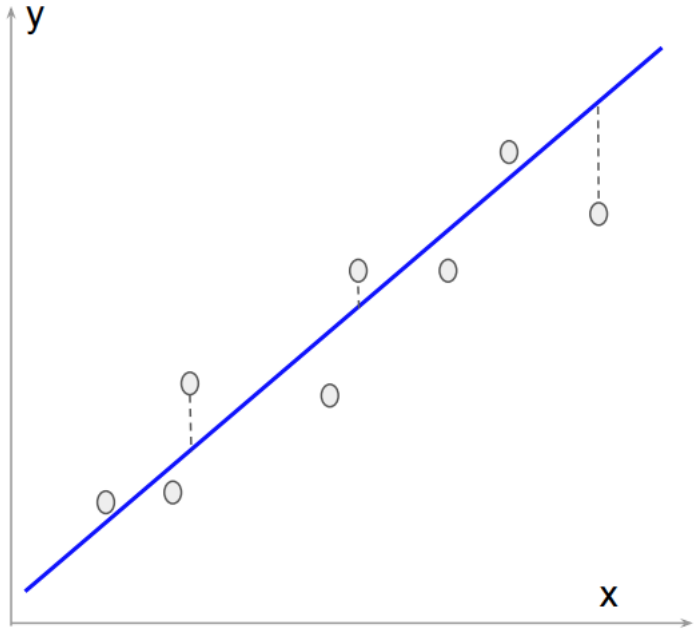
Linear regression

Define the **loss** (cost) function as:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

In linear regression, we want to find the $\theta = \{\mathbf{w}, b\}$ that **minimize** this squared error loss.

$$\hat{\theta} = \arg\min_{\mathbf{w},b} L(\mathbf{w}, b)$$



Linear regression

How do we minimize the loss?

From calculus, we know that the minimum value of a function is at a **critical point**: a place where the gradient is zero:

$$\frac{\partial L}{\partial w_i} = 0 \quad \text{and} \quad \frac{\partial L}{\partial b} = 0$$

Approach: take gradients and set them to zero, then solve for **w** and b in closed form.

This is possible and leads to what are called the **normal equations** for **ordinary least squares**.

Normal equations

Start by expressing the difference between the target and predictions on the training set X, $\mathbf{y}$ in vector form:

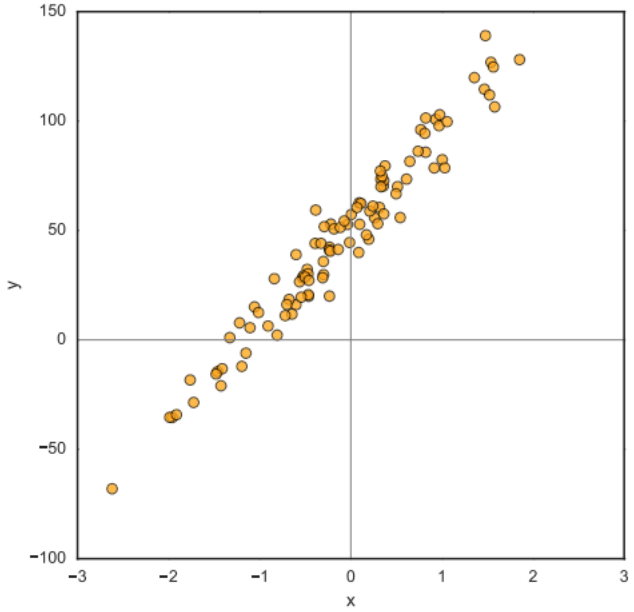
$$\mathbf{e} = \mathbf{y} - X\mathbf{w}$$

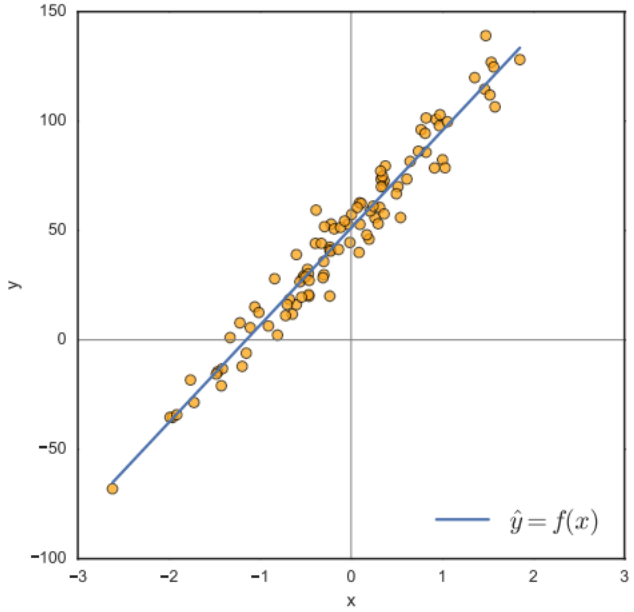
with X being our training examples with a one prepended to each row. This allows the bias parameter to be treated like any other weight.

$$X = \begin{bmatrix} 1 & \mathbf{x}_1^T \\ 1 & \mathbf{x}_2^T \\ \vdots & \vdots \\ 1 & \mathbf{x}_N^T \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Normal Equation $\mathbf{w} = (X^T X)^{-1} X^T \mathbf{y}$

Example



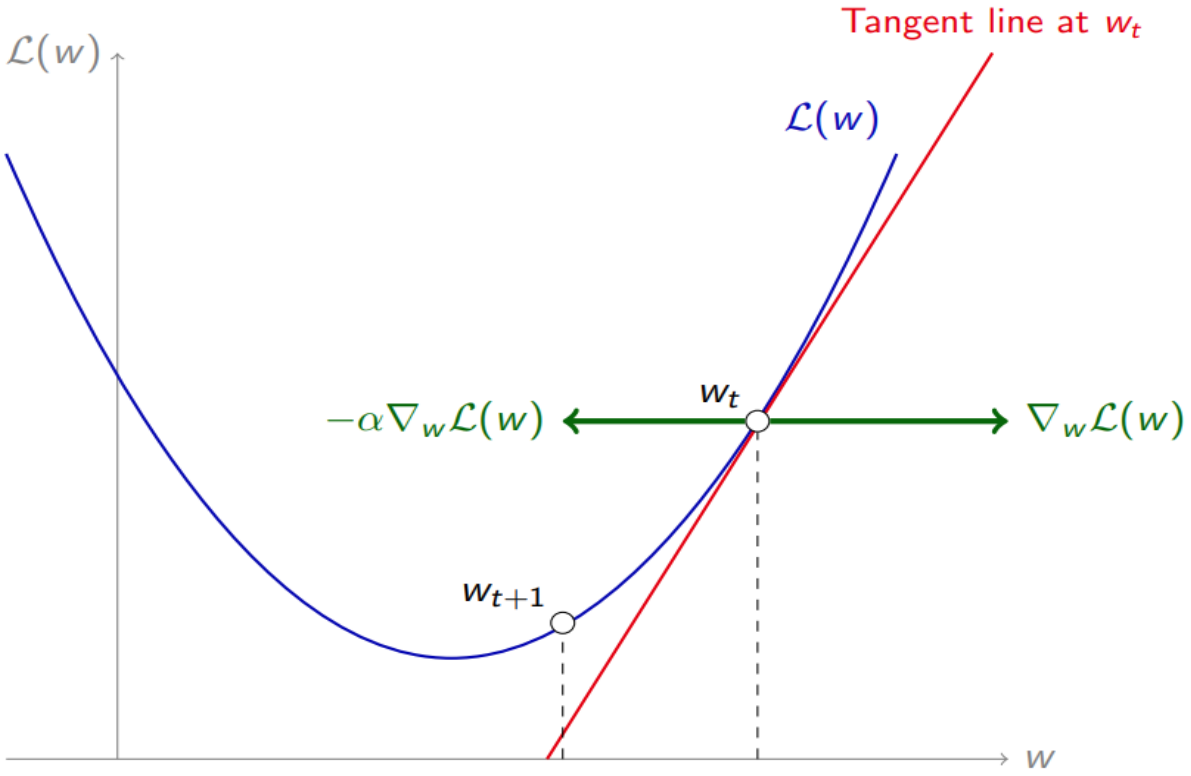


w = 44, b = 51

Gradient descent

- Often in machine learning we will take a similar approach to producing an algorithm as we did for linear regression.
- That is, specify a loss function and then figure out how to minimize it.
- Usually, however, the approach of setting the gradients equal to zero and solving for the parameters closed form won't work.
- An alternative, more general approach, is to start with a random guess for the parameters, and then **iteratively improve** this guess.
- One very common way of doing this is **gradient descent** (steepest descent).

Gradient descent: the idea



Gradient descent: the algorithm

Algorithm

1. Start with a random guess for the initial parameters θ_0
2. While not converged:
   2.1 Set $\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta L(\theta_t)$

The value of α_t is called the **learning rate**. Decide convergence

when:

- The change in θ is small $|\theta_{t+1} - \theta_t| < T$
- Maximum number of iterations are reached.
- Loss stops improving.

Gradient descent for linear regression

Loss function:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i))^2$$

Gradient of the loss function wrt. $\mathbf{w}$:

$$\begin{aligned}\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) &= \frac{1}{2}\sum_{i=1}^{N}\nabla_{\mathbf{w}}(y_i - f(\mathbf{x}_i))^2 \\ &= \sum_{i=1}^{N}(f(\mathbf{x}_i) - y_i)\mathbf{x}_i\end{aligned}$$

Gradient of the loss function wrt. b:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial b} &= \frac{1}{2}\sum_{i=1}^{N}\partial_b(y_i - f(\mathbf{x}_i))^2 \\ &= \sum_{i=1}^{N}(f(\mathbf{x}_i) - y_i)\end{aligned}$$

Gradient descent for linear regression

Update rules:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \sum_{i=1}^{N} (f(\mathbf{x}_i) - y_i)\mathbf{x}_i$$

$$b_{t+1} = b_t - \alpha_t \sum_{i=1}^{N} (f(\mathbf{x}_i) - y_i)$$

Note: may want to divide learning rate α_t by N to make learning independent of number of data points.

Properties of gradient descent

The learning rate α_t:

- Gradient descent will converge on convex functions as long as the learning rate α_t keeps getting smaller over time.
- Gradient descent may be very slow if α_t is too small.
- Gradient descent may diverge if α_t is too large.

Gradient descent requires a full pass over the dataset each time it updates the parameters (weights and biases). Slow for large datasets!

Stochastic gradient descent

Idea: Instead of computing the gradient of the loss for the full dataset, estimate it using a **single randomly chosen example** and then take step in that direction.

Total loss:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i))^2$$

Loss for a single example $\mathbf{x}_i$:

$$\frac{1}{2}(y_i - f(\mathbf{x}_i))^2$$

SGD update rules:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t(f(\mathbf{x}_i) - y_i)\mathbf{x}_i \qquad b_{t+1} = b_t - \alpha_t(f(\mathbf{x}_i) - y_i)$$

Properties of SGD

Usually much faster than batch gradient descent, since we update the weights every iteration instead of every epoch.

Can operate on large datasets, since we only need to process a single example at a time.

Important to shuffle training data to ensure the gradient estimates are unbiased (i.i.d.)

But: gradient updates are noisy, since it is only an estimate of the full gradient.

Convergence may be slower as you near the optimum solution.

Linear regression

Assumptions of linear regression:

- Linear relationship between variates and covariate
- All other variation can be modelled as zero mean Gaussian noise

If these assumptions are violated, then linear regression may be a bad model.

Implications:

1. Linear regression is sensitive to extreme outliers.
2. Linear regression may not be appropriate when the noise is not Gaussian.
3. Linear regression may not give good results if the relationship between the variates and covariates is not approximately linear.

Noise: any unmeasured quantities, including random variation and measurement error.

Linear classification

Linear classification

Recall the binary classification setup:

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

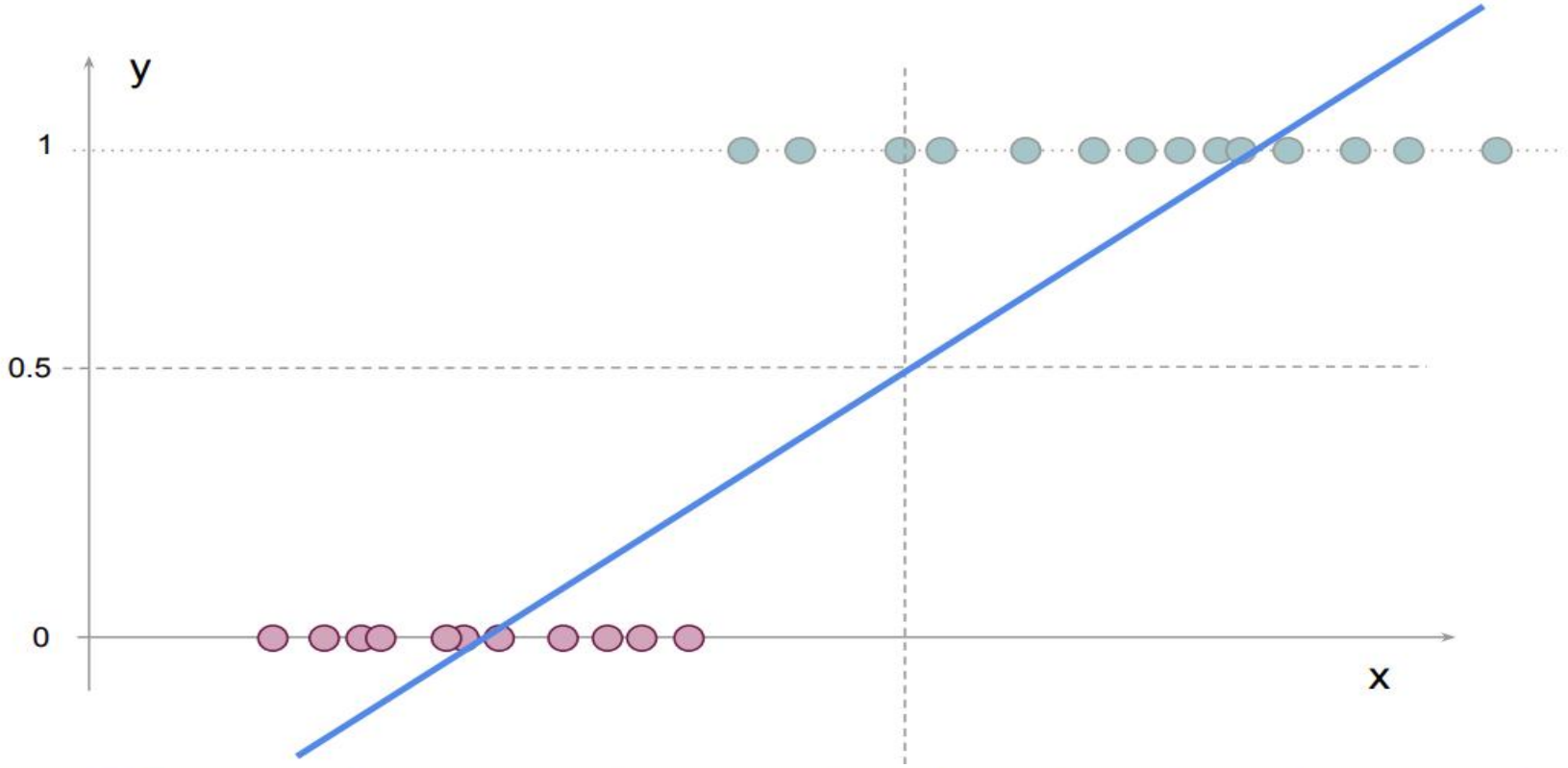
Q: Is it possible to use our linear regression model to do classification?

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



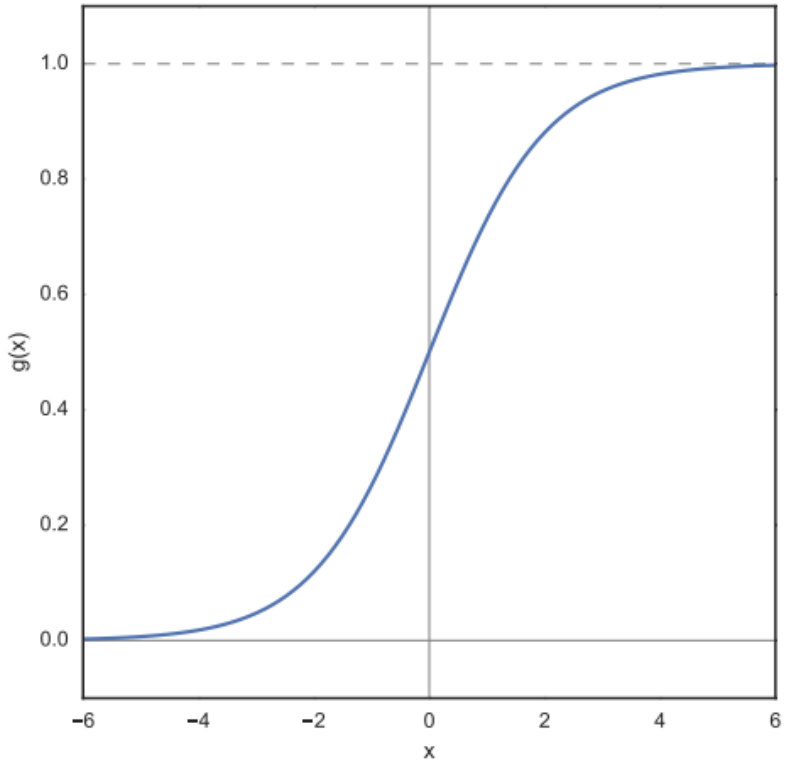
Logistic regression

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

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

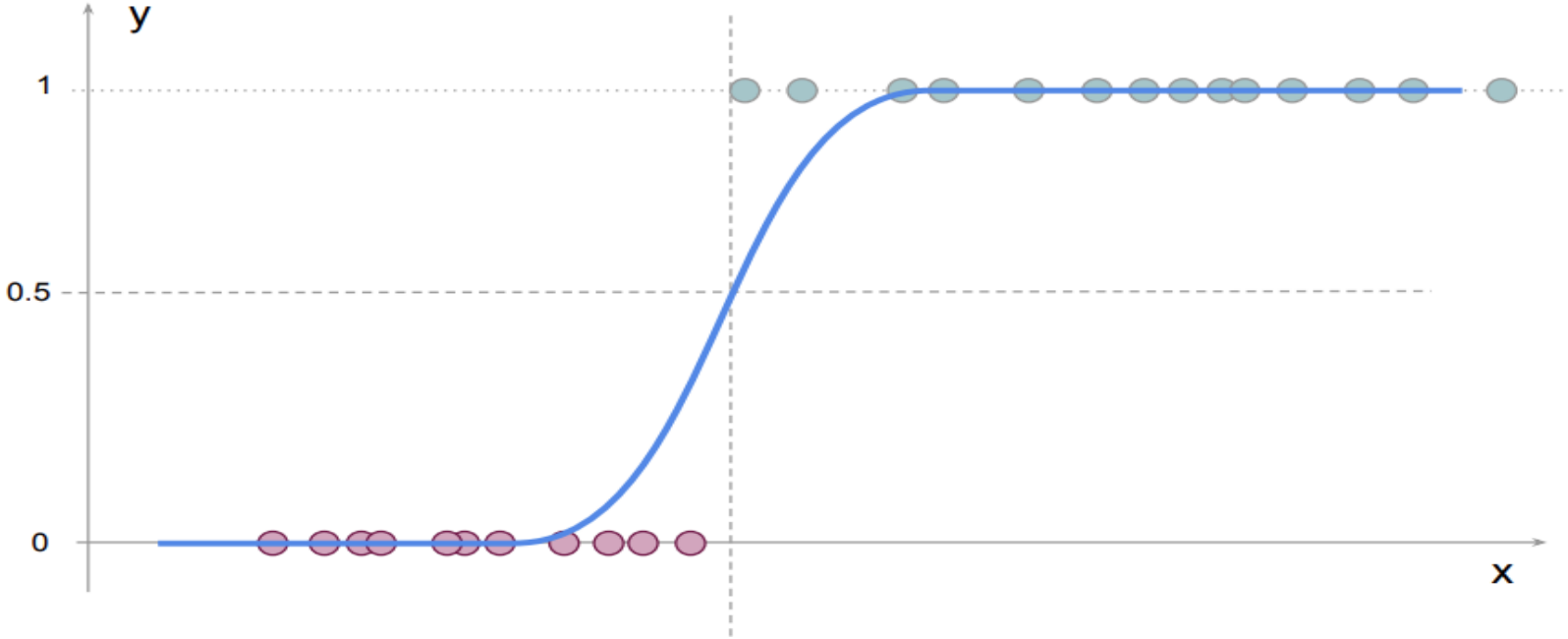
with $g(.)$ being the sigmoid (logistic) function, which "squashes" its input to the range $(0, 1)$:

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



Logistic regression

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



Logistic regression: Loss Functions

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).

Probability Distribution

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

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

Binary cross entropy loss

a loss function to minimize 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

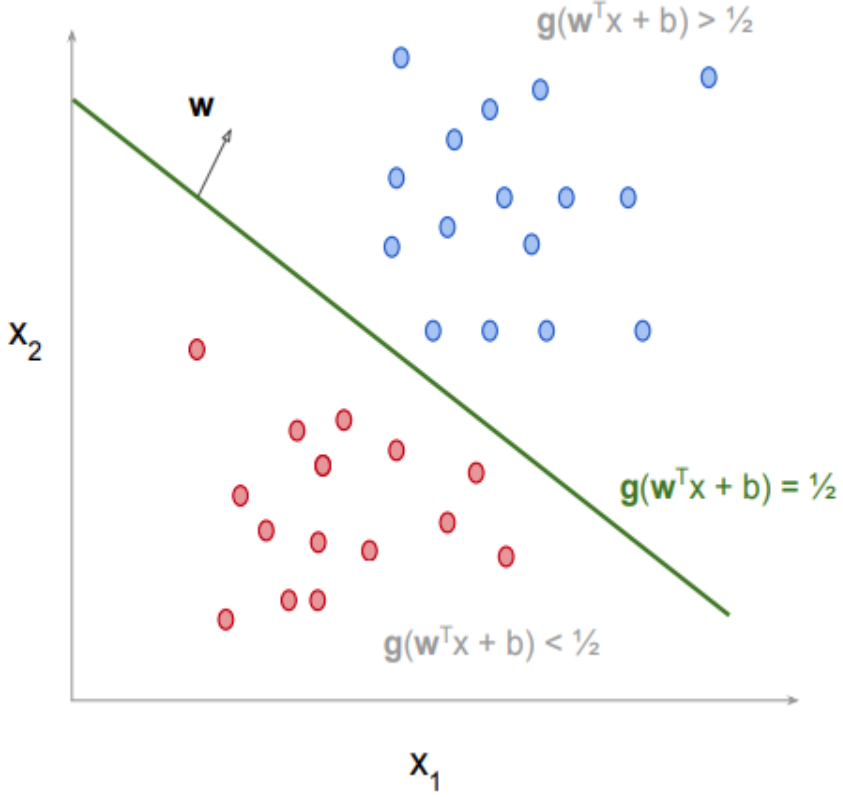
$$f(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x} + b)$$

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

$$\nabla_\theta L = \frac{\partial L}{\partial \theta_i}$$

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

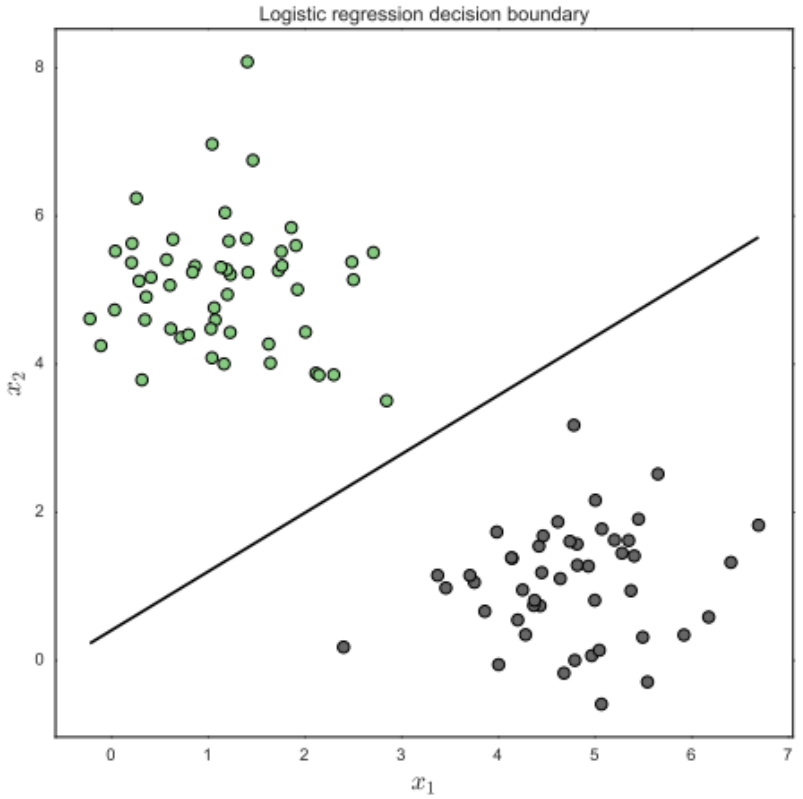
$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta_t)$$

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!

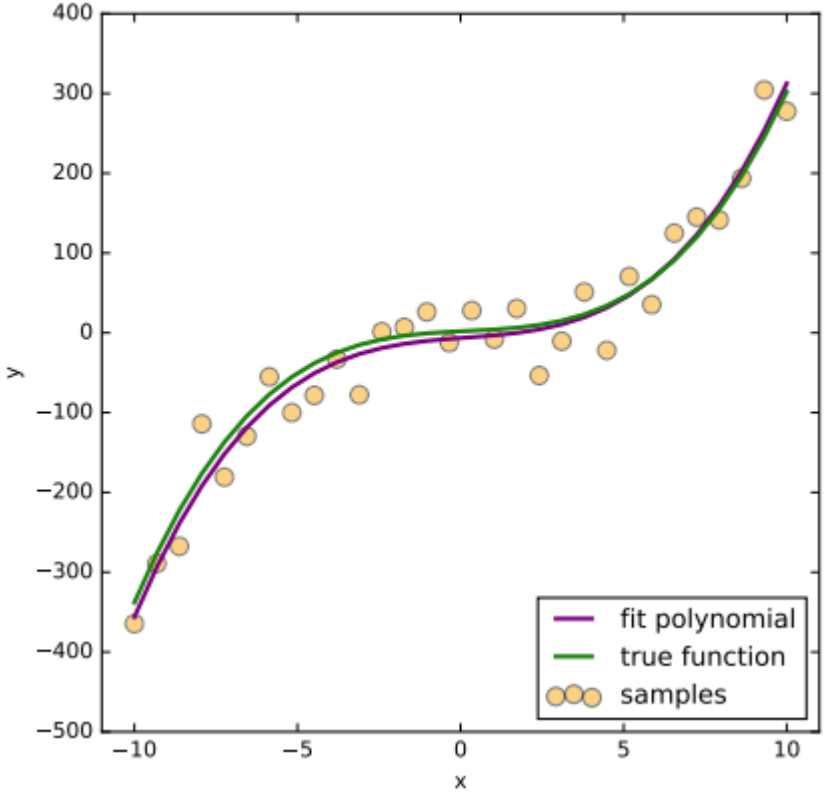
Only needs for function to be linear in the parameters

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 which contain powers of all input variables and cross terms (if necessary).

Nonlinear mappings

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

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

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 **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 $\varphi(\mathbf{x})$.

Feature engineering

Designing a good feature mapping function $\varphi(\mathbf{x})$ 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

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

Examples

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

Feature learning

Feature engineering is notoriously difficult!

Alternative: learn features from the data directly.

Known as **representation learning**: $\varphi(\mathbf{x})$ 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

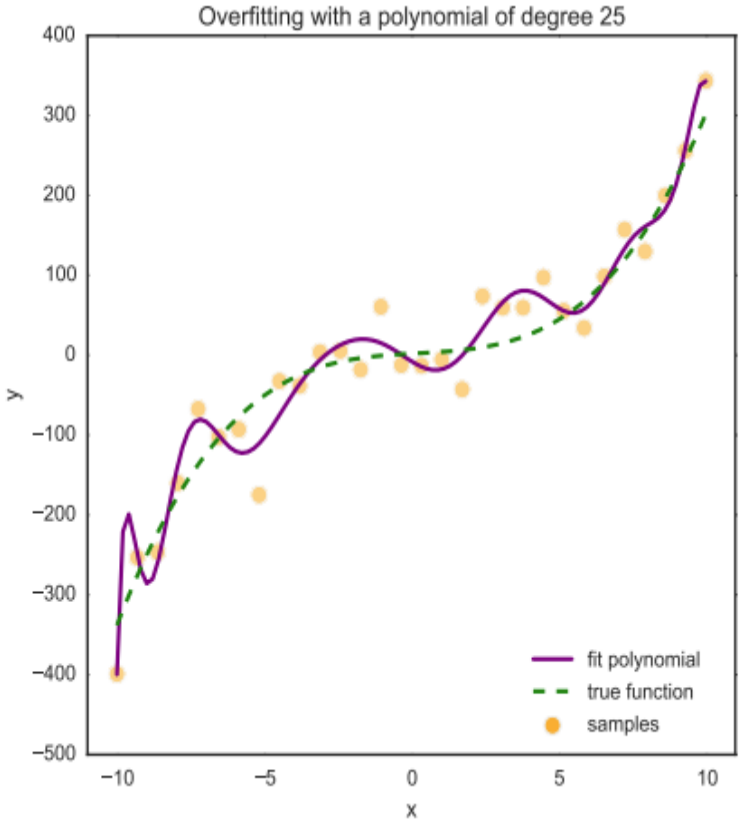
If the model has too many degrees of freedom, you can end up fitting not only the patterns of interest, but also the **noise**.

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

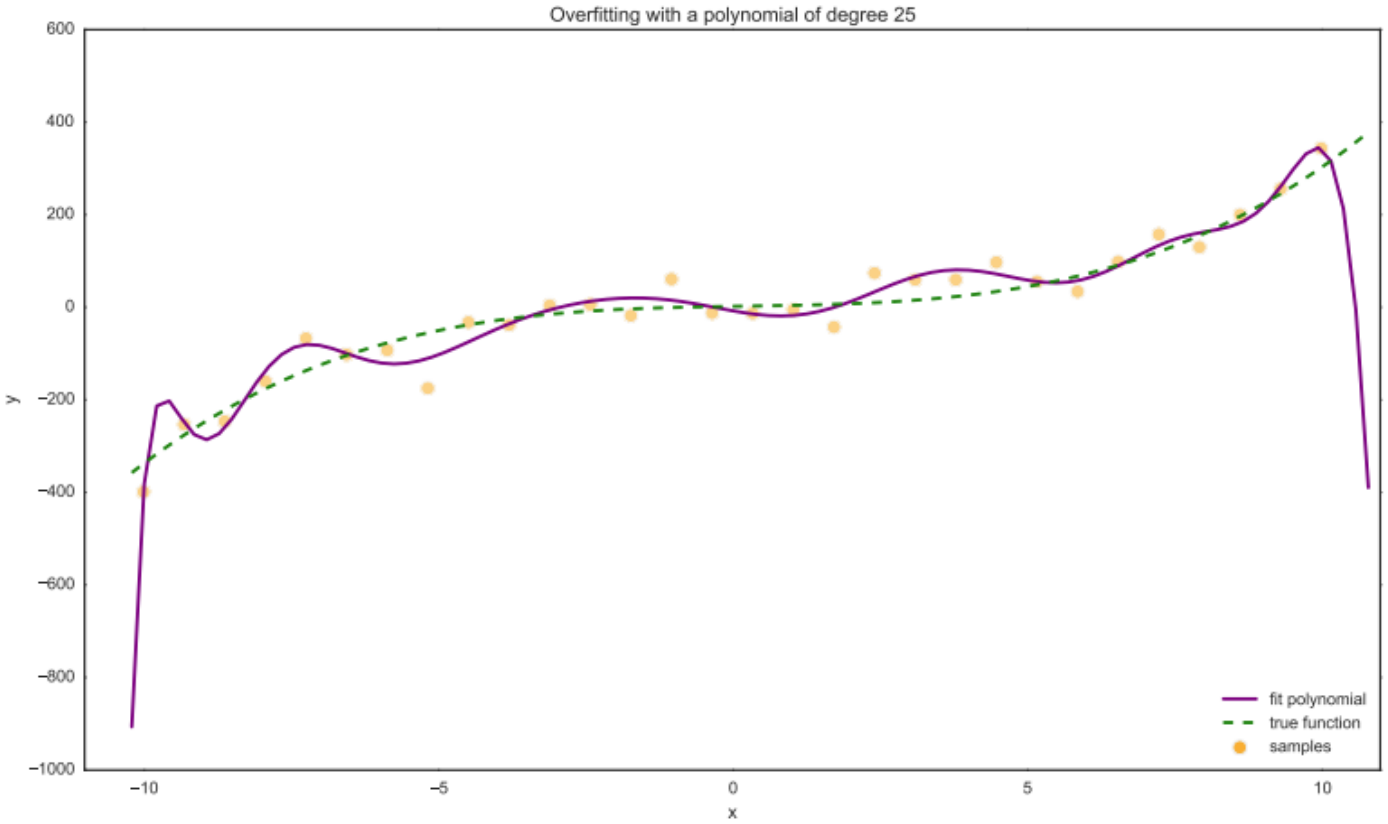
This leads to poor generalization: model fits the training data well but does poor on unseen data

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

Multiple outputs

Multiple regression

What if the target is not a scalar $y \in \mathrm{R}$ but actually a vector $\mathrm{y} \in \mathrm{R}^K$?
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 $\{$politics, religion, sports, fashion, $\ldots\}$
- Action classification: person is $\{$walking, running, sitting, standing, 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.

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

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.

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
  http://www.youtube.com/watch?v=VR0W_PNwLGw
- Lecture 5: Regularization 2
  http://www.youtube.com/watch?v=VR0W_PNwLGw