Supervised Machine Learning

→ What is Machine Learning?

Machine Learning is the field of study that gives the computer the ability to learn without being explicitly programmed - Arthur Samuel (1959)

→ Machine Learning Algorithms

* Supervised learning
* Unsupervised learning
* Recommender Systems
* Reinforcement learning

→ Supervised learning

Input (X)	Output (Y)	Application
email	Spam? (0/1)	Spam filtering
audio	text transcripts	Speech recognition
English	Spanish	Machine translation
ad, user info	Click? (0/1)	Online advertising
image, radar info	position of other cars	self driving car
image of phone	defect ? (0/1)	visual inspection

Supervised learning algorithms works on labeled data. (Includes both input features and output labels)

Regression is a type of supervised learning algorithm that learns to predict numbers out of infinitely many possible numbers.

Classification is a type of supervised learning algorithm that learns to predict a small finite and limited set of possible output categories.

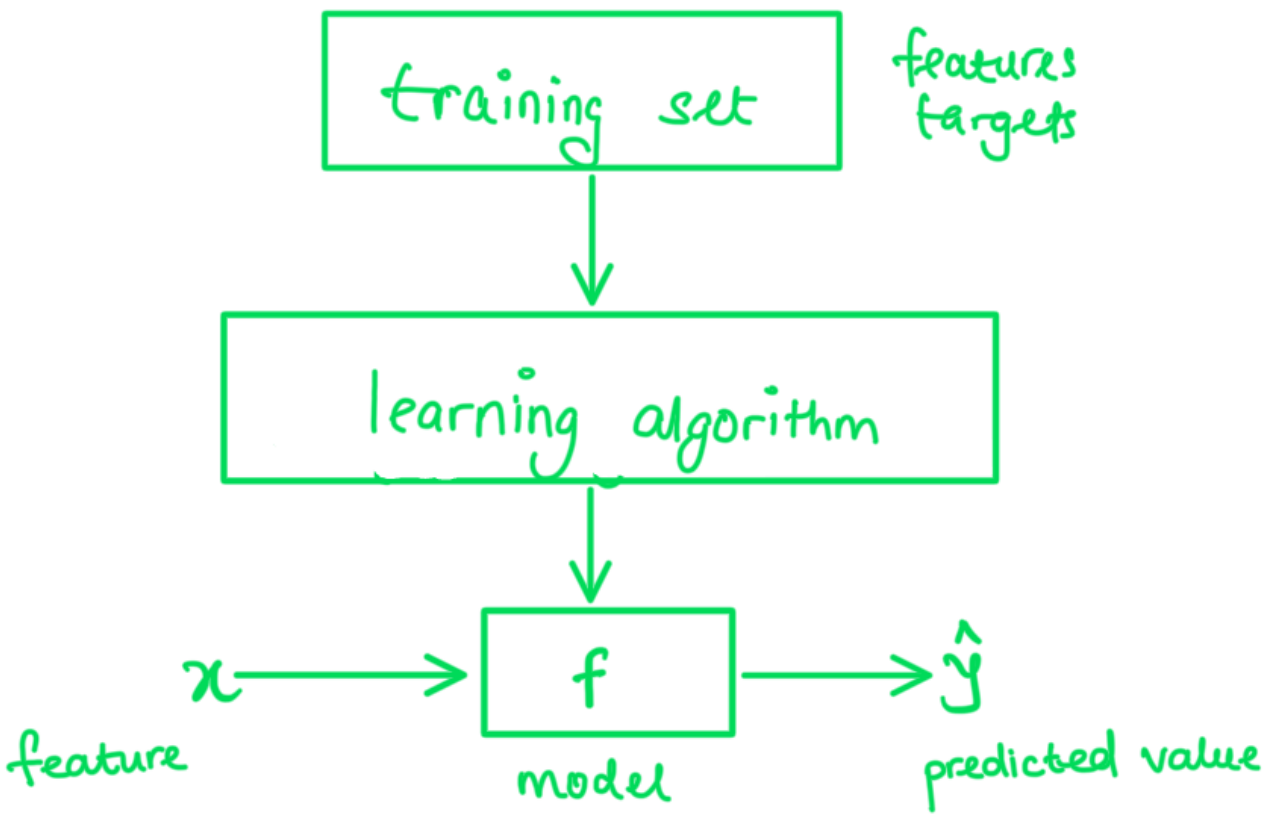
→ Unsupervised learning

Unsupervised learning works on unlabeled data.

Clustering algorithm is a type of unsupervised learning algorithm which takes data without labels and tries to automatically group them into clusters.

Apart from clustering, we also have anomaly detection and dimensionality reduction.

→ Linear Regression



$$f_{w,b}(x) = f(x) = wx + b$$

→ Cost function

lost function measures how well the model is doing, we use the squared error lost function:

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

The goal is to minimize $J(w,b)$ as a function of w and b:

→ Gradient descent

$$w = w - \alpha \frac{\partial}{\partial w} J(w,b) \quad \text{and} \quad b = b - \alpha \frac{\partial}{\partial b} J(w,b)$$

α is called the learning rate and is always a positive number.

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x^{(i)} \quad \text{and} \quad \frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

→ Learning rate (α)

If α is too small, gradient descent will work but it will be slow.

If α is too large, gradient descent may overshoot and may never reach the minimum. (fail to converge or diverge).

Gradient descent can reach local minimum with fixed learning rate because as we get close to a local minimum, the derivatives will become smaller and the update steps will beame smaller.

Linear regression with multiple input variables

x_j = j^{th} feature

n = number of features

$\vec{x}^{(i)}$ = features of the i^{th} training example

$x_j^{(i)}$ = value of feature j in i^{th} training example.

Model:

$f_{w,b}(x) = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$

Let: $\vec{w} = [w_1 \; w_2 \; \cdots \; w_n]$

$\vec{x} = [x_1 \; x_2 \; \cdots \; x_n]$

$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$

dot product

```
f = 0
for i in range(0,n):
    f = f + w[i] * x[i]
f = f + b
```

Vectorization

$f = np.dot(w, x) + b$

[runs faster than loop and uses numpy - able to use parallel hardware]

Gradient Descent

$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$ and $b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$

$w_j = w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$

and $b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right)$

Normal Equation (An alternative to gradient descent)

Good for:

* Only for linear regression
* Solve for w, b without iterations.

Disadvantages:

* Doesn't generalize to other learning algorithms
* Slow when number of features is large ($n > 10,000$)

What we need to know:

* Normal equation method may be used in ML libraries that implement linear regression.
* Gradient descent is the recommended method for finding parameters w and b.

Feature Scaling

Different feature scale can slow down gradient descent. So we have to scale to similar ranges to improve gradient descent speed and model training efficiency.

How to scale?

* Mean normalization

Let μ_j be the mean for the j^{th} feature and $a \leq x_j \leq b$.

$$x_j \rightarrow \frac{x_j - \mu_j}{b - a} \qquad \begin{cases} b \simeq \max \\ a \simeq \min \end{cases}$$

* Z-score normalization

$$x_j \rightarrow \frac{x_j - \mu_j}{\sigma_j}$$

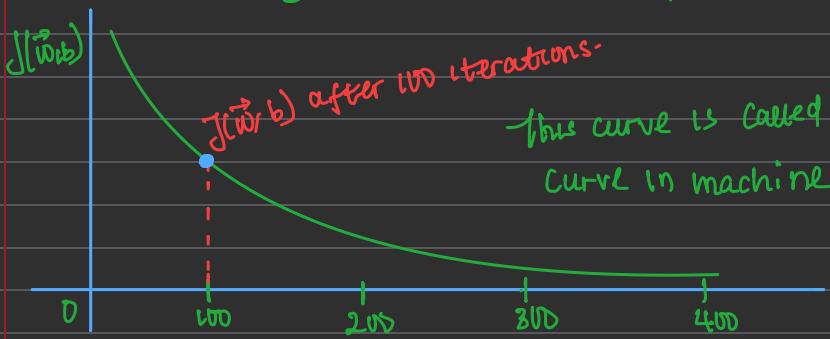
We mostly want to aim for the range $-1 \le x_j \le 1$ for each feature x_j but others like $-3 \le x_j \le 3$ and $-0.3 \le x_j \le 0.3$ are completely okay as well.

$0 \le x_1 \le 3$	okay, no rescaling
$-2 \le x_2 \le 0.5$	okay, no rescaling
$-100 \le x_3 \le 100$	too large → rescale
$-0.001 \le x_4 \le 0.001$	too small → rescale
$98.6 \le x_5 \le 105$	too large → rescale

Checking gradient descent for convergence

Objective : $\min_{\vec{w},b} J(\vec{w},b)$

Plot $J(\vec{w},b)$ against the number of iterations



* If gradient descent is implement properly, cost $J(\vec{w},b)$ should decrease after every iteration.

* If $J(\vec{w},b)$ should increase after one iteration, that means either α is chosen poorly and it usually means α is too large or there could be bug in the code.

* By the time we reach like 300 iterations, the cost is leveling off and no longer decreasing much and by 400 iterations, it has flattened out and it means gradient descent has converged.

We can also use an automatic convergence test.

Let ε be 10^{-3}. If $J(\vec{w}, b)$ decreases by $\leq \varepsilon$ in one iteration, declare convergence. Choosing ε is pretty difficult and so learning curve is more reliable to observe convergence

Choosing learning rate

With small enough α, $J(\vec{w}, b)$ should decrease on every iteration
If α is too small, gradient descent takes a lot more iteration to converge.
We can try: ... 0.001 0.01 0.1 1 ...

Feature Engineering

$$f = w_1 x_1 + w_2 x_2 + b$$

x_1: frontage, x_2: depth

We might have the intuition that area = frontage x depth and which is a better representation of the house and so create a new feature $x_3 = x_1 x_2$ and so:

$$f = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Feature engineering is using intuition to design new features by transforming or combining original features.

Polynomial Regression.

Suppose we want to fit polynomials to our dataset, we can use feature engineering to create new features as powers of our original features.

Classification

In classification problems, we want to answer the question whether something belongs to a class/category or not like whether an email is spam or not.

Logistic Regression

Logistic regression is a classification algorithm that predicts the probability of a binary outcome. Despite the name regression, it is used for classification problems - like predicting whether an email is spam(1) or not(0).

The core idea is to take a linear combination of features, then squash it through a function that outputs a number between 0 and 1, which we interpret as probability.

the model:

* We get the linear combination (same as linear regression):

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = wx + b$$

* Squash through the sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

* The full model:

$$f_{w,b}(x) = \sigma(wx+b) = \frac{1}{1+e^{-(wx+b)}}$$

The output is between 0 and 1, interpreted as $P(y=1|x)$

Once we have a probability output

$$\hat{y} = \begin{cases} 1 & \text{if } f_{w,b}(x) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

0.5 is the default threshold. But we can adjust depending on whether we care more about false positives or false negatives.

$f(x) \geq 0.5$ is equivalent to $z \geq 0$, which is equivalent to $wx+b \geq 0$. So decision boundary is the line (or hyperplane) where $wx+b=0$

Loss function

The squared error doesn't work well for logistic regression, instead we use the binary cross entropy (also called log loss):

$$L(w,b) = \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(f(x^{(i)})\right) + (1-y^{(i)}) \log\left(1 - f(x^{(i)})\right) \right]$$

This looks ugly but it is actually elegant.

If $y=1$, the loss is $-\log(f(x))$. If $f(x) \to 1$, loss $\to 0$, if $f(x) \to 0$, loss $\to \infty$

If $y=0$, the loss is $-\log(1-f(x))$. If $f(x) \to 0$, loss $\to 0$, if $f(x) \to 1$, loss $\to \infty$

So confident wrong predictions gets punished extremely heavily, and confident right predictions gets rewarded.

Why this loss specifically?

* It comes from MLE under the Bernoulli assumption. If we assume each y is drawn from a Bernoulli distribution with parameter $f(x)$, the likelihood is:

$$P(y|x) = f(x)^y (1-f(x))^{1-y}$$

Taking negative log gives cross entropy. So minimizing cross entropy is maximum likelihood estimate for logistic regression

* It makes the optimization landscape convex. Squared error with a sigmoid gives a non-convex loss with a local minima. Cross entropy with sigmoid gives a convex loss with a single global minima. Gradient descent converges faster.

Calculating gradients

$$\frac{\partial L}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(f(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left(f(x^{(i)}) - y^{(i)} \right)$$

Regularized logistic regression

We can add L1 or L2 regularization

* L2 (Ridge logistic regression)

$$L_{reg}(w, b) = L(w, b) + \frac{\lambda}{2m} \sum w_j^2$$

L1 (Lasso logistic regression):

$$L_{reg}(w,b) = L(w,b) + \frac{\lambda}{m}\sum|w_j|$$

Multi-class extensions

Logistic regression is binary by default. For multi-class (3+ classes), we have two approaches:

* One Vs Rest (OVR):
Train one binary logistic regression per class (Class "k" vs everything else). At prediction time, pick the class with the highest probability.

* Softmax regression (Multinomial logistic regression):
Generalize sigmoid to softmax:

$$P(y=k|x) = \frac{e^{w_k x + b_k}}{\sum_j e^{w_j x + b_j}}$$

Steps for softmax

* Compute k linear scores:
For an input x with n features, compute one score per class
$z_k = w_k x + b_k$ for $k = 1, 2, \ldots, K$
We end up with K numbers, one per class. These are called logits.

* Convert logits to probabilities
We need to turn these K logits into K probabilities that:
1. Are all between 0 and 1
2. Sum to exactly 1

The softmax function does this:

$$P(y=k|x) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_k}}$$

Numerical Stability: If logits are large (e.g. $z = 1000$) then e^{1000} overflows to infinity, same as Naive Bayes underflow in the opposite direction. The fix is to subtract the max logits before exponentiating.

How softmax generalizes sigmoid: Softmax for $K = 2$ classes is mathematically equivalent to sigmoid

$$P(y=1) = \frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{z_2 - z_1}} = \frac{1}{1 + e^{-(z_1 - z_2)}}$$

let $z = z_1 - z_2$ and we have sigmoid $\sigma(z)$

Cross entropy loss for softmax

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log P(y=k|x^{(i)})$$

In practice y is one-hot encoded - a vector of zeros with a 1 in the position of the correct class. So for a single example with class 2 as the true label (in a 3-class problem)

* $y = [0, 1, 0]$
* The loss reduces to: $-\log(P(\text{class } 2))$

Gradients of the softmax

$$P_k = \frac{e^{z_k}}{\sum_{s=1}^{K} e^{z_i}}$$

Categorical cross entropy for a single sample with one hot labels:

$$L = -\sum_{k=1}^{K} y_k \log P_k$$

Lets compute $\partial P_k / \partial z_i$

The softmax for class k depends on every logit z_i (because every z appears in the denominator). We have two cases:

Case A When $i = k$ (differentiating P_k with respect to its own logit z_k)

$$P_k = \frac{e^{z_k}}{\sum_j e^{z_j}}, \text{ let } S = \sum_j e^{z_j}, P = \frac{e^{z_k}}{S}$$

$$\frac{\partial P_k}{\partial z_k} = \frac{e^{z_k} \cdot S - e^{z_k} \cdot e^{z_k}}{S^2} = \frac{e^{z_k}}{S} \cdot \frac{S - e^{z_k}}{S} = P_k \cdot (1 - P_k)$$

Case B When $i \neq k$ (differentiating P_k w.r.t a different class's logit z_i)

$$P_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

Now e^{z_k} doesn't depend on z_i, so the numerator is constant w.r.t z_i. Only denominator has z_i

$$\frac{\partial P_k}{\partial z_i} = e^{z_k} \cdot \frac{\partial}{\partial z_i}\left(\frac{1}{S}\right) = e^{z_k} \cdot \frac{-1}{S^2} \cdot e^{z_i} = -\frac{e^{z_k}}{S} \cdot \frac{e^{z_i}}{S} = -P_k \cdot P_i$$

Combining both using kronecker delta:

$$\frac{\partial p_k}{\partial z_i} = p_k(\delta_{ki} - p_i)$$

where $\delta_{ki} = 1$ if $i = k$ and 0 otherwise

$$L = -\sum_{k=1}^{K} y_k \log p_k$$

$$\frac{\partial L}{\partial z_i} = -\sum_{k=1}^{K} y_k \cdot \frac{1}{p_k} p_k(\delta_{ki} - p_i) = -\sum_{k=1}^{K} y_k(\delta_{ki} - p_i)$$

$$= -\sum_{k=1}^{K} y_k \delta_{ki} + \sum_{k=1}^{K} y_k p_i$$

But $\sum y_k \delta_{ki} = y_i$, $\delta_{ki} = 1$ if $k = i$ and 0 otherwise

Also $\sum_{k=1}^{K} y_k = 1$ since y is one-hot

$$\frac{\partial L}{\partial z_i} = -y_i + p_i = p_i - y_i$$