Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function.

To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point.

If instead one takes steps proportional to the *positive* of the gradient, one approaches a local maximum of that function; the procedure is then known as **gradient ascent**.

Gradient descent is also known as **steepest descent**. However, gradient descent should not be confused with the method of steepest descent for approximating integrals.

Gradient descent is a popular method in the field of machine learning because part of the process of machine learning is to find the highest accuracy, or to minimize the error rate, given a set of training data.

Gradient descent is used to find the minimum error by minimizing a "cost" function.

Gradient Descent for Machine Learning

Optimization is a big part of machine learning. Almost every machine learning algorithm has an optimization algorithm at its core.

In this post, you will discover a simple optimization algorithm that you can use with any machine learning algorithm. It is easy to understand and easy to implement. After reading this post you will know:

- What is gradient descent?
- How can gradient descent be used in algorithms like linear regression?
- How can gradient descent scale to very large datasets?
- What are some tips for getting the most from gradient descent?

Gradient Descent

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Intuition for Gradient Descent

Think of a large bowl like what you would eat cereal out of or store fruit in. This bowl is a plot of the cost function (f).

A random position on the surface of the bowl is the cost of the current values of the coefficients (cost).

The bottom of the bowl is the cost of the best set of coefficients, the minimum of the function.

The goal is to continue to try different values for the coefficients, evaluate their cost and select new coefficients that have a slightly better (lower) cost.

Repeating this process enough times will lead to the bottom of the bowl and you will know the values of the coefficients that result in the minimum cost.

Gradient Descent Procedure

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

coefficient = 0.0

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

cost = f(coefficient)

or

cost = evaluate(f(coefficient))

The derivative of the cost is calculated.

The derivative is a concept from calculus and refers to the slope of the function at a given point.

We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

delta = derivative(cost)

Now that we know from the derivative which direction is downhill, we can now update the coefficient values.

A learning rate parameter (alpha) must be specified that controls how much the coefficients can change on each update.

coefficient = coefficient – (alpha * delta)

This process is repeated until the cost of the coefficients (cost) is 0.0 or close enough to zero to be good enough.

You can see how simple gradient descent is.

It does require you to know the gradient of your cost function or the function you are optimizing, but besides that, it's very straightforward.

Next we will see how we can use this in machine learning algorithms.

Batch Gradient Descent for Machine Learning

The goal of all supervised machine learning algorithms is to best estimate a target function (f) that maps input data (X) onto output variables (Y). This describes all classification and regression problems.

Some machine learning algorithms have coefficients that characterize the algorithms estimate for the target function (f). Different algorithms have different representations and different coefficients, but many of them require a process of optimization to find the set of coefficients that result in the best estimate of the target function.

Common examples of algorithms with coefficients that can be optimized using gradient descent are Linear Regression and Logistic Regression.

The evaluation of how close a fit a machine learning model estimates the target function can be calculated a number of different ways, often specific to the machine learning algorithm. The cost function involves evaluating the coefficients in the machine learning model by calculating a prediction for the model for each training instance in the dataset and comparing the predictions to the actual output values and calculating a sum or average error (such as the Sum of Squared Residuals or SSR in the case of linear regression).

From the cost function a derivative can be calculated for each coefficient so that it can be updated using exactly the update equation described above.

The cost is calculated for a machine learning algorithm over the entire training dataset for each iteration of the gradient descent algorithm. One iteration of the algorithm is called one batch and this form of gradient descent is referred to as batch gradient descent.

Batch gradient descent is the most common form of gradient descent described in machine learning.

Stochastic Gradient Descent for Machine Learning

Gradient descent can be slow to run on very large datasets.

Because one iteration of the gradient descent algorithm requires a prediction for each instance in the training dataset, it can take a long time when you have many millions of instances.

In situations when you have large amounts of data, you can use a variation of gradient descent called stochastic gradient descent.

In this variation, the gradient descent procedure described above is run but the update to the coefficients is performed for each training instance, rather than at the end of the batch of instances.

The first step of the procedure requires that the order of the training dataset is randomized. This is to mix up the order that updates are made to the coefficients. Because the coefficients are updated after every training instance, the updates will be noisy jumping all over the place, and so will the corresponding cost function. By mixing up the order for the updates to the coefficients, it harnesses this random walk and avoids it getting distracted or stuck.

The update procedure for the coefficients is the same as that above, except the cost is not summed over all training patterns, but instead calculated for one training pattern.

The learning can be much faster with stochastic gradient descent for very large training datasets and often you only need a small number of passes through the dataset to reach a good or good enough set of coefficients, e.g. 1-to-10 passes through the dataset.

An analogy for understanding gradient descent

The basic intuition behind gradient descent can be illustrated by a hypothetical scenario.

A person is stuck in the mountains and is trying to get down (i.e. trying to find the minima).

There is heavy fog such that visibility is extremely low.

Therefore, the path down the mountain is not visible, so he must use local information to find the minima.

He can use the method of gradient descent, which involves looking at the steepness of the hill at his current position, then proceeding in the direction with the steepest descent (i.e. downhill).

If he was trying to find the top of the mountain (i.e. the maxima), then he would proceed in the direction steepest ascent (i.e. uphill).

Using this method, he would eventually find his way down the mountain.

However, assume also that the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have at the moment.

It takes quite some time to measure the steepness of the hill with the instrument, thus he should minimize his use of the instrument if he wanted to get down the mountain before sunset.

The difficulty then is choosing the frequency at which he should measure the steepness of the hill so not to go off track.

In this analogy, the person represents the backpropagation algorithm, and the path taken down the mountain represents the sequence of parameter settings that the algorithm will explore.

The steepness of the hill represents the slope of the error surface at that point.

The instrument used to measure steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point).

The direction he chooses to travel in aligns with the gradient of the error surface at that point.

The amount of time he travels before taking another measurement is the learning rate of the algorithm.

In numerical analysis, **hill climbing** is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing achieves optimal solutions in convex problems – otherwise it will find only local optima (solutions that cannot be improved by considering a neighbouring configuration), which are not necessarily the best possible solution (the global optimum) out of all possible solutions (the search space). Examples of algorithms that solve convex problems by hill-climbing include the simplex algorithm for linear programming and binary search. To attempt overcoming being stuck in local optima, one could use restarts (i.e. repeated local search), or more complex schemes based on iterations (like iterated local search), or on memory (like reactive search optimization and tabu search), or on memory-less stochastic modifications (like simulated annealing).

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems, so long as a small number of increments typically converges on a good solution (the optimal solution or a close approximation). At the other extreme, bubble sort can be viewed as a hill climbing algorithm (every adjacent element exchange decreases the number of disordered element pairs), yet this approach is far from efficient for even modest N, as the number of exchanges required grows quadratically.

Hill climbing attempts to maximize (or minimize) a target function **f(x)**, where **f(x)** is a vector of continuous and/or discrete values. At each iteration, hill climbing will adjust a single element in **x** and determine whether the change improves the value of **f(x)**. With hill climbing, any change that improves **f(x)** is accepted, and the process continues until no change can be found to improve the value of **f(x)**. Then x is said to be "locally optimal".

In discrete vector spaces, each possible value for **x** may be visualized as a vertex in a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of **f(x)**, until a local maximum (or local minimum) $\mathbf{x_m}$ is reached.

In **simple hill climbing**, the first closer node is chosen, whereas in **steepest ascent hill climbing** all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space which are not solutions. Steepest ascent hill climbing is similar to **best-first search**, which tries all possible extensions of the current path instead of only one.

Stochastic hill climbing does not examine all neighbors before deciding how to move. Rather, it selects a neighbor at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

Simulated annealing (**SA**) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to alternatives such as gradient descent.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Both are attributes of the material that depend on its thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy. The simulation of annealing as an approach that reduces a minimization of a function of large number of variables to the statistical mechanics of equilibration (annealing) of the mathematically equivalent artificial multiatomic system was first formulated by Armen G. Khachaturyan, Svetlana V. Semenovskaya, Boris K. Vainshtein in 1979[1] and by Armen G. Khachaturyan, Svetlana V. Semenovskaya, Boris

K. Vainshtein in 1981.[2] These authors used computer simulation mimicking annealing and cooling of such a system to find its global minimum.

This notion of **slow cooling** implemented in the Simulated Annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored (accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the optimal solution). The simulation can be performed either by a solution of kinetic equations for density functions or by using the stochastic sampling method. The method is an adaptation of the Metropolis–Hastings algorithm, a Monte Carlo method to generate sample states of a thermodynamic system, invented by M.N. Rosenbluth and published by N. Metropolis et al. in 1953.

Regression Model (Simple Linear Regression): Relates the response variable (also called the **Dependent Variable**) to **a quantitative explanatory variable** (also called the Independent Variable).

$$y = \beta_0 + \beta_1 x + \varepsilon$$

In the simple linear model, the average value of (also called the **expected value of**) is restricted to be 0 for a given value of x. This restriction indicates that the average (expected) value of the response variable y for a given value of x is described by a straight line:

$$E(y) = \beta_0 + \beta_1 x$$

This model is very restrictive because in many research settings a straight line does not adequately represent the relationship between the response and explanatory variable.

A linear equation may not be adequate to represent the relationships in many situations.

A **General Polynomial Regression Model** relating a dependent variable y to a single quantitative independent variable x is given by:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_p x^p + \varepsilon$$

with

$$E(y) = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_p x^p$$

The choice of p and hence the choice of an appropriate regression model will depend on the experimental situation.

The **Multiple Regression Model,** which relates a response variable y to a set of k quantitative explanatory variables, is a direct extension of the polynomial regression model in one independent variable. The multiple regression model is expressed as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon$$

Any of the k explanatory variables may be powers of the independent variables, for example, $x_3 = x_1^2$, or a **cross-product term,** $x_4 = x_1x_2$, or a nonlinear function such as $x_5 = log(x_1)$, and so on. For the above definitions we would have the following model:

$$\begin{aligned} y &= \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_3 + \beta_4x_4 + \beta_5x_5 + \varepsilon \\ &= \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_1^2 + \beta_4x_1x_2 + \beta_5log(x_1) + \varepsilon \end{aligned}$$

The simplest type of multiple regression equation is a **first-order model,** in which each of the independent variables appears, but there are no cross-product terms or terms in powers of the independent variables. For example, when three quantitative independent variables are involved, the first-order multiple regression model is

$$y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_3 + \varepsilon$$

For these first-order models, we can attach some meaning to the βs. The parameter β_0 is the y-intercept, which represents the expected value of y when each x is zero. For cases in which it does not make sense to have each x be zero, β_0 (or its estimate) should be used only as part of the prediction equation, and not given an interpretation by itself.

The other parameters ($\beta_1, \beta_2, \ldots, \beta_k$) in the multiple regression equation are sometimes called **partial slopes.** In linear regression, the parameter β_1 is the slope of the regression line and it represents the expected change in y for a unit increase in x. In a first-order multiple regression model, β_1 represents the expected change in y for a unit increase in x_1 *when all other xs are held constant*. In general then, $\beta_j (j \neq 0)$ represents the expected change in y for a unit increase in x_j while holding all other xs constant. The usual assumptions for a multiple regression model are shown here.

The **assumptions for multiple regression** are as follows:

1. The mathematical form of the relation is correct, so $E(\varepsilon_i) = 0$ for all i.
2. Var $(\varepsilon_i) = \sigma_\varepsilon^2$ for all i.
3. The ε_is are independent.
4. ε_i is normally distributed.

Sample problem on Multiple Regression

A brand manager for a new food product collected data on y = brand recognition (percent of potential consumers who can describe what the product is), x_1 = length in seconds of an introductory TV commercial, and x_2 = number of repetitions of the commercial over a 2-week period. What does the brand manager assume if a first-order model

$$\hat{y} = 0.31 + 0.042x_1 + 1.41x_2$$

is used to predict y?

Logistic Regression

In many research studies, the response variable may be represented as one of two possible values. Thus, the response variable is a binary random variable taking on the values 0 and 1.

When the response variable y is binary, the distribution of y reduces to a single value, the probability $p = \Pr(y = 1)$. We want to relate p to a linear combination of the independent variables. The difficulty is that p varies between zero and one, whereas linear combinations of the explanatory variables can vary between $-\infty$ and $+\infty$.

Transformation of probabilities into an odds ratio.

As the probabilities vary between zero and one, the odds ratio varies between zero and infinity.

By taking the logarithm of the odds ratio, we will have a transformed variable that will vary between __ and __ when the probabilities vary between zero and one.

The model often used to study the association between a binary response and a set of explanatory variables is given by **logistic regression analysis.**

In this model, the natural logarithm of the odds ratio is related to the explanatory variables by a linear model.

We will consider the situation where we have a single independent variable, but this model can be generalized to multiple independent variables.

Let $p(x)$ be the probability that y equals 1 when the independent variable equals x. We model the log-odds ratio to a linear model in x, a **Simple Logistic Regression Model:**

$$\ln\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x$$

This transformation can be formulated directly in terms of $p(x)$ as:

$$p(x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}}$$

The values of β_0 and β_1 would be estimated from the observed data using maximum likelihood estimation.

What happens in logistic regression is we have a bunch of data, and with the data we try to build an equation to do classification for us.

The regression aspects mean that we try to find a best-fit set of parameters.

Finding the best fit is similar to regression, and in this method, it's how we train our classifier.

We'll use optimization algorithms to find these best-fit parameters.

General approach to logistic regression

1. Collect: Any method.
2. Prepare: Numeric values are needed for a distance calculation. A structured data format is best.
3. Analyze: Any method.
4. Train: We'll spend most of the time training, where we try to find optimal coefficients to classify our data.
5. Test: Classification is quick and easy once the training step is done.
6. Use: This application needs to get some input data and output structured numeric values. Next, the application applies the simple regression calculation on this input data and determines which class the input data should belong to. The application then takes some action on the calculated class.

Logistic regression

Pros: Computationally inexpensive, easy to implement, knowledge representation easy to interpret

Cons: Prone to underfitting, may have low accuracy

Works with: Numeric values, nominal values

Heaviside step function is that at the point where it steps from 0 to 1, it does so instantly.

This instantaneous step is sometimes difficult to deal with.

There's another function that behaves in a similar fashion, but it's much easier to deal with mathematically.

This function is called the *sigmoid*.

The sigmoid is given by the following equation:

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

Two plots of the sigmoid are given in figure below.

At 0 the value of the sigmoid is 0.5.

For increasing values of x, the sigmoid will approach 1, and for decreasing values of x, the sigmoid will approach 0.

On a large enough scale (the bottom frame of figure below), the sigmoid looks like a step function.

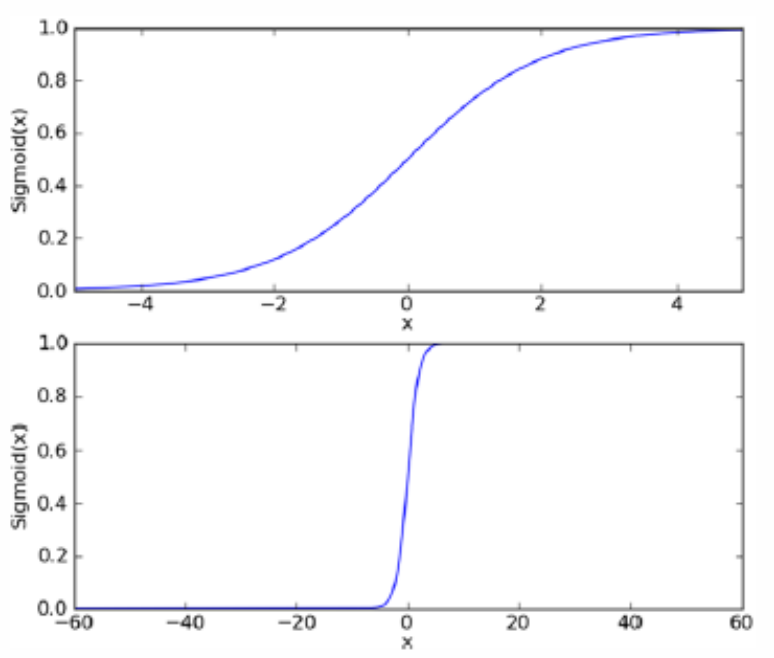


Figure: A plot of the sigmoid function on two scales; the top plot shows the sigmoid from -5 to 5, and it exhibits a smooth transition. The bottom plot shows a much larger scale where the sigmoid appears similar to a step function at x=0.

The question now becomes, what are the best weights, or regression coefficients to use, and how do we find them?

Using optimization to find the best regression coefficients

The input to the sigmoid function described will be z, where z is given by the following:

In vector notation we can write this as $z=w^Tx$.

All that means is that we have two vectors of numbers and we'll multiply each element and add them up to get one number.

The vector **x** is our input data, and we want to find the best coefficients **w**, so that this classifier will be as successful as possible.

Logistic regression

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x.

However, it is easy to construct examples where this method performs very poorly.

Intuitively, it also doesn't make sense for h_(x) to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$.

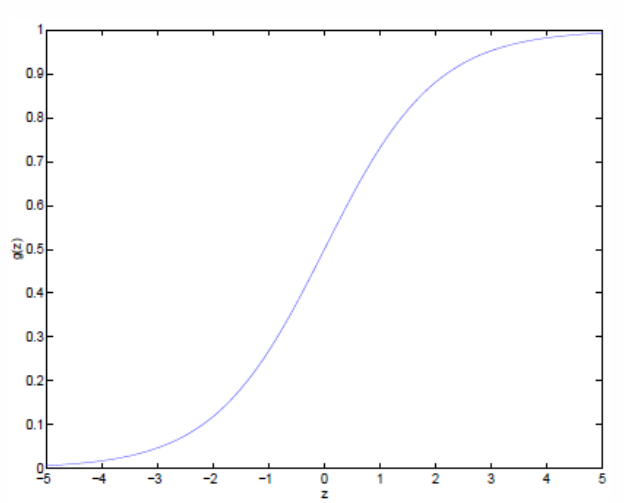
θ

To fix this, let's change the form for our hypotheses hθ(x). We will choose:

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

where

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



Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^{n} \theta_j x_j$.

A useful property of the derivative of the sigmoid function, which we write as g′:

$$\begin{aligned} g'(z) &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2} \left(e^{-z}\right) \\ &= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\ &= g(z)(1-g(z)). \end{aligned}$$

So, given the logistic regression model, how do we fit θ for it?

Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, let's endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood.

Let us assume that:

$$\begin{aligned} P(y = 1 \mid x; \theta) &= h_\theta(x) \\ P(y = 0 \mid x; \theta) &= 1 - h_\theta(x) \end{aligned}$$

This can be written more compactly as:

$$p(y \mid x; \theta) = (h_\theta(x))^y \left(1 - h_\theta(x)\right)^{1-y}$$

Likelihood

Let $X_1, X_2, ..., X_n$ have a joint density function $f(X_1, X_2, ..., X_n|\theta)$. Given $X_1 = x_1, X_2 = x_2, ..., X_n = x_n$ is observed, the function of θ defined by:

$$L(\theta) = L(\theta|x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_n|\theta) \tag{1}$$

is the *likelihood function*.

- The likelihood function is not a probability density function.
- It is an important component of both frequentist and Bayesian analyses
- It measures the support provided by the data for each possible value of the parameter. If we compare the likelihood function at two parameter points and find that $L(\theta_1|x) > L(\theta_2|x)$ then the sample we actually observed is more likely to have occurred if $\theta = \theta_1$ than if $\theta = \theta_2$. This can be interpreted as θ_1 is a more plausible value for θ than θ_2.

Assuming that the m training examples were generated independently, we can then write down the likelihood of the parameters as:

$$\begin{aligned} L(\theta) &= p(\vec{y} \mid X; \theta) \\ &= \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^{m} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}} \end{aligned}$$

As before, it will be easier to maximize the log likelihood:

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \end{aligned}$$

Python code for Logistic regression

```
print(__doc__)


import numpy as np

import matplotlib.pyplot as plt

from sklearn import linear_model, datasets


# import some data to play with

iris = datasets.load_iris()

X = iris.data[:, :2]  # we only take the first two features.

Y = iris.target


h = .02  # step size in the mesh


logreg = linear_model.LogisticRegression(C=1e5)


# we create an instance of Neighbours Classifier and fit the data.

logreg.fit(X, Y)


# Plot the decision boundary. For that, we will assign a color to each

# point in the mesh [x_min, x_max]x[y_min, y_max].

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5

y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])


# Put the result into a color plot

Z = Z.reshape(xx.shape)

plt.figure(1, figsize=(4, 3))
```

```
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())
plt.show()
```