Previously...

Simple linear regression

Inputs: one attribute: $x_1, x_2, \ldots, x_n$.
Expected outputs: $y_1, y_2, \ldots, y_n$.
We are looking for parameters θ_0, θ_1, such that
$J(\theta_0, \theta_1) = \sum_{i=1}^{n} (\theta_0 + \theta_1 x_i - y_i)^2$ is smallest as possible.

Previously...

Training

Option 1: solve system of equations and get

$$\theta_1 = \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i - n \sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i - n \sum_{i=1}^{n} x_i^2} \qquad \theta_0 = \frac{1}{n} \left(\sum_{i=1}^{n} y_i - \theta_1 \sum_{i=1}^{n} x_i \right)$$

Option 2: gradient descent:

$$\theta_0 = \theta_0 - \alpha \frac{\partial J}{\partial \theta_0} = \theta_0 - \alpha \sum_i (\theta_0 + \theta_1 x_i - y_i)$$
$$\theta_1 = \theta_1 - \alpha \frac{\partial J}{\partial \theta_1} = \theta_1 - \alpha \sum_i (\theta_0 + \theta_1 x_i - y_i) x_i$$

J is convex function, it has at most one local minimum, which is also global and both methods will find same solution (apart from numerical errors).

Prediction from new input

$$y_{new} = \theta_0 + \theta_1 x_{new}$$

Generalized linear regression

We use column vectors for now.
We extend each input with attribute with value 1 (to simplify a lot of things).
Our model is:

$$y = \vec{x}^T \cdot \vec{\theta}$$

Generalized linear regression

We use column vectors for now.
We extend each input with attribute with value 1 (to simplify a lot of things).
Our model is:

$$y = \vec{x}^T \cdot \vec{\theta}$$

Each input will make one row in matrix and expected outputs will be a column vector:

$$X = \begin{pmatrix} (\vec{x}^{(1)})^T \\ (\vec{x}^{(2)})^T \\ \dots \\ (\vec{x}^{(n)})^T \end{pmatrix} \qquad \vec{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(n)} \end{pmatrix}$$

Matrix magic

$$X\vec{\theta} - \vec{y} = \begin{pmatrix} (\vec{x}^{(1)})^T\vec{\theta} - y^{(1)} \\ (\vec{x}^{(2)})^T\vec{\theta} - y^{(2)} \\ \cdots \\ (\vec{x}^{(n)})^T\vec{\theta} - y^{(n)} \end{pmatrix}$$

$$(X\vec{\theta} - \vec{y})^T(X\vec{\theta} - \vec{y}) = \sum_{i=1}^{n}((\vec{x}^{(i)})^T\vec{\theta} - y^{(i)})^2 = J(\vec{\theta})$$

Gradient

Gradient definition:

$$\nabla_{\vec{\theta}} J = \left(\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \ldots, \frac{\partial J}{\partial \theta_n} \right)$$

Shows direction up (i.e. if you move parameters this way, loss will increase).

Gradient of error

$$J(\vec{\theta}) = \sum_{i=1}^{n}((\vec{x}^{(i)})^T\vec{\theta} - y^{(i)})^2$$

One part of the gradient:

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=0}^{n} 2((\vec{x}^{(i)})^T\vec{\theta} - y^{(i)})x_j{}^{(i)}$$

Using matrices:

$$\nabla_{\vec{\theta}} J = 2X^T(X\vec{\theta} - \vec{y})$$

Matrix magic - conclusion

We want to have:

$$\nabla_{\vec{\theta}} J = 2X^T(X\vec{\theta} - \vec{y}) = \vec{0}$$

$$X^T X\vec{\theta} = X^T\vec{y}$$

$$\vec{\theta} = (X^T X)^{-1} X^T\vec{y}$$

These are called normal equations for linear regression.

Source code

```
import numpy as np

X = [[122, 3], [39, 1], [67, 3]]
y = [400000, 76000, 175000]

X = np.hstack([np.array(X, float),
               np.ones(shape=(len(y),1))])
y = np.array(y, float)

XXi = np.linalg.inv(X.T.dot(X))
theta = XXi.dot(X.T).dot(y)
print(theta)

print(np.linalg.solve(X.T.dot(X), X.T.dot(y)))
```

Time complexity

- $X^T X$ – $O(m^2 n)$
- Inversion of matrix / solving system of linear equations – $O(m^3)$.

Numerical methods - gradient descent

We iterate:

$$\vec{\theta} = \vec{\theta} - \alpha \nabla_{\vec{\theta}} J$$

After substiting for our gradient (factor 2 is hidden in α):

$$\vec{\theta} = \vec{\theta} - \alpha X^T (X\vec{\theta} - \vec{y})$$

Stochastic gradient descent

Instead of calculation error and gradient from all training examples, we do update after each example (we calcuate gradient from one example):

- `while (not converged):`
  - `for i in range(n):`
    - $\theta = \theta - \alpha \vec{x}^{(i)}((\vec{x}^{(i)})^T \theta - y^{(i)})$

It usually converges faster than vanilla gradient descent. But, you need to decrease alpha over time (this is not needed for vanilla gradient descent).

Summary

Linear regression

Inputs: rows in matrix X.
Expected outputs: vector $\vec{y}$.
We are looking for parameters $\vec{\theta}$, such that $E = (X\vec{\theta} - \vec{y})^T(X\vec{\theta} - \vec{y})$ was smallest as possible.

Training

Option 1: solve system of equations $X^T X\vec{\theta} = X^T\vec{y}$
Option 2: (stochastic) gradient descent: $\vec{\theta} = \vec{\theta} - \alpha X^T(X\vec{\theta} - \vec{y})$

E is convex function, it has at most one local minimum, which is also global and both methods will find same solution (apart from numerical errors).

Prediction from new input

$y_{new} = \vec{x}_{new}^T \cdot \vec{\theta}$

Other models

Still regression (one real number as an output).
Sometimes data are nonlinear.

- Locally weighted linear regression
- Polynomial regression and its reduction on linear
- Neural nets (not today)

Weighed linear regression

Each example gets a weight. We minimize:

$$E = \sum_{i=1}^{n} w_i((\vec{x}^{(i)})^T\vec{\theta} - y^{(i)})^2 = (X\vec{\theta} - \vec{y})^T I_w (X\vec{\theta} - \vec{y})$$

$$I_w = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & w_n \end{pmatrix}$$

Weighed linear regression

Each example gets a weight. We minimize:

$$E = \sum_{i=1}^{n} w_i((\vec{x}^{(i)})^T\vec{\theta} - y^{(i)})^2 = (X\vec{\theta} - \vec{y})^T I_w (X\vec{\theta} - \vec{y})$$

$$I_w = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & w_n \end{pmatrix}$$

We want zero gradient:

$$\nabla_{\vec{\theta}} E = 2X^T I_w (X\vec{\theta} - \vec{y}) = \vec{0}$$

Solution:

$$X^T I_w X\vec{\theta} = X^T I_w \vec{y}$$

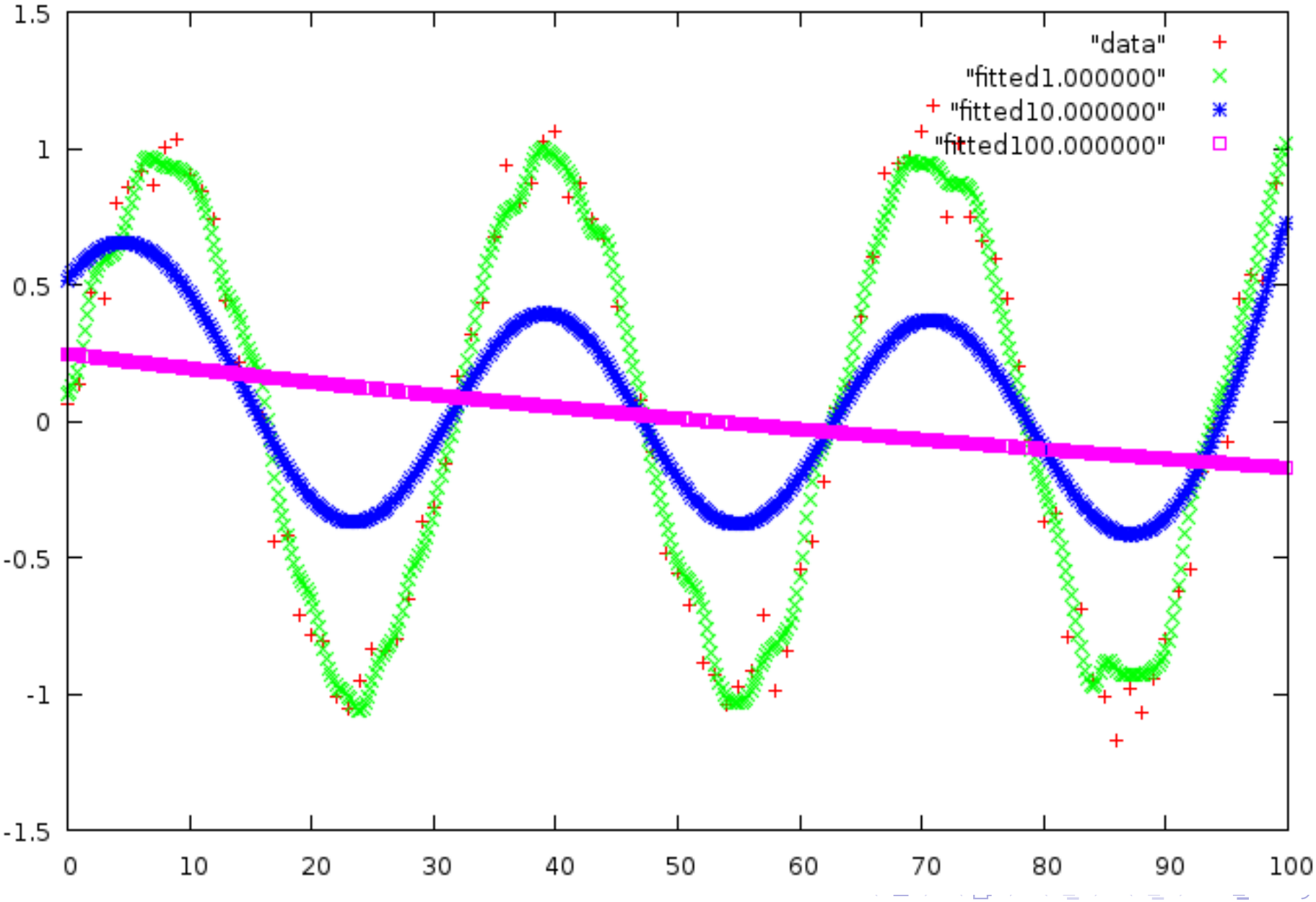
Locally weighted linear regression

We want to predict value at $\vec{x}_{new}$.

We put: $w_i = e^{\frac{-\|\vec{x}_{new} - \vec{x}^{(i)}\|^2}{\sigma^2}}$ (σ is an hyperparameter, which should be set separatelly)

We run weighted linear regression and predict the value (yes we do new training for each new output).

LVLR result


"data"
"fitted1.000000"
"fitted10.000000"
"fitted100.000000"
1.5
1
0.5
0
-0.5
-1
-1.5
0
10
20
30
40
50
60
70
80
90
100

Polynomial regression

One input x, model with degree 2:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$

Polynomial regression

One input x, model with degree 2:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$

Two inputs x_1, x_2, model (up to degree 2):

$$y = \theta_0 + \theta_{10} x_1 + \theta_{01} x_2 + \theta_{11} x_1 x_2 + \theta_{20} x_1^2 + \theta_{02} x_2^2$$

We can use same procedure as last time and find values of θ. Or reduce the problem to linear regression.

Reduction of polynomial regression

For two inputs

Input: $(1, x_1, x_2)$ we change into:
$(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$
And we can solve linear regression (we do not change outputs).

Reduction of polynomial regression

For two inputs

Input: $(1, x_1, x_2)$ we change into:
$(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$
And we can solve linear regression (we do not change outputs).

In general

We have p basis functions: $\phi_1(\vec{x}), \phi_2(\vec{x}), \ldots, \phi_p(\vec{x})$, kde $\phi_i \in \mathbb{R}^m \to \mathbb{R}$.
We preprocess input matrix X into matrix Φ:

$$\begin{pmatrix} \phi_1(\vec{x}^{(1)}) & \phi_2(\vec{x}^{(1)}) & \ldots & \phi_p(\vec{x}^{(1)}) \\ \phi_1(\vec{x}^{(2)}) & \phi_2(\vec{x}^{(2)}) & \ldots & \phi_p(\vec{x}^{(2)}) \\ & & \vdots & \\ \phi_1(\vec{x}^{(n)}) & \phi_2(\vec{x}^{(n)}) & \ldots & \phi_p(\vec{x}^{(n)}) \end{pmatrix}$$

And we solve linear regression, for example the system: $\Phi^T\Phi\vec{\theta} = \Phi^T\vec{y}$

Basis fuctions - examples

Not only polynomials.

- $\phi(\vec{x}) = x_4 x_7$, $\phi(\vec{x}) = x_2$
- 0-1 functions: $\phi(\vec{x}) = x_6 > 0$
- Some preprocessings: $\phi(\vec{x}) = \log(x_5 + 1)$
- Kernel fuctions: $\phi(\vec{x}) = e^{\frac{-\|\vec{z}-\vec{x}\|^2}{\sigma^2}}$

Linear regression with preprocessing