Preliminaries

A vector $\vec{w} \in \mathbb{R}^n$, where $\mathbb{R}^n$ is the n-dimensional vector space over the field of real numbers.

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = [w_1, w_2, \cdots, w_n]^T; \quad ||\vec{w}||_{L_2} = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$$

$$\frac{\partial}{\partial \vec{w}} u = \left[\frac{\partial u}{\partial w_1}, \frac{\partial u}{\partial w_2}, \cdots, \frac{\partial u}{\partial w_n} \right]^T$$

Inner products

$$< \vec{u}, \vec{v} >= \vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2 + \cdots$$

Probability

$$P(Y = y | X = x)$$

X, Y are random variables, x, y are specific realizations (observed values) of those random variables.
Random variable: Maps of outcomes of random processes to numbers.
Random process: Different times, different outcomes (though everything including the inputs are the same).
Regular variable - takes on a single value if the conditions (including inputs) are the same. **Random variable** - can take many values even if the conditions are the same.

Volume 1

The shortest definitions

- Machine Learning: Computers learning from examples. $\vec{x}^{(i)}$=input feature, $y^{(i)}$=target.
- Training set: Examples seen by a machine that it uses to learn.
  $\{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \cdots, (\vec{x}^{(m)}, y^{(m)})\}$
- Test set: Unseen examples for a machine, to test it's learning.
  $\{(\vec{x}^{(1)}, ?), (\vec{x}^{(2)}, ?), \cdots, (\vec{x}^{(m)}, ?)\}$
- Supervised learning: Show a machine features ($\vec{x}^{(i)}$) and targets ($y^{(i)}$), then ask it to learn.
- Unsupervised learning: Show features but not targets (!!?!!), then ask it to learn [1].

Modeling $\rightarrow$ Learning $\rightarrow$ Inference.
$\widehat{y} = f(\vec{x}, \vec{w})$?? $\rightarrow$ Training (value of $\vec{w}$?) $\rightarrow$ Testing ($\widehat{y}$ for test data ?).

Regression

Regression is a supervised machine learning problem. Let's consider the following scenario:

You're planning a trek to a mountain peak (at an altitude of 4500 m), and as you pack, a question crosses your mind: how cold will it be up there? You know the air gets colder as you climb, but by how much? Will it just be chilly, or will it drop below freezing? Figuring this out could make all the difference – packing too much could slow you down, while packing too little might leave you shivering at the top. If only you had a way to predict the temperature before you start climbing!

Alt (km)	Temp (°C)
1	15
2	7
3	3
4	−6
5	−10
6	−20

On your last hike, you recorded temperatures at different altitudes. When you check the numbers, you see a pattern. You use them to understand the trend.

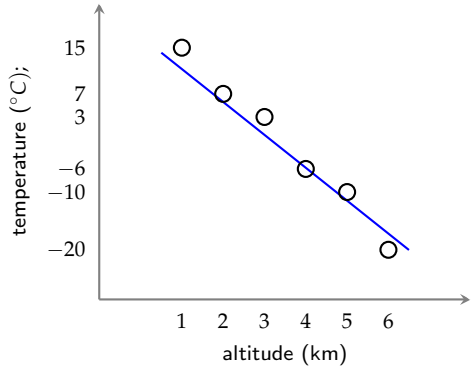
What could the temperature at 4.5 km altitude be? The table above with the altitude-temperature data represents your training data. However, this data doesn't include the temperature at 4.5 km altitude. To estimate this missing temperature, you need a functional relationship—a mathematical function—that connects every value in the domain of altitude ($\mathcal{D}_{alt}$) to the domain of temperature ($\mathcal{D}_{temp}$). Let's denote this function as $h : \mathcal{D}_{alt} \rightarrow \mathcal{D}_{temp}$. Now what might that function look like?

Modeling

Let x represent a feature and y the target variable we aim to predict. We have chosen a linear model, making this a univariate (single-variable - x) linear regression problem. Let $h : \mathcal{D}_x \rightarrow \mathcal{D}_y$ denote the prediction function. Then the predicted y (denoted as $\widehat{y}$) is given as

$$\widehat{y} = h(x) = w_0 + w_1 x$$

Here, w_0 and w_1 are the *model parameters* of the linear regression model.



Here is the plot of the training data. We can observe a linear relationship between altitude and temperature. Therefore, we might model the function h as a line. In mathematical terms, this can be written as $\textit{predicted altitude} = h(\textit{altitude}) = w_0 + w_1 \cdot \textit{altitude}$. Of course, we could have chosen other models (other equations or functions) as well. The key is to select a model that best captures the pattern in the data. This decision $\rightarrow$ *modeling*.

Learning (or training)

In the training phase, we calculate the values of the model parameters (here, w_0 and w_1) given a set of training data. Let $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$ denote a given set of m training samples. The superscripts indicate the sample number in the dataset. Using these, how can we find w_0 and w_1?

Our goal here is to find the best fit line $\widehat{y}$. What is a best fit line? It is the line positioned such that it minimizes the total error between the observed data points $(x^{(i)}, y^{(i)})$ and the predicted values on the line $(x^{(i)}, \widehat{y}^{(i)})$. How do we find that best fit line? By calculating the right values for the model parameters (w_0 and w_1) so that the goal above is achieved. Mathematically, this can be achieved by formulating an objective function and then optimizing (either minimizing or maximizing) it to satisfy the primary goal outlined above. Let us design the objective function $J : \mathbb{R}^2 \rightarrow [0, \infty)$ such that it calculates the mean squared error between the observed data points and the predictions by the linear model:

$$J(w_0, w_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(x^{(i)} - x^{(i)}\right)^2 + \left(\widehat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)}\right)^2$$
$$= \frac{1}{2m} \sum_{i=1}^{m} \left(w_0 + w_1 x^{(i)} - y^{(i)}\right)^2$$

Note that, we have defined the objective function above as a function of the model parameters. Our primary goal of finding the best-fit line will be achieved if we can find the right values of w_0 and w_1 so that $J(w_0, w_1)$ (overall error) is minimized, which gives our optimization problem:

$$\min_{w_0, w_1} J(w_0, w_1) = \min_{w_0, w_1} \frac{1}{2m} \sum_{i=1}^{m} \left(w_0 + w_1 x^{(i)} - y^{(i)}\right)^2$$

How to solve this optimization problem (i.e., find $w_0^*, w_1^* = \arg\min_{w_0, w_1} J(w_0, w_1)$)? One option: the gradient descent algorithm.

Gradient descent algorithm

Start with random values of model parameters w_0 and w_1 (or maybe just zeros)[1]. Next, run the following updates iteratively:

$$w_0 := w_0 - \alpha \frac{\partial}{\partial w_0} J(w_0, w_1) = w_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right)$$
$$w_1 := w_1 - \alpha \frac{\partial}{\partial w_1} J(w_0, w_1) = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right) x^{(i)}$$

The parameter α represents the gradient descent step-size, which is a hyperparameter. The iterations will stop once the pre-set stopping criteria (such as, $\Delta J(w_0, w_1) < \epsilon$) are met.

(Optional) Code example

For a linear regression code example see this python script.

Multivariate linear regression

As opposed to the previous example of predicting the temperature based on altitude, which had only one feature, x (the altitude), real-world problems typically involve multiple features $(x_1, x_2, \cdots, x_n)$ to represent an entity (such as, the altitude, time of the day, season, weather conditions, topography, and so on). Our linear model will then be

$$\widehat{y} = h(\vec{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w_0 + \sum_{i=1}^{n} w_i x_i = w_0 + \vec{w} \cdot \vec{x}$$
$$\vec{w} = [w_1, w_2, \cdots, w_n]^T, \quad \vec{x} = [x_1, x_2, \cdots, x_n]^T$$

$\widehat{y}$ now is a high-dimensional line (planes, hyperplanes). The gradient descent update equations for the multivariate case will be

$$w_0 := w_0 - \alpha \frac{\partial}{\partial w_0} J(w_0, \vec{w}) = w_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right)$$
$$\vec{w} := \vec{w} - \alpha \frac{\partial}{\partial \vec{w}} J(w_0, \vec{w}) = \vec{w} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right) \vec{x}^{(i)}$$

Alternatively, the problem can be solved using the Normal equation, avoiding iterative gradient descent:

$$\begin{bmatrix} w_0 \\ \vec{w} \end{bmatrix} = (X^T X)^{-1} X^T \vec{y}; \quad \vec{w} = [w_1, \cdots, w_n]^T, \quad \vec{y} = [y^{(1)}, \cdots, y^{(m)}]^T, X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}$$

However, matrix multiplication and inversion becomes computationally expensive for large datasets (many data points: large m) and high-dimensionality in data (many features: large n), making gradient descent more scalable.

[1] Initialization is important: depending on how it's initialized, you might get stuck in local minima, if they exist. For now, let's keep things simple.

Inference (or testing)

Once we found the optimum model parameters $(w_0^*, \vec{w}^*)$ from training, we essentially found the optimum model $\widehat{y} = h(x) = w_0^* + \vec{w}^* \cdot \vec{x}$. Now, given a test data sample $(\vec{x}_{test}, y_{test})$, we can predict the target as

$$\widehat{y}_{test} = h(\vec{x}_{test}) = w_0^* + \vec{w}^* \cdot \vec{x}_{test}$$

Note that we did not use y_{test} in the prediction phase. In real scenario y_{test} is generally absent. While testing, we can compare y_{test} (actual target) with $\widehat{y}_{test}$ (predicted target) to evaluate the model's performance.

Polynomial regression

Polynomial regression. Model as a polynomial instead of a linear function.

$$\widehat{y} = h(x) = w_0 + w_1 x + w_2 x^2 + w_3 \sqrt{x} + w_4 x^3 + \cdots$$

Feature scaling

If the range of values of different features $x_1, x_2, \cdots$ are too different, then the model parameters $w_1, w_2, \cdots$ will also be different in range. It will create problems in gradient descent convergence. Gradient descent will bounce for a long time before getting to a solution. Feature scaling removes this large differences in the ranges of features.

$$\text{mean normalization: } x := \frac{x - mean}{max - min}; \quad z - \text{score normalization: } x := \frac{x - mean}{std}$$

Feature engineering

Design new features by transforming or combining original features.

Classification

Classification is also a supervised machine learning problem. The main difference here from regression is that the target axis (variable the model is trying to predict) is now discrete instead of continuous. Let's go back to our mountain trekking example:

> On your planned trip to the mountain peak, you know water is essential. But there's a problem – you can't afford for your water to freeze. If the temperature drops below zero, the water will turn to ice, leaving you stuck without anything to drink. You start wondering: at the altitude you're aiming for, will the temperature stay above freezing, or will it go negative?

Alt (km)	Temp ($^\circ C$)	Will freeze?
1	15	No ($y = 0$)
2	7	No ($y = 0$)
3	3	No ($y = 0$)
4	-6	Yes ($y = 1$)
5	-10	Yes ($y = 1$)
6	-20	Yes ($y = 1$)

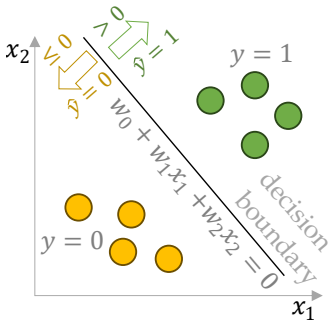
Notice that your target variable (what you are trying to predict) has become discrete.

Modeling

For univariate linear regression case, our prediction model was $w_0 + w_1 x$. Now we are trying to see if this $w_0 + w_1 x$ is greater than or leass than a threshold (zero in this case). Therefore, the linear classification model will be

$$\widehat{y} = \begin{cases} 1, & \text{if } w_0 + w_1 x > 0 \\ 0, & \text{otherwise} \end{cases} \text{ or, for multiple variable case, } \widehat{y} = \begin{cases} 1, & \text{if } w_0 + \vec{w} \cdot \vec{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Note that the target or prediction both have 2 levels ($y, \widehat{y} \in \{0, 1\}$). This is a binary classification problem. For multiple classes, the target would be discretized into multiple levels. Also note that the feature space here is divided into 2 half-spaces (i. $w_0 + \vec{w} \cdot \vec{x} > 0$ and ii. $w_0 + \vec{w} \cdot \vec{x} \leq 0$) with the hyperplane $w_0 + \vec{w} \cdot \vec{x} = 0$. This hyperplane has a dimensionality one less than the ambient space, and this is widely known as the classification decision boundary. Also note that the decision boundary here is linear in $\vec{x}$. In other examples, we might need nonlinear decision boundaries.



Logistic regression

Though the name has regression in it, it is essentially a binary classification model. It takes the piecewise equation for $\widehat{y}$ above and makes it a single-rule function with the help of a sigmoid function (also called logistic function) as follows:

$$\widehat{y} = h(\vec{x}) = g(w_0 + \vec{w} \cdot \vec{x});\ g(z) = \frac{1}{1 + e^{-z}} \text{ (sigmoid)}$$

Here, $\widehat{y} \notin \{0, 1\}$ only when $z \to 0$. Otherwise, $\widehat{y} \in \{0, 1\}$ for every other $z \in \mathbb{R}$. We can ignore the limiting case for now. Or, if we want to be more strict,

$$\widehat{y} = \begin{cases} 1, & \text{if } h(\vec{x}) > 0.5 \\ 0, & \text{otherwise} \end{cases},\ h(\vec{x}) = P(Y = 1 | \vec{X} = \vec{x}; w_0, \vec{w}) = g(w_0 + \vec{w} \cdot \vec{x}),\ g(z) = \frac{1}{1 + e^{-z}}$$

Here strictly, $\widehat{y} \in \{0, 1\}$, i.e., a binary classifier. The equation above basically boils down to the classification model mentioned before

$$\widehat{y} = \begin{cases} 1, & \text{if } w_0 + \vec{w} \cdot \vec{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

If we now want a nonlinear classifier (nonlinear decision boundary), we can do that by introducing polynomials in the inequalities (i.e., the decision boundary) as follows:

$$\widehat{y} = \begin{cases} 1, & \text{if } w_0 + w_1 x + w_2 x^2 + w_3 \sqrt{x} + \cdots > 0 \\ 0, & \text{otherwise} \end{cases}$$

Learning (training)

Let $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$ denote a given set of m training samples. The superscripts indicate the sample number in the dataset. Using these, we have to find w_0 and w_1. Our goal here is to match y with $\widehat{y}$ given the observed data points $(x^{(i)}, y^{(i)})$ and the predictions $(x^{(i)}, \widehat{y}^{(i)})$ by calculating the right values for the model parameters (w_0 and $\vec{w}$). Here, we will also design an objective function $J : \mathbb{R}^{n+1} \to [0, \infty)$ such that it calculates the mismatch cost between the observed data points and the predictions by the linear classification model:

$$J(w_0, \vec{w}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h(\vec{x}^{(i)}) + (1 - y^{(i)}) \log(1 - h(\vec{x}^{(i)})) \right)$$

Say for example, $m = 1$ (which gets rid of all superscripts). If $y = 1$, $J(w_0, \vec{w}) = -\log h(\vec{x})$. When $\widehat{y} \to 1$, $h(\vec{x}) \to 1$, $J(w_0, \vec{w}) \to 0$ ($y, \widehat{y}$ matching case) and when $\widehat{y} \to 0$, $h(\vec{x}) \to 0$, $J(w_0, \vec{w}) \to \infty$ ($y, \widehat{y}$ mismatch case). Similarly, if $y = 0$, $J(w_0, \vec{w}) = -\log(1 - h(\vec{x}))$. When $\widehat{y} \to 0$, $h(\vec{x}) \to 0$, $J(w_0, \vec{w}) \to 0$ ($y, \widehat{y}$ matching case) and when $\widehat{y} \to 1$, $h(\vec{x}) \to 1$, $J(w_0, \vec{w}) \to \infty$ ($y, \widehat{y}$ mismatch case). *Therefore, when y matches $\widehat{y}$, there is a low cost ($J(w_0, \vec{w}) \to 0$), and when there is a mismatch, there is a high cost ($J(w_0, \vec{w}) \to \infty$).* Therefore, to find the right values of w_0 and $\vec{w}$ that pushes for the matching of y and $\widehat{y}$, we need to minimize $J(w_0, w_1)$ with respect to $w_0, \vec{w}$, which gives our optimization problem:

$$\min_{w_0, \vec{w}} J(w_0, \vec{w})$$

How to solve this optimization problem (i.e., find $\arg\min_{w_0, \vec{w}} J(w_0, \vec{w})$)? One option: the gradient descent algorithm:

$$w_0 := w_0 - \alpha \frac{\partial}{\partial w_0} J(w_0, \vec{w}) = w_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h(\vec{x}^{(i)}) - y^{(i)} \right)$$

$$\vec{w} := \vec{w} - \alpha \frac{\partial}{\partial \vec{w}} J(w_0, \vec{w}) = \vec{w} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h(\vec{x}^{(i)}) - y^{(i)} \right) \vec{x}^{(i)}$$

Inference (or testing)

Once we found the optimum model parameters $(w_0^*, \vec{w}^*)$ from training, we essentially found the optimum model. Now, given a test data sample $(\vec{x}_{test}, y_{test})$, we can predict the target as

$$\widehat{y}_{test} = \begin{cases} 1, & \text{if } w_0^* + \vec{w}^* \cdot \vec{x}_{test} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Similar to regression, we did not use y_{test} in the prediction phase. While testing, we can compare y_{test} (actual target) with $\widehat{y}_{test}$ (predicted target) to evaluate the model's performance.

Multiclass classification

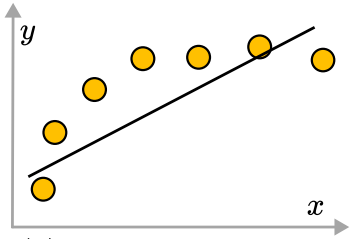
What if now the target variable is discretized into more than 2 levels? i.e., for C levels (or classes) $y, \widehat{y} \in \{0, 1, \cdots, C-1\}$. One option: the *one-vs-all* method. It essentially splits a multiclass classification problem (C classes) into C binary classification problems.

Underfitting and Overfitting

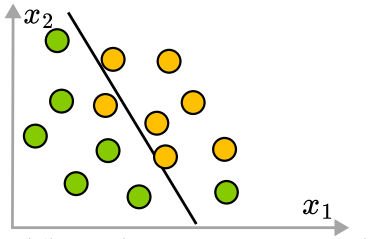
Depending on the complexity of $h(\vec{x})$, we might encounter the following situations:

- **Underfitting (High Bias):** A simple model that does not capture the data patterns well.
- **Overfitting (High Variance):** A complex model that over rectify itself to obsessively match the data.
- **Good Fit:** A model that balances bias and variance effectively.

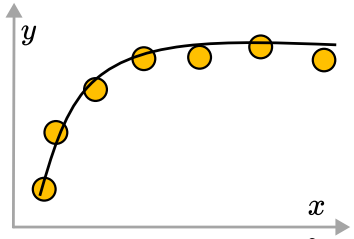
Underfitting
(high bias)



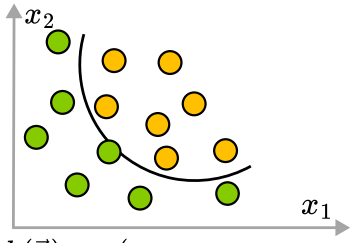
$h(x) = w_0 + w_1 x$



$h(\vec{x}) = g(w_0 + w_1 x_1 + w_2 x_2)$

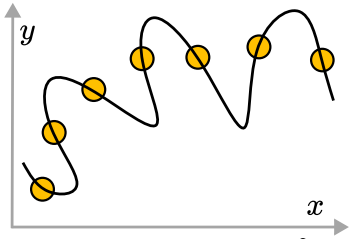


$h(x) = w_0 + w_1 x + w_2 x^2$

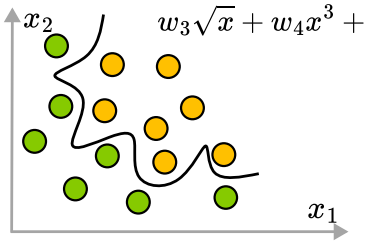


$h(\vec{x}) = g(w_0 + w_1 x_1 + w_2 x_2 + w_2 x_1^2 + w_4 x_2^2)$

Overfitting
(high variance)



$h(x) = w_0 + w_1 x + w_2 x^2 + w_3 \sqrt{x} + w_4 x^3 + \cdots$



$h(\vec{x}) = g(w_0 + w_1 x_1 + w_2 x_2 + w_2 x_1^2 + w_4 x_2^2 + w_5 x_1 x_2 + w_6 \sqrt{x_1} + \cdots)$

Addressing Overfitting

To address overfitting, we can:

1. **Reduce the number of features:** Perform feature selection or model selection. However, this may result in loss of information.
2. **Regularization:** Retain all features but reduce the magnitude of the model parameters w_j. This approach works well when there are many features.

Regularization (linear regression)

Consider a simple liner regression example. With regularization, we modify the cost function to penalize large coefficients. Thus we can have a regularized linear regression

$$J(w_0, \vec{w}) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 + \lambda ||\vec{w}||_{L_2}^2 + \lambda w_0^2\right) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} w_j^2 + \lambda w_0^2\right)$$

where λ is a regularization parameter. Large coefficients contribute more to the cost, and minimizing the cost reduces their magnitude.

Gradient descent update

How would be the optimization? How would be the gradient descent update equation now that you have a new penalized cost function due to regularization? With a little work, it is possible to show that, the gradient descent update equations will be as follows:

$$\vec{w} := \vec{w} - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right) \vec{x}^{(i)} + \frac{\lambda}{m} \vec{w} \right]; \;\; w_0 := w_0 - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y}^{(i)} - y^{(i)} \right) + \frac{\lambda}{m} w_0 \right]$$

Alternatively, the problem can be solved using the Normal equation, avoiding iterative gradient descent:

$$\begin{bmatrix} w_0 \\ \vec{w} \end{bmatrix} = (X^T X + \lambda I)^{-1} X^T \vec{y}; \;\; \vec{w} = [w_1, \cdots, w_n]^T, \;\; \vec{y} = [y^{(1)}, \cdots, y^{(m)}]^T, X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}$$

Regularization (logistic regression)

Similar to above, with regularization, we modify the cost function to penalize large coefficients. Thus we can have a regularized logistic regression

$$J(w_0, \vec{w}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h(\vec{x}^{(i)}) + (1 - y^{(i)}) \log(1 - h(\vec{x}^{(i)})) \right) + \frac{\lambda}{2m} ||\vec{w}||_{L_2}^2 + \frac{\lambda}{2m} w_0^2$$

Gradient descent update

With a little derivation, it is possible to show that, the gradient descent update equations will be as follows:

$$\vec{w} := \vec{w} - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h(\vec{x}^{(i)}) - y^{(i)} \right) \vec{x}^{(i)} + \frac{\lambda}{m} \vec{w} \right]; \;\; w_0 := w_0 - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h(\vec{x}^{(i)}) - y^{(i)} \right) + \frac{\lambda}{m} w_0 \right]$$

Looks similar to the linear regression update, but not the same (as $h(\cdot)$ is different).

Now consider the scenario where you add more polynomial terms in Logistic Regression. Then the logistic regression hypothesis will become:

$$h(\vec{x}) = g(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + \cdots).$$

Here, the number of terms grows rapidly with the number of features. With new polynomial terms, we are essentially adding new features. All of these can increase the risk of overfitting.

Neural Networks

Though very oversimplified, some can call logistic regression the simplest neural network that consists of a single logistic unit:

$$h(\vec{x}) = g(\vec{w} \cdot \vec{x}).$$

where $\vec{w} \cdot \vec{x}$ is the linear term, $g(\cdot)$ is the nonlinear term (also called activation function). Complex (real / actual) neural networks are built with different combinations, modifications, additions (layers), etc. of this simple unit. This simple unit has two layers: input layer (Layer-1; consisting of $\vec{x} = [x_1, x_2, \cdots]^T$) and the output layer (rightmost layer; which is $g(\vec{w} \cdot \vec{x})$).

For more complex problems, neural networks can include multiple layers: **Input Layer:** Represents the input features. **Hidden Layers:** Consist of units that compute activations. **Output Layer:** Produces the final predictions. Let us now add one hidden layer (Layer-2) in between these two layers:

$$\begin{array}{lll} Layer-1: & Layer-2: & Layer-3: \\ \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix} & \vec{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} g(\vec{w}_1^{(1)} \cdot \vec{x}) \\ g(\vec{w}_2^{(1)} \cdot \vec{x}) \\ \vdots \end{bmatrix} & h(\vec{x}) = g(\vec{w}^{(2)} \cdot \vec{a}^{(2)}) \end{array}$$

Similarly, what if we add one more hidden layer (Layer-3) in between, and add one more node in the output layer:

$$\begin{array}{llll} Layer-1: & Layer-2: & Layer-3: & Layer-4: \\ \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix} & \vec{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} g(\vec{w}_1^{(1)} \cdot \vec{x}) \\ g(\vec{w}_2^{(1)} \cdot \vec{x}) \\ \vdots \end{bmatrix} & \vec{a}^{(3)} = \begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \\ \vdots \end{bmatrix} = \begin{bmatrix} g(\vec{w}_1^{(2)} \cdot \vec{x}) \\ g(\vec{w}_2^{(2)} \cdot \vec{x}) \\ \vdots \end{bmatrix} & h(\vec{x}) = \begin{bmatrix} g(\vec{w}_1^{(3)} \cdot \vec{a}^{(3)}) \\ g(\vec{w}_2^{(3)} \cdot \vec{a}^{(3)}) \end{bmatrix} \end{array}$$

Real world neural networks are more complex with more other components, but this is the general idea. The way a network is designed is called architecture.

Now say, how to use a neural network in multiclass classification? Use 1-vs-all method? For K classes, we can have K output nodes in the output layer - one node per class. The output nodes generate a probability-like value (between 0 to 1). The class with the node with the highest value is selected as prediction, in the inference phase. For example, in a 4-class classification problem (e.g., bus, truck, train, car), the output layer will have 4 units.

Binary classification	Multiclass classification (K classes)
$y = 0$ *or* 1	$y \in \mathbb{R}^K$, e.g., $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ for class 2.
1 output unit	K output units

If the goal is to do a 4-class classification, each unit of the output layer (last layer) is dedicated to 1 separate class and each performs one-vs-all classification. For example, let's say we want to do a 4 cass classification (bus truck,train,car). So the output layer will have 4 units. Unit 1 will be for bus, Unit 2 will be for truck, Unit 3 for train, and Unit 4 for car.

What this means is, in Unit 1, the bus will be class 1 ($y = 1$) and all of the truck, train, and car will be class 0 ($y = 0$). Similarly, in Unit 2, truck$\rightarrow y = 1$ and bus, train, car$\rightarrow y = 0$., and so on upto Unit 4.

Now recall each unit of the output unit is a logistic unit (hidden layers can have other activations; this layer can have other activations too). Note that each unit always outputs values between 0 and 1. This means Unit 1 will have a probability like value ($0 \rightarrow 1$) for bus. If $g(\cdot) = 1$, we call it a bus, if $g(\cdot) = 0$, we call it not a bus, if $g(\cdot) = 0.8$, we call it a bus with 80% chance. The same is in Unit 2 for truck, Unit 3 for train, and Unit 4 for car. Finally, we get a $\mathbb{R}^K$ vector. $[0.2, 0, 0, 0.8] \in \mathbb{R}^4$ means Bus with 20% chance and Car with 80% chance, and so on.

Cost function

For a Neural Network, the cost function will be a generalization of the logistic regression cost function

$$J(w) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{u=1}^{U} y_u^{(i)} \log(h_{w_u}(\vec{x}^{(i)})) + (1 - y_u^{(i)}) \log(1 - h_{w_u}(\vec{x}^{(i)}))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(w_{ij}^{(l)})^2,$$

where, L is the total number of layers, s_l is the number of units in the l-th layer.

Now how to optimize this cost function? ($\min_w J(w) = how?$). We need $(\partial/\partial w_{ij}^{(l)})J(w)$. To compute the derivative of $J(w)$ here, we need an algorithm called the backpropagation algorithm.

The main idea: for the outermost layer L, we have the formula for residue $\vec{\delta}^{(L)} = \vec{a}^{(L)} - \vec{y}$. There are specific formula to compute residues of the previous layers, i.e., $\vec{\delta}^{(L-1)}, \vec{\delta}^{(L-2)}, \cdots, \vec{\delta}^{(1)}$. Next we weighted sum these δ terms and then the sum over all training data $\rightarrow$ this sum gives the partial derivative of $J(w)$ with respect to w.Then we can use this in gradient descent (or other optimization algorithms).

Evaluate a hypothesis

Split datasets into training testing validation sets. Use the training data to find the model parameters, the validation data to tune the hyperparamaeters, and the test data to finally evaluate the model.

To evaluate a hypothesis, you can compute the cost function on the test set:

$$J_{\text{test}}(w) = \frac{1}{2m_{\text{test}}}\sum_{i=1}^{m_{\text{test}}}\left(h_w(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)}\right)^2$$

This is the squared error cost. For other problems, we use different cost functions. For example, in logistic regression:

$$J_{\text{test}}(w) = -\frac{1}{m_{\text{test}}}\sum_{i=1}^{m_{\text{test}}}\left(y_{\text{test}}^{(i)} \log h_w(x_{\text{test}}^{(i)}) + (1 - y_{\text{test}}^{(i)}) \log(1 - h_w(x_{\text{test}}^{(i)}))\right)$$

For classification problems, the misclassification error can be calculated, which can be defined as:

$$\text{error}(h_w(x), y) = \begin{cases} 1 & \text{if } h_w(\vec{x}) \geq 0.5 \text{ and } y = 0, \text{ or if } h_w(\vec{x}) < 0.5 \text{ and } y = 1 \\ 0 & \text{otherwise.} \end{cases}$$

$$J_{test}(w) = \frac{1}{m_{test}}\sum_{i=1}^{m_{test}} \text{error}(h_w(\vec{x}_{test}^{(i)}), y_{test}^{(i)})$$

Model Selection

For example, suppose you have the following models:

$$\begin{aligned} h_w(x) &= w_0 + w_1 x && (d = 1) \\ h_w(x) &= w_0 + w_1 x + w_2 x^2 && (d = 2) \\ h_w(x) &= w_0 + w_1 x + w_2 x^2 + w_3 x^3 && (d = 3) \\ &\vdots \\ h_w(x) &= \sum_{i=0}^{10} w_i x^i && (d = 10) \end{aligned}$$

Say, you want to choose among the models above. How do you choose d?

Using Training ad Test Set

You can use your training data to train all choices of these models. This gives you parameters $\vec{w}_{\text{model1}}$, $\vec{w}_{\text{model2}}$, ..., $\vec{w}_{\text{model10}}$. Then you can measure the test error (cost) for all models:

$$J_{\text{test}}(\vec{w}_{\text{model1}}), J_{\text{test}}(\vec{w}_{\text{model2}}), \ldots, J_{\text{test}}(\vec{w}_{\text{model10}})$$

Then, you can select the model with the lowest J_{test}. **Is this a good idea? No!** Why? You have used training data to fit w, which is fine. However, using test data to fit the extra parameter d (another model parameter) is not correct.

Introducing the Validation Set

To resolve this, split the dataset into three parts:

$$\text{Training Set} \quad (\vec{x}_{\text{train}}, \vec{y}_{\text{train}})$$

$$\text{Validation Set} \quad (\vec{x}_{\text{val}}, \vec{y}_{\text{val}})$$

$$\text{Test Set} \quad (\vec{x}_{\text{test}}, \vec{y}_{\text{test}})$$

Now, define the validation cost:

$$J_{\text{val}}(w) = \text{cost on validation set.}$$

Use $(\vec{x}_{\text{val}}, \vec{y}_{\text{val}})$ to select the model (tune d).

Hyperparameter Tuning and Regularization

The parameter d is called a **hyperparameter**. Other hyperparameters include the regularization parameter λ. Use the validation set to select all hyperparameters.

Diagnosing Bias vs. Variance

For this part follow the discussion in the hand-written lecture. The plot in the hand written lecture can help diagnose bias vs. variance problems:

- High J_{val}, High J_{train}: High bias (underfit).
- High J_{val}, Low J_{train}: High variance (overfit).

For example, for the regularization parameter λ:

- Large λ: High bias.
- Small λ: High variance.

Optimize (tune) λ using the validation set. For regularization, the cost function is:

$$J(w) = J_1(w) + \lambda J_2(w)$$

where:

$$J_1(w) = \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log h_w(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_w(x^{(i)})) \right]$$

and:

$$J_2(w) = \frac{1}{2m} \sum_{j=1}^{n} w_j^2$$

Regularization and Bias-Variance Tradeoff

When λ is high, $J_1(w)$ is negligible in comparison to $J_2(w)$:

$$J(w) \approx \lambda J_2(w) \Rightarrow \min J(w) \approx \min J_2(w)$$

When λ is low, $J_2(w)$ is negligible in comparison to $J_1(w)$:

$$J(w) \approx J_1(w) \Rightarrow \min J(w) \approx \min J_1(w)$$

Now,

$$\sum_{j=1}^{n} w_j^2 = \|w\|_{L_2}^2, \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

Minimizing the L_2 norm makes the magnitudes of w_j small, making the model less sensitive to changes in feature values.

Learning Curve Analysis

For this part follow the discussion in the hand-written lecture. For sanity check, improving performance, diagnosing bias, variance, or both:

- Small training set $\Rightarrow$ easy to fit $\Rightarrow$ small J_{train}
- Increasing training set size helps identify bias and variance issues.

High Bias

- The curve will be flat, meaning adding more training data does not help.
- The gap between J_{train} and J_{val} is small.
- The height of both curves is high.
- Stop collecting more training data, as it won't help.

High Variance

- The gap between J_{train} and J_{val} will be large.
- The height of J_{train} will be low, while J_{val} will be high.

Handling Bias and Variance

High Variance

- Get more training data.
- Try a smaller set of features (feature selection/combination).
- Try increasing λ (regularization parameter).

High Bias

- Get more features.
- Create artificial features (e.g., use polynomials).
- Try decreasing λ.

Neural Networks and Overfitting

- Small neural networks have fewer parameters, making them prone to underfitting but computationally cheaper.
- Large neural networks have more parameters, making them prone to overfitting and computationally expensive (requiring regularization).

The number of hidden layers in a neural network is a hyperparameter, which should be tuned using a validation set. **Obtaining relevant features is very important and requires heuristic knowledge.**

Error analysis

Accuracy can be misleading

$$Accuracy = \frac{\#\ correctly\ classified}{total\ \#} \times 100\%$$

For skewed classes #-samples in one class is much higher (or lower) than the other class. In this case accuracy is not a good metric. Before that, let's understand the concept of a confusion matrix
Confusion matrix:

	$y=1$	$y=0$
$\widehat{y}=1$	TP	FP
$\widehat{y}=0$	FN	TN

TP=true +ve, TN=true -ve FP=false +ve, FN=false -ve.

Using the confusion matrix, we can calculate the following metrics:

$$acc = \frac{TP+TN}{TP+TN+FP+FN},\ prec = \frac{TP}{TP+FP},\ rec = \frac{TP}{TP+FN},\ F_1 = 2\frac{prec \times rec}{prec+rec}.$$

Now, consider classification problem where we are trying to classify between a number of buses ($y=1$) and cars ($y=0$). Now, also assume we have completely failed at that; we build a very bad classification model. It's so bad that it always predicts car ($\widehat{y}=0$) no matter what.

Now further assume that you have a test (or validation) set of 100 images, out of which 99 of them are images of a car, and only 1 of them is an image of a bus. What will be the accuracy of your bad model here? Let's start with the confusion matrix for this problem:

	$y=1$	$y=0$
$\widehat{y}=1$	TP	FP
$\widehat{y}=0$	FN	TN

$\rightarrow$

	$y=1$	$y=0$
$\widehat{y}=1$	0	0
$\widehat{y}=0$	1	99

$$\implies acc = \frac{TP+TN}{TP+TN+FP+FN} = 99\%$$

Accuracy is 99% (!!). Does that mean you classifier is good? NO! It still is a bad classifier. You see how accuracy can mislead you in model evaluation.

Now, let's calculate another metric, recall:

$$rec = \frac{TP}{TP+FN} = \frac{0}{0+1} = 0\%$$

The precision is very low, which means the model is not actually a good model.

Let us examine the confusion matrix of a hypothetical good classifier:

	$y=1$	$y=0$
$\widehat{y}=1$	35	0
$\widehat{y}=0$	0	65

$Here,\ acc = 100\%,\ prec = 100\%,\ rec = 100\%,\ F_1 = 100\%.$

For a actually good classifier, all these metrics will be very close to 100%.

Support Vector Machines (SVM)

Recap logistic regression again:

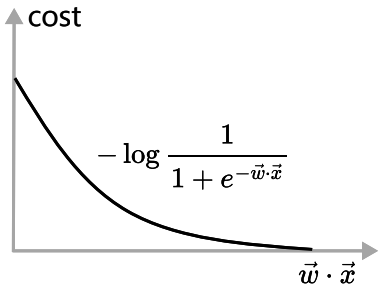
$$h(\vec{x}) = \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}}$$

If $y = 1$, we want $h(\vec{x}) \simeq 1$, i.e., $\vec{w}\cdot\vec{x} >> 0$ and if $y = 0$, we want $h(\vec{x}) \simeq 0$, i.e., $\vec{w}\cdot\vec{x} << 0$.

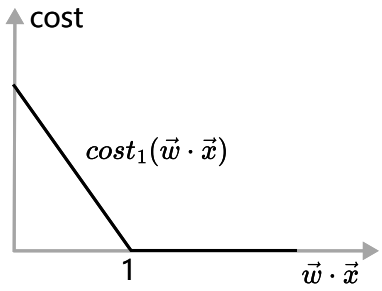
Cost for 1 training example is

$$-y \log \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}} - (1-y)\log(1 - \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}})$$

Log-Regression



SVM

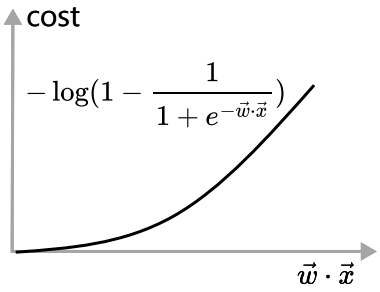


If $y = 1$, $cost = -\log(1/(1+\exp(-\vec{w}\cdot\vec{x})))$, we want $\vec{w}\cdot\vec{x} >> 0$. For SVM, we will modify this cost function profile a bit to $cost_1(\vec{w}\cdot\vec{x})$.

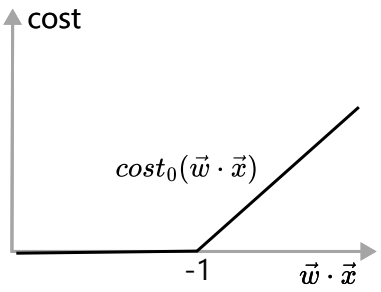
$$cost_1(\vec{w}\cdot\vec{x}) = \begin{cases} -m(\vec{w}\cdot\vec{x}) + m & \text{if } \vec{w}\cdot\vec{x} < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, if $y = 0$, $cost = -\log(1 - 1/(1+\exp(-\vec{w}\cdot\vec{x})))$, we want $\vec{w}\cdot\vec{x} << 0$. For SVM, we will modify this cost function profile a bit to $cost_0(\vec{w}\cdot\vec{x})$.

Log-Regression



SVM



$$cost_0(\vec{w}\cdot\vec{x}) = \begin{cases} m(\vec{w}\cdot\vec{x}) + m & \text{if } \vec{w}\cdot\vec{x} > -1 \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, the SVM cost function

$$J(\vec{w}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} cost_1(\vec{w}\cdot\vec{x}^{(i)}) + (1-y^{(i)}) cost_0(\vec{w}\cdot\vec{x}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

For SVM, we will do a little more notational change. Drop $1/m$ as it will not change the solution with m being a constant multiplier. Then instead of $J_1 + \lambda J_2$, the cost will have the form $CJ_1 + J_2$. Clearly,

$$J(\vec{w}) = C\left[\sum_{i=1}^{m} y^{(i)} cost_1(\vec{w}\cdot\vec{x}^{(i)}) + (1-y^{(i)}) cost_0(\vec{w}\cdot\vec{x}^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n} w_j^2$$

SVM hypothesis

$$h(\vec{x}) = (?)\ \widehat{y} = \begin{cases} 1, & \text{if } \vec{w} \cdot \vec{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

SVM is also known as a large margin classifier.

Large margin intuition

To make the overall SVM cost small, what do we need?

If $y = 1$, we need $\vec{w} \cdot \vec{x} \geq 1$ (not just ≥ 0); and if $y = 0$, we need $\vec{w} \cdot \vec{x} \leq -1$ (not just ≤ 0). Extra safety margin in SVM.

Alternate view of the SVM optimization problem. We can turn the uncostrained optimization above into a costrained optimization as follows:

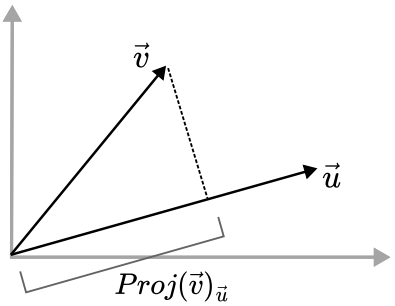
$$\begin{aligned} &\min_{\vec{w}} \frac{1}{2} \sum_{j=1}^{n} w_j^2 \\ &s.t.\ \vec{w} \cdot \vec{x}^{(i)} \geq 1,\ if\ y^{(i)} = 1, \\ &\quad \vec{w} \cdot \vec{x}^{(i)} \leq -1,\ if\ y^{(i)} = 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} &\min_{\vec{w}} \frac{1}{2} ||\vec{w}||_{L_2}^2 \\ &s.t.\ |\vec{w} \cdot \vec{x}^{(i)}| \geq 1 \end{aligned}$$

Linear algebra recap

Let us consider two vectors $\vec{u}$ and $\vec{v}$. $||\vec{u}||_{L_2}$ denotes the L_2 norm of $\vec{u}$ and $Proj(\vec{v})_{\vec{u}}$ denotes the projection of $\vec{v}$ on $\vec{u}$.

If $\vec{u} = [u_1, u_2]^T$, $\vec{v} = [v_1, v_2]^T$, then $||\vec{u}||_{L_2}^2 = \sqrt{u_1^2 + u_2^2}$, also

$$\begin{aligned} \vec{u} \cdot \vec{v} &= \vec{u}^T \vec{v} = u_1 v_1 + u_2 v_2 = ||\vec{u}||_{L_2} ||\vec{v}||_{L_2} \cos\theta \\ &= ||\vec{u}||_{L_2} Proj(\vec{v})_{\vec{u}} \end{aligned}$$



Large margin again

Therefore, the optimization problem above can be re-written as

$$\begin{aligned} &\min_{\vec{w}} \frac{1}{2} \sum_{j=1}^{n} w_j^2 \\ &s.t.\ ||\vec{w}||_{L_2} Proj(\vec{x}^{(i)})_{\vec{w}} \geq 1,\ if\ y^{(i)} = 1, \\ &\quad ||\vec{w}||_{L_2} Proj(\vec{x}^{(i)})_{\vec{w}} \leq -1,\ if\ y^{(i)} = 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} &\min_{\vec{w}} \frac{1}{2} ||\vec{w}||_{L_2}^2 \\ &s.t.\ \left| ||\vec{w}||_{L_2} Proj(\vec{x}^{(i)})_{\vec{w}} \right| \geq 1 \end{aligned}$$

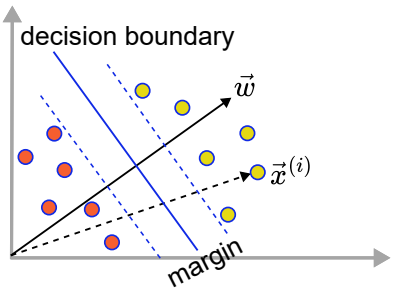
In other words

$$\min_{\vec{w}} \frac{1}{2}||\vec{w}||^2_{L_2}$$

$$s.t. \quad \left|Proj(\vec{x}^{(i)})_{\vec{w}}\right| \geq \frac{1}{||\vec{w}||_{L_2}} \text{ (a +ve number, also big - as minimizing w's L-2 norm)}$$

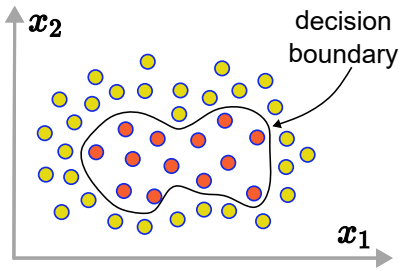
Therefore, $|Proj(\vec{x}^{(i)})_{\vec{w}}|$ has to be big. This condition naturally makes the margin larger. Note that $\vec{w}$ is always perpendicular to the decision boundary.

$|Proj(\vec{x}^{(i)})_{\vec{w}}|$ large means $\vec{x}^{(i)}$ aligns mostly along the direction of $\vec{w}$ and perpendicular to the decision boundary. This makes the margin larger. On the contrary, $|Proj(\vec{x}^{(i)})_{\vec{w}}|$ small means $\vec{x}^{(i)}$ aligns mostly perpendicular to the direction of $\vec{w}$, in other words along the decision boundary. This makes it difficult for the margin to be larger.



Kernels

A kernel in machine learning is a function that transforms data into another space where it becomes easier to model. This is useful when data is not easy to model in its original form. For example, in any linear classifier, a kernel maps the data into another dimensional space where a clear boundary can be drawn. The RBF (Radial Basis Function) kernel, for instance, transforms data based on distance, making complex patterns separable without explicitly computing the new dimensions.



Consider this nonlinear classification problem. It's a two class classification problem, with data being very complex and nonlinear. The feature space contains only two features: x_1 and x_2. You can not solve it with a linear classifier. What will you do then? How will you obtain this non linear decision boundary? One option: replace $\vec{w} \cdot \vec{x}$ with a complex non linear polynomial (like we did before). In other words, the hypothesis will become

$$h(\vec{x}) = \begin{cases} 1, & \text{if } w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2 + \cdots \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Now we know that polynomials have many problems. For one, it blows the dimensionality of the feature space which invites overfitting. Can we think of another alternative? Let us randomly pick two ($j = 2$) points $\vec{l}^{(1)}$ and $\vec{l}^{(2)}$ (let's call them landmarks) from within the red cloud in the figure above such that, the landmarks are closer to the points in the red class and far apart from the points in the yellow class. Now given data points $\vec{x}^{(i)}$ and landmarks $\vec{l}^{(j)}$, let us compute new features $f_j^{(i)}$ that measures the similarity or proximity of the data points $\vec{x}^{(i)}$ to $\vec{l}^{(j)}$. One possible calculation of $f_j^{(i)}$ can be the

following:

$$f_j^{(i)} = \text{similarity}\left(\vec{x}^{(i)} - \vec{l}^{(j)}\right) = \exp\left(-\frac{||\vec{x}^{(i)} - \vec{l}^{(j)}||_{L_2}^2}{2\sigma^2}\right)$$

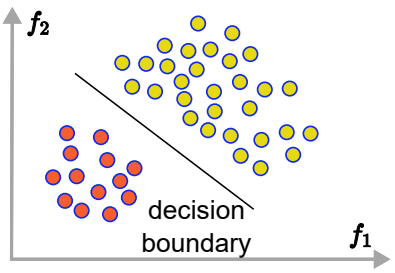
$$\vec{f}^{(i)} = \left[f_1^{(i)}, f_2^{(i)}, \cdots\right]^T; \quad f_0 = \text{DC offset/bias term}$$

The function we are using to measure the similarity is claled the kernel function. In the above example, we are using the Gaussian kernel. We could have chosen other kernel functions too. Now let's see what this kernel does:

$$if\ \vec{x}^{(i)} \simeq \vec{l}^{(j)}, f_j^{(i)} \simeq \exp(-0) = 1\ \text{(a high value)}$$

$$and\ if\ \vec{x}^{(i)} \not\simeq \vec{l}^{(j)}, f_j^{(i)} \simeq \exp(-a\ large\ number) = 1\ \text{(a small value)}$$

Therefore, it is safe to say that all data points in the red class will have smaller $f_j^{(i)}$ values, while those in the yellow class will have larger values. Thus, red class points will cluster at the bottom left and yellow class points at the top right in the transformed f_j-space. The figure on the right shows this phenomenon. We can see that the transformed space has simplified the data to a great extent, now we can use a simple linear decision boundary to separate classes.



The new feature vector in the transformed space

$$\vec{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \end{bmatrix}; \quad \vec{f}^{(i)} = \begin{bmatrix} f_1^{(i)} \\ f_2^{(i)} \\ \vdots \end{bmatrix} \quad for\ the\ i-th\ training\ sample.$$

And finally, the hypothesis or classification model

$$\widehat{y} = h(\vec{x}) = \begin{cases} 1, & \text{if } f_0 + \vec{f} \cdot \vec{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

which is a simple linear model. Now how to choose these landmarks $\vec{l}^{(1)}, \vec{l}^{(2)}, \ldots$? Given training examples $(\vec{x}^{(1)}, y^{(1)})$, $(\vec{x}^{(2)}, y^{(2)})$, ..., $(\vec{x}^{(m)}, y^{(m)})$, choose $\vec{l}^{(1)} = \vec{x}^{(1)}, \vec{l}^{(2)} = \vec{x}^{(2)}, \ldots, \vec{l}^{(m)} = \vec{x}^{(m)}$. i.e., place landmarks $\vec{l}^{(1)}, \vec{l}^{(2)}, \ldots$ at exactly the same location of your training data points. Now given $\vec{x}$, compute

$$f_1 = k(\vec{x}, \vec{l}^{(1)}), \quad f_2 = k(\vec{x}, \vec{l}^{(2)}), \quad \ldots$$

Then new feature vector for $\vec{x}$ ($\vec{x}$ was the old feature vector) is:

$$\vec{f} = \left[f_1, f_2, \cdots\right]^T$$

Everything else later are same. $||\vec{w}||^2$ will be replaced by $\vec{w}^T M \vec{w}$ in regularized ℓ_2, rest kept same.

You can use the kernel idea in other learning algorithms too (e.g., Logistic Regression).

- Large $C \rightarrow$ lower bias, high variance; Small $C \rightarrow$ high bias, low variance.
- Large $\sigma^2 \rightarrow$ features ϕ_i will vary more smoothly, high bias, low variance.

We use a linear kernel (or no kernel) when the number of original features $(x_1, x_2, \ldots, x_n)$ is large and the number of training samples is small. Otherwise, if n is small and m is large $\rightarrow$ we use a nonlinear kernel. If m is too large, create/add more features, then we use logistic regression or SVM without a kernel (linear kernel). Other kernel functions can be used too:

- Polynomial kernel: $K(\vec{x}, \vec{l}) = (\vec{x} \cdot \vec{l} + \text{constant})^{\text{degree}}$
- Other kernels: RBF kernel, string kernel, chi-square kernel, histogram intersection kernel.

SVM is used for binary classification. What about multiclass classification? We use the one-vs-all method.

Unsupervised learning

Given data but not labels $(\vec{x}^{(i)}, y^{(i)} = ?)$. Goal: Find some structure in the data. Clustering: Group data into distinct clusters. We will now see one type of clustering algorithm.

k-Means clustering

Say, the number of clusters K is given. The k-means clustering algorithm has three steps (1. Initialization, 2. Cluster assignment, 3. Move centroid).
Sequence: $1 \rightarrow (2 \rightarrow 3) \rightarrow (2 \rightarrow 3) \rightarrow \cdots$

1. **Initialization**: Randomly initialize K cluster centroids $\vec{\mu}_1, \vec{\mu}_2, \cdots, \vec{\mu}_k \in \mathbb{R}^d$, each corresponding to a cluster.
2. **Cluster assignment**: Go through all m training data points, assign each of them to either of the K centroids based on the distances from the centroids.
   For $i = 1$ to m
   $c^{(i)} :=$ index (1 to K) of cluster centroids closest to $\vec{x}^{(i)}$
   $c^{(i)} = \arg\min_k ||\vec{x}^{(i)} - \vec{\mu}_k||_{L_2};\ k \in \{1, \cdots, K\}$
3. **Move centroid**: Recompute centroids by taking the mean of all points assigned to that centroid.
   For $k = 1$ to K
   $\vec{\mu}_k :=$ average (mean) of points assigned to cluster k
   $\vec{\mu}_k = (1/|\{i|c^{(i)} = k\}|) \sum_{i:c^{(i)}=k} x^{(i)}$

Step 1 will occur only once. Then iteratively repeat steps 2 and 3 until convergence. If $\vec{\mu}_{c^{(i)}}$ denotes the custer centroid to which an example $\vec{x}^{(i)}$ has been assigned, then we can write the optimization problem as follows:

$$\min_{\substack{c^{(1)}, \cdots, c^{(m)} \\ \vec{\mu}_1, \cdots, \vec{\mu}_K}} J(c^{(1)}, \cdots, c^{(m)}, \vec{\mu}_1, \cdots, \vec{\mu}_K) = \frac{1}{m} \sum_{i=1}^{m} ||\vec{x}^{(i)} - \vec{\mu}_{c^{(i)}}||_{L_2}^2$$

Cluster Assignment: Cluster assignment aims to minimize with respect to $c^{(1)}, c^{(2)}, \ldots, c^{(m)}$ while holding $\mu_1, \mu_2, \ldots, \mu_k$ fixed:

$$\min_{c^{(1)},\ldots,c^{(m)}} \sum_{i=1}^{m} \|\vec{x}^{(i)} - \vec{\mu}_{c^{(i)}}\|^2$$

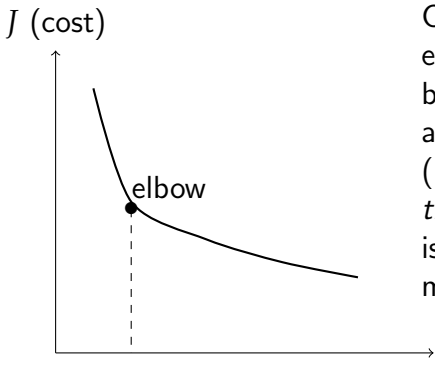
Centroid Update: Move the cluster centroids. Cluster assignment aims to minimize with respect to $\mu_1, \mu_2, \ldots, \mu_k$ while holding $c^{(1)}, c^{(2)}, \ldots, c^{(m)}$ fixed:

$$\min_{\mu_1,\mu_2,\ldots,\mu_k} \sum_{i=1}^{m} \|\vec{x}^{(i)} - \vec{\mu}_{c^{(i)}}\|^2; \qquad \vec{\mu}_k \leftarrow \frac{1}{|C_k|} \sum_{\vec{x}^{(i)} \in C_k} \vec{x}^{(i)}$$

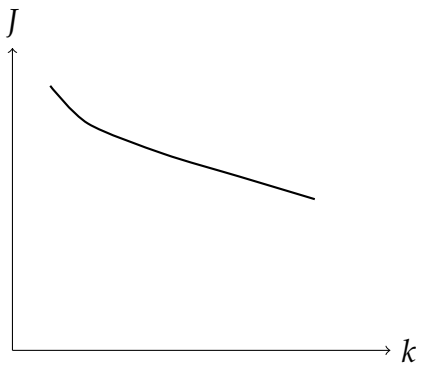
Centroid Initialization: One option is **random initialization**, where centroids are placed at random locations. Another option is to randomly pick k training examples and set the centroids equal to those examples. *Solution can depend on initialization.* To improve results, try different initializations and pick the solution with the **minimum final cost**.

Choosing Number of Clusters (k)

In the k-means clustering algorithm, we began with the assumption that the number of clusters K is given. However, in solving a real problem, we need to estimate K as well, before running the clustering algorithm. One such method to estimate K is Elbow method.



Choose k corresponding to the elbow point. This is found by running the entire k-means algorithm while varying k (number of clusters). *What if there is no clear elbow?* This is one limitation of the elbow method.



Volume 4

Dimensionality reduction

Data compression. Reduce redundancy. Reduce data dimensions. Combine existing features to create new features. More applications: compression, visualization, and so on. Visualization application: say $n = 100$, you cannot plot and visualize data in 100 dimensional space. But you can reduce dimensionalities to $k = 2 or 3$ and plot them and visualize the data. Can be used to address overfitting too. You can also use PCA compression to speed up other learning algorithms (such as supervised learning algorithms). In general, reduce data from $N_1 - D$ to $N_2 - D$ where $N_1 > N_2$.

Principal component analysis (PCA)

Training set=$\{\vec{x}^{(1)}, \vec{x}^{(2)}, \cdots, \vec{x}^{(m)}\}$.

1. Compute mean $\vec{\mu} = (1/m)\sum_{i=1}^{m} \vec{x}^{(i)}$. Replace $\vec{x}^{(i)}$ with $\vec{x}^{(i)} - \vec{\mu}$; $i \in \{1, \cdots m\}$.
2. Compute covariance matrix (size=$n \times n$) $C = (1/m)\sum_{i=1}^{m} (\vec{x}^{(i)})(\vec{x}^{(i)})^T$.
3. Compute eigenvectors of C (also principal vectors or principal directions): $\vec{u}^{(1)}, \vec{u}^{(2)}, \cdots, \vec{u}^{(k)}$. These are the basis vectors of the PCA transformed space.
4. Compute principal components $= \{< \vec{x}^{(i)}, \vec{u}^{(1)} >, < \vec{x}^{(i)}, \vec{u}^{(2)} >, \cdots, < \vec{x}^{(i)}, \vec{u}^{(k)} >\}, i \in \{1, \cdots, m\}$. In matrix formulation,

$$\vec{z}^{(i)} = [\vec{u}^{(1)}, \vec{u}^{(2)}, \cdots, \vec{u}^{(k)}]^T \vec{x}^{(i)}; \;\; i \in \{1, \cdots, m\}$$

5. Reconstruction from PCA space

$$\vec{x}^{(i)}_{approx} = [\vec{u}^{(1)}, \vec{u}^{(2)}, \cdots, \vec{u}^{(k)}]\vec{z}^{(i)}; \;\; i \in \{1, \cdots, m\}$$

Caution: Use the training set to obtain the principal vectors $\vec{u}^{(j)}$, not the validation or test set. Can you explain, why?

PCA objective

$$\min \sum_{i=1}^{m} ||\vec{x}^{(i)} - \vec{x}^{(i)}_{approx}||^2_{L_2}$$

Total variation in data $= (1/m)\sum_{i=1}^{m} ||\vec{x}^{(i)}||^2_{L_2}$. Choose k to be the smallest value so that

$$1 - \frac{\sum_{j=1}^{k} Variance_j}{\sum_{j=1}^{n} Variance_j} = \frac{\sum_{i=1}^{m} ||\vec{x}^{(i)} - \vec{x}^{(i)}_{approx}||^2_{L_2}}{\sum_{i=1}^{m} ||\vec{x}^{(i)}||^2_{L_2}} < \epsilon, \;\; \epsilon = 1\%, 5\%, 10\%, etc.$$

Typically, this threshold is chosen to be 1%, 5%, etc. If the threshold = 1%, it means that at least 99% of variance is retained. You would want to retain high variance. if $k = n$ (i.e., no dimensionality reduction) then $\vec{x}^{(i)} = \vec{x}^{(i)}_{approx}$, and then 100% variance is retained. But you also want $k < n$, which means the retained variance will also be $< 100\%$.

Learning with large datasets

Batch gradient descent

$$\vec{w} := \vec{w} - \alpha \frac{\partial}{\partial \vec{w}} J(\vec{w})$$

Stochastic gradient descent. For 1 of m training examples at a time

$$\vec{w} := \vec{w} - \alpha \frac{\partial}{\partial \vec{w}} J(\vec{w})$$

Repeat for all m training samples.

Mini-batch gradient descent. For $b << m$ of m training examples at a time

$$\vec{w} := \vec{w} - \alpha \frac{\partial}{\partial \vec{w}} J(\vec{w})$$

Repeat for all m training samples.

Online learning

Learning from a continuous stream of data. Suitable when you have an unlimited flow of data. Pseudocode:

Repeat forever:
{
– Collect new data (or mini batch of it) $(\vec{x}, y)$ from the stream.
– Update $\vec{w}$ using new $(\vec{x}, y)$: $\vec{w} := w - \alpha(\partial/\partial\vec{w})J(\vec{w})$.
– Throw away $(\vec{x}, y)$.
}

Map reduce

Say you have 4 machines. Split your batch of data into 4 partitions. Use 4 machines for 4 partitions. 4 machines will give you 4 calculations (such as $\frac{\partial}{\partial w_j} J(\vec{w})$). Then one master machine will combine these calculations from these 4 machines. Thus you can parallelize your computations.

Anomaly detection

Training set $\{\vec{x}^{(1)}, \vec{x}^{(2)}, \cdots\}$, Test data $\vec{x}_{test}$.
Use training data to build model $p(\vec{x})$. Now, if $p(\vec{x}_{test}) < \epsilon$, it's anomalous.

Gaussian distribution. For $x \in \mathbb{R}$, if x is a distributed Gaussian with mean μ. variance σ^2. $x \sim \mathcal{N}(\mu, \sigma^2)$ (meaning x is distributed as a Normal distribution with mean μ and variance σ^2).

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Given $\{x^{(1)}, x^{(2)}, \cdots\}, x^{(i)} \in \mathbb{R}^1$, where we assume $x \sim \mathcal{N}(\mu, \sigma^2)$, parameter estimation means we have to estimate μ and σ. The maximum likelihood estimation in this case is

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$

Algorithm

Say the training set is $\{\vec{x}^{(1)}, \vec{x}^{(2)}, \cdots\}$, $\vec{x}^{(i)} \in \mathbb{R}^n$ and $x_1 \sim \mathcal{N}(\mu_1, \sigma_1^2), x_2 \sim \mathcal{N}(\mu_2, \sigma_2^2), \cdots$, then

$$p(\vec{x}) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)\cdots = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

assuming $x_1, x_2, \cdots, x_n$ are independent. The whole algorithm is as follows:

1. Build feature vector $\vec{x}$ that might be indicative of anomaly.
2. Training: Fit parameters $\mu_1, \cdots, \mu_n, \sigma_1^2, \cdots, \sigma_n^2$.
$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)};\ \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$
3. Testing: Given test data $\vec{x}$, compute $p(\vec{x})$
$$p(\vec{x}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$
4. Anomaly if $p(\vec{x}_{test}) < \epsilon$.

How to evaluate an anomaly detection algorithm? The training set will contain only non-anomalous examples. Cross-validation and a test set will be used, where the labels indicating whether an instance is anomalous or not will be known. The ground truths should be matched with the model predictions to evaluate its performance.

In anomaly detection, the training data is typically unlabeled, whereas the validation and test data are labeled. The distribution of anomalous and non-anomalous examples is usually skewed, meaning that the number of anomalous examples is significantly smaller compared to non-anomalous examples.

To build a feature vector for an anomaly detection algorithm, the appropriate features should be chosen carefully. A histogram should be plotted to analyze the data distribution. If the distribution is not Gaussian, some transformation should be applied to make it closer to Gaussian.

Multivariate Gaussian distribution. $\vec{x} \in \mathbb{R}^n$

Here, we do not model $p(x_1), p(x_2), \cdots$ separately. We model $p(\vec{x})$ jointly. Parameters here are $\vec{\mu} \in \mathbb{R}^n, C \in \mathbb{R}^{n\times n}$ (covariance matrix).

$$p(\vec{x}; \vec{\mu}, C) = \frac{1}{(2\pi)^{(n/2)}|C|^{1/2}} \exp\left(-\frac{1}{2}(\vec{x} - \vec{\mu})^T C^{-1}(\vec{x} - \vec{\mu})\right)$$

Parameter estimation: Given training set $\{\vec{x}^{(1)}, \vec{x}^{(2)}, \cdots, \vec{x}^{(m)}\}, \vec{x} \in \mathbb{R}^n$

$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m} \vec{x}^{(i)}, \quad C = \frac{1}{m}\sum_{i=1}^{m}\left(\vec{x}^{(i)} - \vec{\mu}\right)\left(\vec{x}^{(i)} - \vec{\mu}\right)^T$$

Algorithm

1. Training: Fit parameters $\mu_1, \cdots, \mu_n, \sigma_1^2, \cdots, \sigma_n^2$.

$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m}\vec{x}^{(i)};\ C = \frac{1}{m}\sum_{i=1}^{m}(\vec{x}^{(i)} - \vec{\mu})(\vec{x}^{(i)} - \vec{\mu})^T$$

2. Testing: Given new test example $\vec{x}$, compute $p(\vec{x})$

$$p(\vec{x}) = \frac{1}{(2\pi)^{(n/2)}|C|^{1/2}}\exp\left(-\frac{1}{2}(\vec{x} - \vec{\mu})^T C^{-1}(\vec{x} - \vec{\mu})\right)$$

3. Anomaly if $p(\vec{x}_{test}) < \epsilon$.

The separable model $p(\vec{x}) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)\cdots$ and the multivariate model $p(\vec{x})$ as given above would be same if off diagonal terms in C are zero. The separable model is a special case of the multivariate model.

Recommender System

Netflix, YouTube's video recommendation, Facebook's friend recommendation, etc.

Let us assume, n_u = number of users, n_m = number of objects (movie, video, book, etc.)

$$r(i,j) = 1 \quad \text{if user } j \text{ has rated object } i$$

$$y(i,j) = \text{rating given by user } j \text{ to object } i \quad (\text{defined if } r(i,j) = 1);\ y \in \{1,2,3,4,5\}$$

Problem: Based on this data, recommend a new object to a new user.

Content-based Recommendation

Say the object is a movie (n_m movies). Each movie has two features: drama (x_1), action (x_2). [Each movie is a feature vector $\vec{x}^{(i)}$]
For each user j, learn parameter $\vec{\theta}^{(j)}$
Predict user j's rating for movie i with:

$$(\vec{\theta}^{(j)})^T\vec{x}^{(i)} \quad \text{stars}$$

$$m^{(j)} \rightarrow \text{number of movies rated by user } j$$

To learn $\vec{\theta}^{(j)}$:

$$\min_{\vec{\theta}^{(j)}}\frac{1}{2m^{(j)}}\sum_{i:r(i,j)=1}\left((\vec{\theta}^{(j)})^T\vec{x}^{(i)} - y(i,j)\right)^2 + \frac{\lambda}{2m^{(j)}}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

- Similar to linear regression (least squares)
- You can think with other complex models too

- Can get rid of $m^{(j)}$ as it's a constant

To learn $\vec{\theta}^{(1)}, \vec{\theta}^{(2)}, \ldots, \vec{\theta}^{(n_u)}$:

$$\min_{\vec{\theta}^{(1)},\ldots,\vec{\theta}^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\vec{\theta}^{(j)})^T \vec{x}^{(i)} - y(i,j) \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Features $\vec{x}^{(i)}$ are contents here. That's why it's content-based recommendation.

Collaborative Filtering

Now say you don't know $\vec{x}^{(i)}$, but somehow get an estimate of $\vec{\theta}^{(j)}$. Then you can also estimate $\vec{x}^{(i)}$.

Optimization (given $\vec{\theta}^{(1)}, \ldots, \vec{\theta}^{(n_u)}$) to learn $\vec{x}^{(i)}$:

$$\min_{\vec{x}^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} \left((\vec{\theta}^{(j)})^T \vec{x}^{(i)} - y(i,j) \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

To learn $\vec{x}^{(1)}, \ldots, \vec{x}^{(n_m)}$:

$$\min_{\vec{x}^{(1)},\ldots,\vec{x}^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left((\vec{\theta}^{(j)})^T \vec{x}^{(i)} - y(i,j) \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Now say you don't know $\vec{x}^{(i)}$ or $\vec{\theta}^{(j)}$ (neither of them). One option: go back and forth to optimize $\vec{x}^{(i)}, \vec{\theta}^{(j)}$ sequentially. Or you can do it simultaneously.

Cost function

$$J\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(n_m)}, \vec{\theta}^{(1)}, \ldots, \vec{\theta}^{(n_u)}\right) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left((\vec{\theta}^{(j)})^T \vec{x}^{(i)} - y(i,j) \right)^2$$

$$+ \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$$\min_{\vec{x}^{(1)},\ldots,\vec{x}^{(n_m)},\vec{\theta}^{(1)},\ldots,\vec{\theta}^{(n_u)}} J\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(n_m)}, \vec{\theta}^{(1)}, \ldots, \vec{\theta}^{(n_u)}\right)$$

Initialize $\vec{x}^{(i)}, \vec{\theta}^{(j)}$ randomly, then run the above optimization.

Prediction for New Movie: For a new movie for user j, use $(\vec{\theta}^{(j)})^T \vec{x}^{(i)}$ to predict rating (test phase). Recommend this movie if predicted rating is high.

Alternative Way

Group all ratings of all movies and users (n_u users, n_m movies) into a matrix:

$$\text{ratings} = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & 1 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

$? \Rightarrow$ means unrated
$n_m = 5,\ n_u = 4$ in this example

Let the actual ratings matrix be:

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 5 & 5 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

Predicted ratings:

$$\text{Predicted ratings} = \begin{bmatrix} (\vec{\theta}^{(1)})^T \vec{x}^{(1)} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \vdots & \cdots & (\vec{\theta}^{(n_u)})^T \vec{x}^{(n_m)} \end{bmatrix} = \Theta X^T$$

This collaborative filtering algorithm is also known as **low-rank matrix factorization**. This is a low-rank matrix.

Say, a user likes movie i. Recommend movie j related to movie i based on small $\|x^{(i)} - x^{(j)}\|_{L_2}$.

kNN (k-Nearest Neighbors) classifier

For kNN, Decision tree, and Random Forests, specifically, please refer to the hand written lecture.

Finds the k-nearest training data points. Assign class to the test data based on the class with maximum number of data points within that k-nearest training data points.

- Given $k = 3$, classify test data based on the majority class among k nearest training data points.
- Find the k nearest training data points. Assign the class label to the test data based on the maximum number of data points from a class within those k nearest neighbors.

Decision tree

$x_1, x_2, \cdots$ - features. $a_1, a_2, \cdots$ 0 constants.
Entropy - measure of randomness in the dataset.
Information gain - measure of decrease in entropy after the dataset split.
Leaf node - carries the classification or decision.
Decision node - has 2 or more branches.
root node - topmost decision node.

- Given features x_1, x_2 and constants a_1, a_2, the tree structure is:

$$\text{Is } x_1 > a_1? \begin{cases} \text{No} & \to \text{class 1} \\ \text{Yes} & \to \text{Is } x_2 = a_2? \begin{cases} \text{No} & \to \text{class 2} \\ \text{Yes} & \to \text{class 3} \end{cases} \end{cases}$$

With each decision, the dataset is split.

Definitions:

- **Entropy**: measure of randomness in the dataset.

- **Information Gain**: measure of decrease in entropy after the dataset split.
- **Leaf Node**: carries the classification or decision.
- **Decision Node**: has one or more branches.
- **Root Node**: topmost decision node.

Random forests

Random forests combine decision from many trees.

References

[1] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.